POLITECNICO DI MILANO
School of Industrial and Information Engineering
Master of Science in Computer Science and Engineering
Dipartimento di Elettronica, Informazione e Bioingegneria

GAN-based Matrix Factorization for Recommender Systems

Supervisor: Prof. Paolo Cremonesi
Co-supervisor: Mario Scriminaci

Master Thesis by:
Ervin Dervishaj, 877556

Academic Year 2018-2019

Abstract

The last decade has seen an exponential increase in the amount of available information thanks to the ever-growing number of connected devices and interaction of users with online content like social media, e-commerce, etc. While this translates in more choices for users given their diverse set of preferences, it makes it difficult for them to explore this vast amount of information. Recommender systems (RS) aim to alleviate this problem by filtering the content offered to users by predicting either the rating of items by users or the propensity of users to like specific items. The latter is known as Top-N recommendation in the RS community and it refers to the problem of recommending items to users, preferably in the order from most likely-to-interact to least likely-to-interact.

RS use two main approaches for providing recommendations to users; collaborative filtering and content-based filtering. One of the main algorithms used in collaborative filtering is matrix factorization which constitutes in estimating the user preferences by decomposing a user-item interaction matrix into matrices of lower dimensionality of latent features of users and items.

The burst of big data has triggered a corresponding response in the machine learning community in trying to come up with new techniques to extract relevant information from data. One such technique is Generative Adversarial Nets (GAN) proposed in 2014 by Goodfellow et al. which initiated a fresh interest in generative modelling. Under this modelling paradigm, GANs have shown great results in estimating high-dimensional, degenerate distributions in Computer Vision, Natural Language Processing and various other scientific fields. Despite their popularity and abilities in learning arbitrary distributions, GANs, and more generally generative modelling, have not been widely applied in RS.

In this thesis we investigate a novel approach that estimates the user and item latent factors in a matrix factorization setting through the application of Generative Adversarial Networks for generic Top-N recommendation problem. We detail the formulation of this approach and show its

performance through different experiments on well know datasets in the RS community.

Sommario

Durante l'ultimo decennio c'è stato un aumento esponenziale nell'ammontare delle informazioni disponibili, questo grazie alla crescita percepita del numero di strumenti collegati e interazioni degli utenti con i contenuti online come i social media, e-commerce, etc. Anche se questo significa un aumento delle possibilità di scelta per gli utenti, dato le diverse preferenze personali, diventa comunque difficile l'esplorazione di tutti i dati e informazioni disponibili. A questo proposito nascono i Sistemi di Raccomandazione (SR), che hanno come obiettivo quello di ridurre questa problematica filtrando i contenuti offerti facendo una predizione sulla valutazione dei vari item da parte dell'utente o la propensione a preferire item specifici. Quest'ultimo viene riconosciuto nelle communita di SR anche come Raccomandazioni Top-N e si riferisce a problematiche di raccomandazioni di item agli utenti, preferibilmente nell'ordine di interazioni maggiori a quelle meno probabili.

Per poter offrire delle raccomandazioni all'utente i SR usano due approcci principali: quello collaborativo e l'approccio basato sul contenuto. Un terzo approccio, quello ibrido, può essere costruito dalla combinazione dei due precedenti. La fattorizzazione matriciale è uno degli algoritmi principali usati nell'approccio collaborativo e consiste nell'estimazione delle preferenze degli utenti basandosi sulla decomposizione della matrice dell'interazione utente-item in matrice di minori dimensioni di fattori latenti di utente e item.

L'esplosione dei big data ha influenzato un'analoga risposta nella communita di machine learning per quanto riguarda l'intento di trovare nuove tecniche ed estrarre le informazioni rilevanti dai data disponibili. Una simile tecnica è il Generative Adversial Net (GAN) proposta nel 2014 da Goodfellow et. al, che porto alla crescita di un interesse nei confronti dei modelli generativi. Sotto questo paradigma di modellazione, i GANs hanno dimostrato ottimi risultati nell'estimare distribuzioni multi-dimensionali e degenere in Visione Artificiale, Elaborazione del linguaggio naturale e altri vari campi scientifici. Nonostante la loro popolarità e abilità nell'imparare distribuzioni arbitrarie, i GANs e altri simili modelli, non sono stati implementati

ampiamente nell'ambito dei RS.

L'obiettivo principale di questa tesi è quello di esplorare l'applicabilità di un approccio nuovo nello stimare i fattori latenti di utente e item in una matrice fattoriale tramite l'utilizzo dei GANs per problemi di raccomandazione Top-N. La formulazione di questo approccio verrà descritto con dettaglio e la performance sarà presentata tramite vari esperimenti su dataset ben riconosciuti nell'ambito della comunita di SR.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Before the explosion of the Internet people used to get their movie recommendations from family and friends. Today it is as easy as signing up to Netflix and watching the first 5 minutes of a movie or series to get a recommendation. It was indeed Netflix that gave Recommender Systems (RS) the place it has nowadays in most of the products and services that we use. The Netflix Prize competition, a competition organized by Netflix in 2006, consisted in finding the best algorithm for estimating user ratings on various movies. RS is an information filtering technique [56] that allows the prediction and ranking of items for users in terms of preference. RS run underneath many popular websites; Amazon uses it to power its purchase suggestions, Facebook for friend discovery, Twitter for personalized feed of tweets. All these service providers tap into the information a RS provides them in order to offer a customized user experience.

The main reasons for the proliferation of RS is the availability of user-item data created during the interaction with RS-powered products and user-/item-specific metadata. These two types of data power the two main paradigms of techniques in RS: collaborative filtering (CF) and content-based filtering (CBF). Both CF and CBF have been and still are widely used to build recommender systems mainly because of their simplicity, explainability and straightforward implementation. They both have pros and cons and because of this they are usually combined in a hybrid RS in order to get the best of both paradigms.

In the context of filtering data for users, RS try to address two different but related problems – the first one is predicting as correctly as possible how a specific user would rate a set of items; the second one is constructing a ranked list from a set of items for a specific user in decreasing order of preference. This second problem is called the Top-N recommendation prob-

lem. This, due to the assumption that in most scenarios, users are mainly interested in the items that are ranked at the top of the list.

Over the years many different algorithms have been adopted and created for the two RS problems introduced above. One class of such algorithms is Matrix Factorization (MF). MF approaches are model-based; they construct a model from a user-item interaction matrix and use this model to predict ratings. By ordering the set of items according to these predicted ratings we can perform Top-N recommendations. MF models aim to represent users in a much lower dimensionality than that of the user-item interaction matrix by deriving latent factors for both items and users and predicting ratings from the multiplication of these factors. MF techniques in RS took off during the Netflix Prize competition and since then multiple derivatives have been developed.

The defining feature of MF is matrix multiplication which in itself is a linear operation. While powerful, this linearity can introduce some bottlenecks since the relation of users to items might be much more complex and only explained through some non-linear mapping. Such non-linearity can be easily achieved by using Multilayer Perceptrons (MLP) with non-linear activation functions. Deep Learning (DL), an extension of MLP-based Neural Networks to high number of hidden layers, introduced end-to-end training for Neural Networks and has been very successful in numerous tasks and fields like Computer Vision, Natural Language Processing, Signal Processing and recently also RS.

The success of DL has mainly been on discriminative modelling where a model is built to discriminate between samples and is able to assign new unseen data points the correct class. In 2014 Goodfellow et al. introduced Generative Adversarial Networks (GAN), a new framework built on top of Neural Networks for performing generative modelling. GANs have seen a great interest in the ML community for their ability to generate images from noise with a higher resolution than previous approaches. They do this by approximating the distribution of the training data and by allowing sampling from this learned distribution.

However, despite their clear presence in ML research, GANs have not been widely used in RS. In this thesis we show a GAN-based recommender system where we utilize the adversarial learning nature of GANs to learn the latent factors instead of learning them directly from the ratings data. Our work is based on the assumption that the representation in latent factors that MF builds for users/items can be taken as the preference of users on the latent factors and by analogy how much of these factors are present in each item. Such preference can be quite complex or even degenerate and

GANs are suited to estimate such distributions. We give the formulation and architecture of our approach and evaluate it across multiple datasets which are commonly used to test new RS algorithms.

Chapter 2

State of the Art

RS are software tools and techniques providing suggestions for users on items they might like [43]. In an ever-increasing number of choices in almost all areas of life, be that music, books, movies, news, travel, etc. it is difficult for users to explore new items that they might potentially like and enjoy. RS translate this challenge into two distinct problems; item rating prediction and item ranking. Both problems share characteristics but are fundamentally different.

In this chapter we dive into the generic formulation of a recommendation problem and relevant literature review. After covering preliminaries of RS we jump into one of the most important models, namely Matrix Factorization and finally end with literature review of Generative Adversarial Networks and its applications in the field of RS, the topic of this thesis.

2.1 Preliminaries of a Recommender System

RS use data in order to provide helpful suggestions to users. Usually this data comes in the form of historical user profiles, a record of items that a specific user has interacted in the past, and item/user metadata which are individual items'/users' information like for example actors in a movie, loudness of a song or user demographic data. Past user interactions, or feedback as they are also called, can be of two types: *explicit* or *implicit*.

Explicit feedback is the clear indication of how much a user is interested in an item [31]. This is usually expressed in terms of ratings or reviews and is a common approach followed by many service providers like Netflix and Amazon, where a user can specify how much they liked a movie they watched or write a review for an item they bought. The rating systems can vary among products and services but usually follow a scale from 1 to 5 or

from 1 to 10 (with 5 and 10 being the highest ratings respectively and 1 being the lowest rating) with a 0.5 scaling step.

While explicit feedback is straightforward it is also difficult to obtain since it involves a direct action from the user [31]. Implicit feedback on the other hand can be acquired indirectly throughout the usage of the RS by the user [40, 31]. Taking as example the case of a video-on-demand (VOD) service, a user selecting and watching a movie through this service can be considered as an implicit feedback signaling the system that the user showed some interest in the movie. Implicit feedback has a unary nature in that it only indicates interaction of users with items. This poses a difficult question regarding negative experiences from implicit feedback: if a RS does not have feedback from a user on a specific item, is it because the user did not like the item or because the user is not aware of the item at all?

The data consumed by RS defines the 3 different approaches that can be used to construct a RS:

- **Content-based filtering**: this approach provides as recommendation those items that are similar with items in a user's historical profile. In order to find similar items, item metadata are used. The reverse, recommending users to items by using user's metadata, can also be applied in this context.

- **Collaborative filtering**: the most important approach for RS which finds the most similar users to the current user and recommends items that those users have rated.

- **Hybrid**: a combination of collaborative filtering and content-based filtering.

2.1.1 Data Structures

Whichever the approach on top of which a RS is built, the data it consumes usually follow a 2-dimensional matrix structure. The 3 most common data matrices are the User Rating Matrix, the Item Content Matrix and the User Content Matrix:

- **User Rating Matrix** (URM): given a set of users U and a set of items I, the URM has a shape $|U| \times |I|$ where rows represent users and columns represent items and each cell (u, i) is the past feedback of user u on item i, hereafter written as r_{ui}. The items with which user u has interacted in the past is the set I_u and users that have rated item

i is the set U_i. If the feedback taken into account is implicit then we have:

$$r_{ui} = \begin{cases} 1 & \text{if user } u \text{ interacted with item } i \\ 0 & \text{otherwise} \end{cases}$$

If the feedback is explicit then r_{ui} is defined within minimum and maximum ratings. Usually users only interact with a small subset of I thus introducing a high degree of sparsity in the URM [33]. This matrix is key for CF approaches.

- **Item Content Matrix** (ICM): given a set of items I and a set of item metadata M, the ICM has a shape $|I| \times |M|$ where rows represent items and columns represent item metadata and each cell (i, m) is a value that indicates that metadata m is a characteristic of item i. Each cell of the ICM can either be binary, integer or real valued depending on the context of the metadata. For example if we consider the genre *action* as metadata m and the movie "Top Gun" as item i then the value of cell (i, m) in the ICM would be 1 since "Top Gun" is an action movie. For a romantic comedy j like "The Proposal", cell (j, m) would be 0.

  The ICM is usually used to build CBF approaches for RS.

- **User Content Matrix** (UCM): given a set of users U and a set of user metadata N, the UCM has a shape $|U| \times |N|$ where rows represent users and columns represent user metadata and each cell (u, n) indicates that metadata n is a characteristic of user u. Common user metadata are user demographics like age, sex, geographical location, etc. The UCM is also used to build CBF approaches for RS.

2.2 Recommender Systems Approaches

As discussed in 2.1 the data used defines the type of RS approach. In this section we briefly explore the main techniques for building RS.

2.2.1 Content-based Filtering (CBF)

CBF exploits side information on items and users to provide meaningful recommendations. There are two ways to construct a CBF RS, *item-based* CBF and *user-based* CBF. Starting from the ICM, CBF constructs a similarity between past items of a user and other items in I through the metadata the items share and recommends the ones with highest similarity. This similarity can be described by a $|I| \times |I|$ shaped matrix, the item-similarity matrix

SIM^I. Each cell (i, j) of SIM^I represents a similarity score between item i and item j. Once this similarity matrix is built, the estimated rating of a user u on a new item i can be computed as follows:

$$\hat{r}_{ui} = \frac{\sum_{j \in I_u} r_{uj} \times SIM^I_{i,j}}{\sum_{j \in I} |SIM^I_{i,j}|}$$

After sorting the items in decreasing order according to this estimated rating, the first N items can be recommended.

User-based CBF on the other hand uses the UCM. In the same manner we can construct a user-similarity matrix SIM^U where each cell (u, v) denotes a similarity between users u and v. The estimated rating of user u on a new item i can be computed as follows:

$$\hat{r}_{ui} = \frac{\sum_{j \in I_v} r_{vj} \times SIM^U_{uv}}{\sum_{v \in U} |SIM^U_{uv}|}$$

Again, it suffices to sort the predicted ratings of items in decreasing order and recommend the first N items.

CBF has some advantages with respect to CF. The most important one is the possibility for CBF to provide personalized recommendation for cold start users. This scenario is pretty common and represents a problem for most of CF techniques which we will discuss in sections 2.3 and 2.4. This usecase presents itself when we want to recommend items to users with few to no historical interactions. While CF solely depends on these interactions to provide meaningful recommendations, CBF can use the user-similarity matrix (assuming we have metadata on the users) to recommend highly rated items from those users that are most similar with the cold start user.

2.2.2 Collaborative Filtering (CF)

This thesis focuses on CF approaches. CF is the most widely used RS technique [43] and it is based on an underlying distance/similarity metric between users and items. As an example, given user u and his historical profile on a VOD service, we want to provide some personalized movie recommendations to u. If we happen to have another user v in the system that matches to a great extent the historical profile of u we can recommend to u movies that v has rated highly but u has not yet interacted with. CF depends entirely on the historical profiles of users/items so it needs the URM as input in order to provide recommendations. The way the URM is used inside CF derives the two variants available – *Memory-based* CF and *Model-based* CF.

2.2.3 Hybrid RS

A hybrid RS is the resulting combination of applying both CBF and CF approaches to solve a recommendation problem. Since the focus of this thesis is on CF we will not explore this approach further.

2.3 Memory-based CF

Memory-based CF approaches are some of the first techniques used in the context of CF [43]. They work by utilizing the complete historical profile of a specific user u when trying to find the user's *neighborhood*; a set of users that are most similar to u. Usually memory-based CF models are also called neighborhood models. By analogy one can also find the neighborhood of an item i.

2.3.1 User-based

User-based CF involves computing the neighborhood of a user u, that is a fixed number K of users that are most similar to u and use their ratings to inference the preferences of u on the set of items I. The underlying assumption is that if two users have rated the same items, they are more likely to prefer the same items. To generated recommendations we first compute the similarity matrix SIM_K^U which is a $|U| \times |U|$ shaped matrix where each row is sparse with only K real valued elements denoting the similarity of a user u to all other K users. Once we have this similarity matrix we can compute the rating of an item i by user u as follows:

$$\hat{r}_{ui} = \frac{\sum_{v \in U \wedge u \neq v} I_v[i] \times SIM_K^U[u, v]}{\sum_{v \in U} |SIM_K^U[u, v]|}$$

where $I_v[i]$ denotes element i in the historical profile of user v, $SIM_K^U[u, v]$ denotes the similarity between users u and v. For the final recommendation suffices to sort in decreasing order the ratings $\hat{r}_{ui}$ for each item i and recommend them.

2.3.2 Item-based

Item-based CF on the other hand computes first the neighborhood of an item i. Again, this means finding K most similar items for i which results in the matrix SIM_K^I with dimensions $|I| \times |I|$ where each row is sparse with only K real valued elements denoting the similarity of an item i to all other

K items. If items i and j are similar and user u has already rated i then we can infer his rating on j by the similarity of both items [2]:

$$\hat{r}_{ui} = \frac{\sum_{j \in I \wedge j \neq i} I_u[j] \times SIM_K^I[i,j]}{\sum_{j \in I} |SIM_K^I[i,j]|}$$

We can observe that for both user and item variants of memory-based CF the computation of the similarity between users and items is the key component in predicting the ratings of users on items they have not interacted with in the past. We come across the computation of such similarities also in CBF (section 2.2.1) so in the next section we review the main similarity functions that can be used to compute these matrices.

2.3.3 Similarities

As anticipated in sections 2.2.1 and 2.3 we need a way to compute whether two users/items are similar with one another. For CBF we have at our disposal the UCM/ICM and for CF the URM. Thus each user/item lives in n-dimensional space depending on the matrix used. The similarity functions we will describe all take as input two n-dimensional vectors and output a real valued scalar indicating the similarity between the vectors.

Cosine similarity

Cosine similarity measures the angle between two n-dimensional column vectors. Given $\mathbf{a}$ and $\mathbf{b}$ the cosine similarity between them is given by:

$$cosine(\mathbf{a}, \mathbf{b}) = \frac{\mathbf{a}^T\mathbf{b}}{\|\mathbf{a}\|\|\mathbf{b}\|}$$

with $\mathbf{a}^T\mathbf{b}$ denoting the dot product of the two vectors and $\|\cdot\|$ denoting the norm of a vector. The cosine of an angle can vary from -1 to 1 and thus also cosine similarity ranges in $[-1, 1]$ with -1 indicating inverse similarity (parallel unit vectors but in opposite directions), 1 complete similarity (parallel unit vectors) and 0 indicating orthogonality (no similarity).

Pearson Correlation

Pearson Correlation is a metric used to compute the linear correlation between two random variables. Given such two variables X and Y their Pearson Correlation is:

$$\rho_{X,Y} = \frac{cov(X, Y)}{\sigma_X \sigma_Y}$$

Since in our usecase we want to compute the correlation between two vectors, we can use the Pearson Correlation in the context of a population sample, denoted by $r_{\mathbf{a},\mathbf{b}}$:

$$r_{\mathbf{a},\mathbf{b}} = \frac{\sum_{i=1}^{n}(\mathbf{a}_i - \bar{\mathbf{a}})(\mathbf{b}_i - \bar{\mathbf{b}})}{\sqrt{\sum_{i=1}^{n}(\mathbf{a}_i - \bar{\mathbf{a}})^2}\sqrt{\sum_{i=1}^{n}(\mathbf{b}_i - \bar{\mathbf{b}})^2}}$$

where $\bar{\mathbf{a}}$ and $\bar{\mathbf{b}}$ are the respective means of vectors $\mathbf{a}$ and $\mathbf{b}$. Just like Cosine similarity also Pearson Correlation coefficient ranges in $[-1, 1]$ with -1 indicating total negative linear correlation, 1 total positive linear correlation and 0 indicating no correlation at all.

Jaccard Coefficient

Jaccard coefficient is a set theoretic measure used to compute the similarity between two finite sets. The coefficient is computed as follows:

$$J(A, B) = \frac{|A \cap B|}{|A| \cup |B|} = \frac{|A \cap B|}{|A| + |B| - |A \cap B|}$$

where A and B are two finite sets and $|\cdot|$ is the set cardinality operator.

Tanimoto Coefficient

The Jaccard coefficient cannot be applied as-is to the RS usecase because it is conceived for sets. Tanimoto coefficient [48, 44] on the other hand is a form of Jaccard coefficient applicable to vectors:

$$T(X, Y) = \frac{\sum_i X_i \wedge Y_i}{\sum_i X_i \vee Y_i}$$

This coefficient is especially useful for similarity between users/items when the feedback is implicit. Taking as example two different users identified by their implicit historical profiles we can translate the meaning of Tanimoto coefficient as the ratio between items rated by both users versus items rated by each user separately.

2.4 Model-based CF

Different from memory-based CF, model-based CF uses the URM to build a model which can be then used to provide recommendations to users. Such

models are usually built using data mining and/or machine learning approaches. We review in this section SLIM, a machine learning based technique to learning the item-item similarity matrix SIM^I, and Matrix Factorization as a more general framework of model-based CF.

2.4.1 SLIM

SLIM stands for Sparse Linear Methods [39]. It was introduced as a model for the Top-N recommendation problem. Despite building a model it mimics more the working of a memory-based CF technique. SLIM creates a model of the item-item similarity matrix which it learns through the following optimization criteria:

$$\min_{SIM^I} \frac{1}{2}\|URM - URM \times SIM^I\|_F^2 - \frac{\beta}{2}\|SIM^I\|_F^2 + \lambda\|SIM^I\|_1$$
$$\text{subject to} \quad SIM^I \geq 0, \quad \text{diag}(SIM^I) = 0$$

where $\|\cdot\|_F$ is the Frobenius norm of a matrix and acts as a regularizer on the matrix, $\|\cdot\|_1$ is the l_1-norm by which sparsity is promoted and $diag(\cdot)$ is the main diagonal of a matrix. This optimization is conditioned on $SIM^I \geq 0$ since the similarity between two items cannot be negative and $\text{diag}(SIM^I) = 0$ because we want to avoid trivial solutions where the similarity matrix is the identity matrix.

Once SLIM learns the similarity matrix it predicts the rating on an item by as user in the following way:

$$\hat{r}_{ui} = \sum_{j \in I_u} r_{uj} SIM^I_{ij}$$

One can observe that the rating prediction by SLIM is similar to item memory-based CF and indeed they differ in the way the item-item similarity matrix is constructed. Finally according to the predicted ratings we can order in a decreasing order the items with which the user has not interacted in the past and recommend the first N of them.

2.4.2 Matrix Factorization

Matrix factorization is the operation of decomposing a matrix into the multiplication of two other matrices. It became quite popular during the Netflix Prize competition where the now famous Funk SVD [18] approach resulted in very good performance compared to the other more traditional solutions.

In the context of RS, MF plays an important role as a generic framework on top of which multiple algorithms are built. The two matrices resulting from MF are considered in RS as *latent factors* of users and items (figure 2.1).

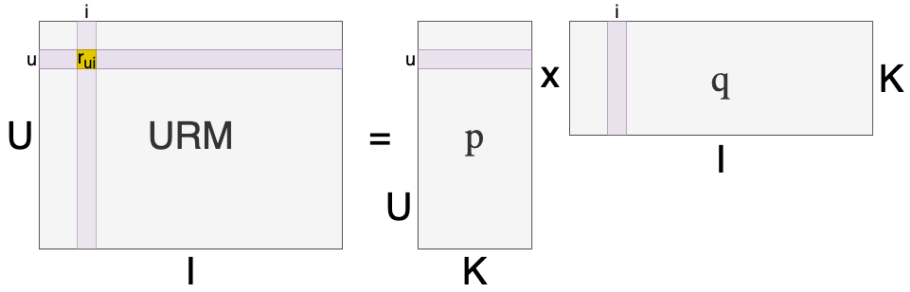


Figure 2.1: Matrix factorization in RS.

In RS, a perfect reconstruction of the URM matrix is not very useful because of the sparsity of the URM. If the resulting matrices reconstruct zeros for non-rated items then it is impossible to recommend new items from this approach. For this reason in RS almost always a truncated reconstruction of the URM is desirable. In this case MF brings both users and items onto a fixed dimensional space [31], the latent space, and the rating of a user u on an item i, r_{ui}, is derived from the inner product of the corresponding latent factors:

$$\hat{r}_{ui} = \mathbf{p}_u^T \mathbf{q}_i \qquad \mathbf{p} \in \mathbb{R}^{K \times |U|}, \mathbf{q} \in \mathbb{R}^{K \times |I|}, K \ll \min(|U|, |I|)$$

We describe next some of the main algorithms for MF in RS. These algorithms were mainly used for explicit feedback but then later adapted for implicit feedback too.

FunkSVD

FunkSVD [18] was created by Simon Funk in 2006 during the Netflix Prize competition. It is an MF approach where the latent factors of users and items are learned by optimizing the following regularized loss function:

$$\min_{\mathbf{p},\mathbf{q}} \sum_{(u,i) \in \mathcal{O}^+} (r_{ui} - \mathbf{p}_u^T \mathbf{q}_i)^2 + \lambda_p \|\mathbf{p}_u\|^2 + \lambda_q \|\mathbf{q}_u\|^2$$

where $\mathcal{O}^+$ is the set user-item pairs for which we know the rating. From the loss function we can see that FunkSVD learns latent factors in such a way as to reduce the Mean Squared Error (MSE) between the predicted and actual ratings in the URM. However following this loss function resulted in high

overfitting so additional regularizers were included in order help alleviate this problem. FunkSVD uses stochastic gradient descent with the following update rules:

$$\mathbf{p}_{uk} \longleftarrow \mathbf{p}_{uk} + \alpha[(r_{ui} - \mathbf{p}_u^T \mathbf{q}_i)\mathbf{q}_{ik} - \lambda_p \mathbf{p}_{uk}] \quad k \in 1 \dots K$$
$$\mathbf{q}_{ik} \longleftarrow \mathbf{q}_{ik} + \alpha[(r_{ui} - \mathbf{p}_u^T \mathbf{q}_i)\mathbf{p}_{uk} - \lambda_q \mathbf{q}_{ik}] \quad k \in 1 \dots K$$

PureSVD

PureSVD [15] was introduced by Cremonesi et al. in 2009 as a solution to Top-N recommendation. It relies on the ubiquitous **Singular Value Decomposition** (SVD) to get the factorization of the URM. SVD reconstructs the URM as follows:

$$U\hat{R}M = \mathbf{M}\mathbf{\Sigma}\mathbf{N}^T$$

where $\mathbf{M}$ is a $|U| \times K$ real orthonormal matrix, $\mathbf{\Sigma}$ is a $K \times K$ matrix containing only the top K singular values and $\mathbf{N}$ is a $|I| \times K$ real orthonormal matrix. By defining $\mathbf{p} = \mathbf{M}\mathbf{\Sigma}$ we retrieve the user latent factors and with $\mathbf{q} = \mathbf{N}^T$ we retrive the item latent factors. In order to apply SVD the authors in [15] impute the missing ratings in the URM with zeros and say that in using PureSVD even imputation with other values would not affect the performance of the algorithm. PureSVD is one of the strongest baselines for Top-N recommendation problem. Its drawbacks are mostly related to computational issues; even though the URM is sparse PureSVD performs the decomposition into dense matrices thus making optimized libraries that deal with sparse matrices not usable and this usually brings memory issues.

Weighted Regularized Matrix Factorization

Weighted Regularized Matrix Factorization (WRMF) [27] is a MF technique that is adapted for implicit feedback and uses *alternating-least-squares* (ALS) [6] as a training methodology. It works by first binarizing the URM (if not already binary):

$$x_{ui} = \begin{cases} 1 & r_{ui} > 0 \\ 0 & r_{ui} = 0 \end{cases}$$

Then it assigns a confidence value c_{ui} to each of the implicit ratings:

$$c_{ui} = 1 + \alpha r_{ui}$$

This confidence value is a way to quantify how confident the recommendation engine is about the preference of user u on item i. Finally WRMF optimizes the following function:

$$\min_{\mathbf{p},\mathbf{q}} \quad \sum_{u,i} c_{ui}(x_{ui} - \mathbf{p}_u^T \mathbf{q}_i)^2 + \lambda(\sum_u \|\mathbf{p}_u\|^2 + \sum_i \|\mathbf{q}_i\|^2)$$

where $\lambda(\sum_u \|\mathbf{p}_u\|^2 + \sum_i \|\mathbf{q}_i\|^2)$ is a regularizer that helps avoid overfitting. In order to train the model and learn the latent factors we can apply ALS by considering one of the latent factors matrices as constant, one at a time, and optimizing for the other matrix. When fixing one of the matrices in this fashion we can derive closed form solution for the other matrix. First the authors set C^u to be a diagonal matrix of shape $|I| \times |I|$ with $C^u_{ii} = c_{ui}$. Then by fixing $\mathbf{q}$ we can get each user's latent factors by solving analytically:

$$\mathbf{p}_u = (\mathbf{q}^T C^u \mathbf{q}) + \lambda I)^{-1} \mathbf{q}^T C^u x_u \tag{2.1}$$

In the same way, by setting C^i to be a diagonal matrix of shape $|U| \times |U|$ with $C^i_{uu} = c_{ui}$ and fixing $\mathbf{p}$ we can solve for item latent factors analytically:

$$\mathbf{q}_i = (\mathbf{p}^T C^i \mathbf{p}) + \lambda I)^{-1} \mathbf{p}^T C^i x(i) \tag{2.2}$$

where $x(i)$ is the vector denoting the preferences of all users on item i. The training is performed by updating iteratively for each user equation 2.1 and for each item equation 2.2 until both matrices of latent factors stabilize.

Bayesian Personalized Ranking

Bayesian Personalized Ranking (BPR) [42] is a generic optimization criteria that explicitly tackles personalized ranking. By generic it is meant that it can be used with different RS approaches because it only defines the final optimization objective. BPR works based on the assumptions that users like equally items that they have interacted in the past and also like any item they have interacted in the past more than items they have not interacted yet. Putting this in notation BPR says:

$$I_u^+ >_u I_u^-$$

where I_u^+ denotes items user u has interacted with in the past, I_u^- denotes items user u has not interacted with yet and $>_u$ is the preference operation that implies that user u likes item on the left more than item on the right. Starting from the URM we can build the following training set of size $|U| \times |I| \times |I|$:

$$D_s = \{(u, i, j) \,|\, i \in I_u^+ \wedge j \in I_u^-\}$$

where the pairs (u, i, j) are such that they satisfy the operator $>_u$.

Given parameters Θ of a RS model class, we can learn the personalized ranking of items by maximizing the following posterior:

$$p(\Theta| >_u) \propto p(>_u |\Theta)p(\Theta)$$

Assuming that the ordering imposed by $>_u$ is independent for each user u we can rewrite the likelihood as a product over all users:

$$\prod_{u \in U} p(>_u |\Theta) = \prod_{(u,i,j) \in |U| \times |I| \times |I|} p(i >_u j|\Theta)^{\mathbb{1}\{(u,i,j) \in D_s\}} (1 - p(i >_u j|\Theta))^{\mathbb{1}\{(u,i,j) \notin D_s\}}$$

where $\mathbb{1}$ is the indicator function. We can simplify the above formula to:

$$\prod_{u \in U} p(>_u |\Theta) = \prod_{(u,i,j) \in |U| \times |I| \times |I|} p(i >_u j|\Theta)$$

under the following two properties that $>_u$ satisfies:

$$\forall i, j \in I : i \neq j \implies i >_u j \vee j >_u i \quad \text{(totality)}$$
$$\forall i, j \in I : i >_u j \wedge j >_u i \implies i = j \quad \text{(antisymmetry)}$$

Finally we describe $p(i >_u j|\Theta)$ as the probability that user u likes item i between i and j by setting:

$$p(i >_u j|\Theta) := \sigma(\hat{x}_{uij}(\Theta))$$

where $\hat{x}_{uij}$ describes how is item i related to item j with regards to user u and $\sigma(\cdot)$ is the sigmoid function.

The above steps describe the likelihood. For the prior we can assume a normal distribution with mean zero and variance-covariance matrix $\Sigma_\Theta = \lambda_\Theta I$:

$$p(\Theta) = N(0, \Sigma_\Theta)$$

We can now describe the optimization criteria denoted as BPR-OPT:

$$
\begin{aligned}
\text{BPR-OPT} : &= \ln(p(\Theta| >_u)) \\
&= \ln(p(>_u |\Theta)p(\Theta)) \\
&= \ln \prod_{(u,i,j)\in D_s} \sigma(\hat{x}_{uij} p(\Theta) \\
&= \sum_{(u,i,j)\in D_s} \ln \sigma(\hat{x}_{uij}) + \ln p(\Theta) \\
&= \sum_{(u,i,j)\in D_s} \ln \sigma(\hat{x}_{uij}) - \lambda_\Theta \|\Theta\|^2
\end{aligned}
$$

2.4.3 Evaluation

Just like other ML subfield, we can understand how good the recommendation is only through specific evaluation. Evaluating RS represents a difficult task [26] because of the final aim of the recommendations. Considering the two recommendation problems, item rating prediction and item ranking, they require evaluation with different metrics. On the other hand also the different type of feedback needs to be accounted when selecting the metrics through which to perform the evaulation. More importantly different domains require and give birth to different metrics [12].

Almost all recommendation engines try to optimize their suggestions such that the user can find his/her preferred item on the top of the suggestions. It makes for a poor user experience if a user gets to find an item he/she prefers only after scrolling multiple pages of recommendations. This gives rise to evaluation on a specific recommendation list length. In this thesis we use a combination of accuracy and ranking metrics in order to understand how good our proposed model behaves on different datasets and on different recommendation list cutoffs.

Accuracy Metrics

Accuracy metrics are used to make a comparison between the users' true preferences on items and the items that the RS suggests. This comparison is done per user and given as a total average. Given the two types of feedback in RS we have different accuracy concept for each of them. If the ratings are explicit then accuracy means for the RS to be able to predict the ratings that users have given to items they have interacted with. On the other hand if the feedback is implicit, accuracy takes the meaning of classification; whether the RS is able to classify items as an item that the user would interact with or not.

Explicit feedback accuracy metrics When predicting items' ratings, usually metrics that compute a deviation of the prediction from the ground truth are used. Some of the most common are Mean Absolute Error (MAE), Mean Squared Error (MSE) and Root Mean Square Error(RMSE):

$$\text{MAE} = \frac{\sum_{i=1}^{|I|} |r_{ui} - \hat{r}_{ui}|}{|I|}$$

$$\text{MSE} = \frac{\sum_{i=1}^{|I|} (r_{ui} - \hat{r}_{ui})^2}{|I|}$$

$$\text{RMSE} = \sqrt{\text{MSE}} = \sqrt{\frac{\sum_{i=1}^{|I|} (r_{ui} - \hat{r}_{ui})^2}{|I|}}$$

where $\hat{r}_{ui}$ is the predicted rating of user u on item i. These metrics are only used with explicit ratings and since in this work we deal only with implicit feedback we do not explore them any further.

Implicit feedback accuracy metrics We now go over some of the main accuracy metrics used for RS with implicit feedback. Usually when dealing with only implicit feedback we are interested in the ranking of the recommended items and not their absolute ratings. In such cases accuracy takes more the meaning of classification. When evaluating on a holdout set, we want a RS model to be able to classify items in the hold out set as items that a user would like to interact with in the future. So the problem is shifted to a classification problem. Such classification problem in RS has only two classes; an item is either *relevant* for a user or *irrelevant*. As is usual for classification tasks we can build a confusion matrix as shown on table 2.1.

	Recommended	Not Recommended
Relevant	True Positives (TP)	False Negatives (FN)
Irrelevant	False Positives (FP)	True Negatives (TN)

Table 2.1: Confusion matrix for RS.

The values in the confusion matrix have the following meaning:

- True positives (TP): number of items that are relevant for the user and were also recommended by the RS.
- False negatives (FN): number of items that are relevant for the user but were not in the list of recommendations.
- False positives (FP): number of items that are irrelevant for the user but the RS included them in the suggestions.

- True negatives (TN): number of items irrelevant for the user and also missing from the provided recommendations.

Two very common and popular classification metrics are **precision** and **recall**. Precision measures the ratio of the relevant recommended items to the total number of recommended items. Recall on the other hand measures the ratio of relevant items included in the recommendations to the total number of relevant items of a specific user. Formally they are expressed as:

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

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

Naturally if we were to recommend to users a recommendation list equal to the number of items in the catalogue then FN would be 0 and we would have perfect recall. But this is a trivial solution and something that brings poor user experience. Users are interested only on the top results of the list of recommendations [15] and hence it makes sense to be able to recommend relevant items at the top of the list. The notion of cutoff can thus be introduced for the classification metrics presented above. Given a cutoff of K, precision@K and recall@K retain the same meaning as before but only for the top K items of the recommendation list. So in the case of precision@K we are checking how many of the first K items recommended are actually relevant items for the user and in the case of recall@K we are checking how many of all the relevant items for a user are in the first K recommended items.

Ranking metrics As mentioned in the previous section, users are usually interested in seeing relevant items at the top of the list. This implies that an important aspect of the recommendation is the final ranking of items presented to the user. To measure the fitness of the ranking of recommendations specific metrics are used [26], different from the classification ones since they are not enough to measure RS from the viewpoint of ranking. Consider a RS model that suggests 10 items for a user and only 2 out of 10 are relevant, placed randomly in the list of 10 items. Consider a second RS model that again suggests 10 items out of which only 2 are relevant but these are placed 1^{st} and 2^{nd} in the list of 10. If we consider only a metric like precision, both model will result in equal accuracy. However the second model can rank the relevant items at the top of the list and this model is preferable to the first one.

The following 2 ranking metrics are widely used in the RS community:

- **Mean Average Precision (mAP)**: mAP is one of the most important ranking metrics for RS and IR alike [35]. It is based on the metric *average-precision*(AP) which is usually measured on a cutoff K:

$$\text{AP@K} = \frac{\sum_{i=1}^{K} precision@i \cdot \text{rel}(r)}{\#\text{ relevant items}}$$

  where rel(i) is the indicator function:

$$\text{rel}(i) = \begin{cases} 1 & \text{if item at position } i \text{ is relevant} \\ 0 & \text{otherwise} \end{cases}$$

  AP measures the average precision for each cutoff value from 1 to K if the item at that cutoff is relevant. Finally we can get mAP@K as the mean of average-precision@K for all users:

$$\text{mAP@K} = \frac{\sum_{u \in U} \text{AP}_u\text{@K}}{|U|}$$

- **Normalized Discounted Cumulative Gain (nDCG)** [53]: DCG is a metric that measures a discounted weighted sum of the relevancy of each item from the top of the recommendation list until a cutoff K. Given again the relevancy indicator defined for mAP, rel(i), DCG is given as:

$$\text{DCG@K} = \sum_{i=1}^{K} \text{rel}(i) D(i)$$

  where D(i) is the discount factor, usually $D(i) = \frac{1}{\log(1+i)}$. Finally DCG is normalized with the *ideal* DCG:

$$\text{idealDCG@K} = \sum_{i=1}^{K} D(i)$$

$$\text{nDCG@K} = \frac{\text{DCG@K}}{\text{idealDCG@K}}$$

Beyond accuracy metrics Recommending to users items that are preferred by everyone does not provide the best user experience [26, 15]. So it is important for a model to be able to recommend items that capture the true preferences of users. Recommending only popular items results in a non-personalized experience even though the recommendations as measured by the the previous metrics might show great performance, especially

in datasets where ratings are highly biased towards popular items. This creates the need of evaluating RS model with other metrics beyond the ones of accuracy and ranking. One such metric is item coverage [19]. It is defined as the ratio of the total number of unique items recommended to all users versus the total number of items in the catalogue. More formally, assuming that to user u the set of R_u of items is recommended then the coverage for user u is:

$$\text{coverage(u)} = \frac{|R_u|}{|I|}$$

The coverage of a RS model is then the average of coverage(u) over all the users:

$$\text{coverage} = \frac{\sum_u \text{coverage(u)}}{|U|}$$

This metric allows one to understand how good a recommender system is able to address the popularity bias in a dataset. If the coverage of the model is high along side other metrics that measure accuracy then one can deduce that the model has a high accuracy while at the same time making sure to recommend to users a large array of different items and not only a small subset of the catalogue.

2.5 Generative Modelling

In this section and section 2.6 we investigate the application of generative modelling in RS. Generative modelling falls under the field of statistics and machine learning as one of the major paradigms of modelling a specific problem of classification/regression, the other one being discriminative modelling [37, 28]. Considering a classification task, discriminative classifiers model the conditional class distribution $p(y|x)$ of input variables x and label y by minimizing an appropriate loss function. By optimizing such functions discriminative modelling builds classifiers that are less domain-specific and that have only one objective, assigning labels as correctly as possible [28]. Taking Support Vector Machines (SVM)[14] as an example of a discriminative model, they aim to maximize the margin between the decision boundary between the sets of points belonging to 2 different classes. Generative modelling on the other hand learns a joint probability distribution $p(x, y)$. Given this learned joint distribution, Bayes' rule can be used to derive the conditional class distribution $p(y|x)$ as in the case of discriminative modelling:

$$p(x, y) = p(x|y)p(y) = p(y|x)p(x) \implies p(y|x) \propto p(x|y)p(y)$$

So by modelling the joint distribution of the variables and the label we retrieve much more information about the problem at hand. Such modelling provides also the conditional distribution of the variables given the label $p(x|y)$ which translates in the ability of generating new data belonging to that label. Hence the name *generative* modelling. The possibility of generating new data is essential for domains where getting new data is very expensive or not even possible at all. The other very important benefit of generative modelling is the opportunity to provide information to the model through a prior distribution over the variables. This is very handy in cases when for a specific usecase we have a domain expert that can drive the modelling process via this prior. Despite these advantages, generative modelling has also some drawbacks. First it requires additional work especially in the cases when we are interested only in the conditional probability of the label given the variables. For such cases it is usually better to use discriminative models since they tend to provide also better results [37, 28]. Second, generative modelling might require more data to build a sound probabilistic interpretation of the problem whereas a discriminative model might need fewer data to model only the conditional distribution.

2.5.1 Generative Adversarial Networks

In this thesis we focus on a specific type of generative models called Generative Adversarial Networks (GAN) [22]. They were invented by Goodfellow et. al in 2014 and since then they have found their way to multiple fields like computer vision, natural language processing, speech processing, malware detection, etc. [52]. In figure 2.2 we give a taxonomy of generative models as given by [21]. In this taxonomy, GANs are part of a class of generative models called implicit density estimating generative models.

In a nutshell a large amount of generative models are built on the framework of **Maximum Likelihood Estimation** (MLE). Such models estimate a probability distribution through a fixed representation parameterized by some parameters θ. The task of the model is to learn these parameters θ such that the likelihood of the training data is maximized. Assuming the distribution as represented by the model is $p_{model}(\theta)$ and the training dataset contains m data points $x^{(i)}$, then by MLE we have [21]:

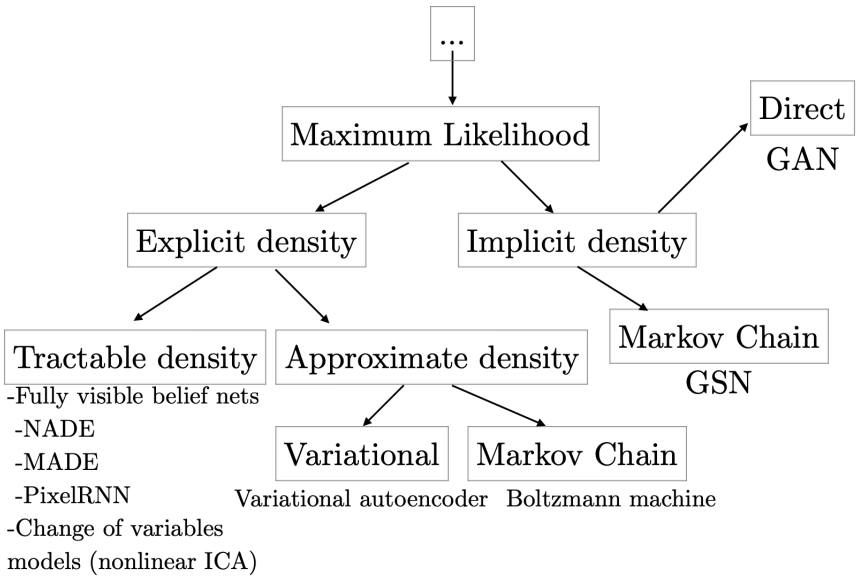


Figure 2.2: Taxonomy of generative models [21].

$$\begin{aligned}
\theta^* &= \arg\max_\theta \prod_{i=1}^{m} p_{model}\big(x^{(i)}, \theta\big) \\
&= \arg\max_\theta \log \prod_{i=1}^{m} p_{model}\big(x^{(i)}, \theta\big) \quad (\arg\max_z f(z) = \arg\max_z \log f(z)) \\
&= \arg\max_\theta \prod_{i=1}^{m} \log p_{model}\big(x^{(i)}, \theta\big)
\end{aligned}$$

In order to solve for θ we require the representation of the distribution induces by p_{model}. Such models are depicted in the left site of the taxonomy tree in figure 2.2. They explicitly define a density function. A major drawback of these models is the necessity for carefully defining p_{model} which usually required domain-specific knowledge. This becomes difficult when the problem to be modelled is characterized by very high-dimensional data, like in the case of computer vision or RS.

GANs on the other hand are part of another branch in the taxonomy tree. They build a model representative of the distribution to be modelled but they do this implicitly; this means that we do not have a fixed form of the distribution like for example a normal distribution which has a certain shape (which changes based on its parameters) but a model of the distribution with which we can interact in a non direct way [21] like for example, sampling

from it.

GAN Framework

GANs are built on top of a minimax zero-sum game played by two players. One of them is the **generator** and the other the **discriminator**. The generator's task is to generate data that look as closely as possible to the real training data. The discriminator's task is to differentiate between the real training data and synthetic data coming from the generator. The generator is trained so that it can fool the discriminator into classifying the data it generates as real data. Both the generator, G, and discriminator, D, are neural networks, differentiable on their input and parameters so the whole GAN can be trained by backpropagation [45]. We give a visual representation of GAN in figure 2.3.

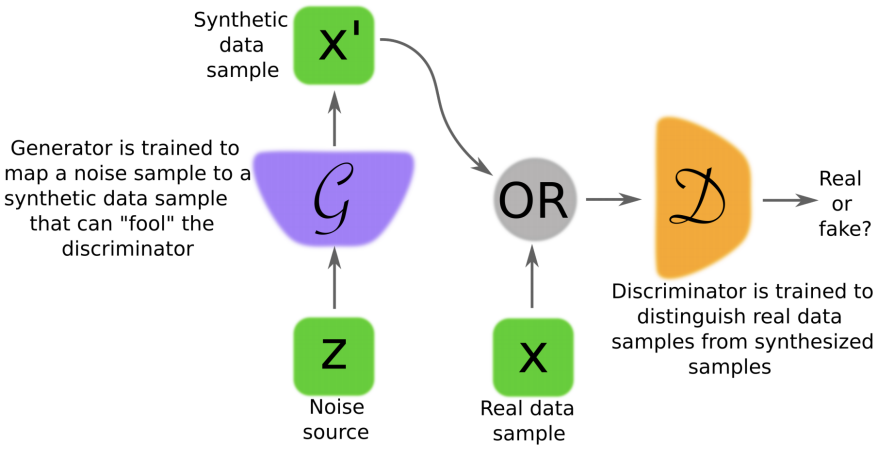


Figure 2.3: A depiction of a GAN [16].

The generator network in GAN does not see the real training data during the whole process of using a GAN. It takes as input a noise vector $\mathbf{z}$ sampled from a prior distribution $p_{\mathbf{z}}$, known as the latent space, and produces a synthetic data point in the real data space. So the function of network G is $\mathcal{G} : \mathcal{G}(\mathbf{z}) \rightarrow \mathbb{R}^{|\mathbf{x}|}$ where $|\mathbf{x}|$ is the dimensionality of the real data. The discriminator D on the other hand is simply a binary classifier and it trained through supervised learning in distinguishing among real and "fake" data. Its function is $\mathcal{D} : \mathcal{D}(\mathbf{x}) \rightarrow (0, 1)$ [16].

Training of GAN is done through simultaneous stochastic gradient descent and is show in figure 2.4. The cost function each of the networks

Algorithm 1 Minibatch stochastic gradient descent training of generative adversarial nets. The number of steps to apply to the discriminator, k, is a hyperparameter. We used $k = 1$, the least expensive option, in our experiments.

for number of training iterations **do**

for k steps **do**

- Sample minibatch of m noise samples $\{\boldsymbol{z}^{(1)}, \ldots, \boldsymbol{z}^{(m)}\}$ from noise prior $p_g(\boldsymbol{z})$.
- Sample minibatch of m examples $\{\boldsymbol{x}^{(1)}, \ldots, \boldsymbol{x}^{(m)}\}$ from data generating distribution $p_{\text{data}}(\boldsymbol{x})$.
- Update the discriminator by ascending its stochastic gradient:

$$\nabla_{\theta_d} \frac{1}{m} \sum_{i=1}^{m} \left[\log D\left(\boldsymbol{x}^{(i)}\right) + \log\left(1 - D\left(G\left(\boldsymbol{z}^{(i)}\right)\right)\right) \right].$$

end for

- Sample minibatch of m noise samples $\{\boldsymbol{z}^{(1)}, \ldots, \boldsymbol{z}^{(m)}\}$ from noise prior $p_g(\boldsymbol{z})$.
- Update the generator by descending its stochastic gradient:

$$\nabla_{\theta_g} \frac{1}{m} \sum_{i=1}^{m} \log\left(1 - D\left(G\left(\boldsymbol{z}^{(i)}\right)\right)\right).$$

end for

Figure 2.4: GAN algorithm [22].

optimize is defined in terms of both networks parameters:

$$\min_G \max_D \mathbb{E}_{\mathbf{x} \sim p_{data}}[log\, D(\mathbf{x})] + \mathbb{E}_{\mathbf{z} \sim p_{\mathbf{z}}}\left[log\big(1 - D(G(\mathbf{z}))\big)\right] \tag{2.3}$$

The discriminator is trying to maximize equation 2.3. When doing so D only updates its own set of parameters and has not effect on the parameters of the generator. On the other hand the generator is trying to minimize equation 2.3 and it also does so by updating only its set of parameters. We can see that G optimizes only the second term of the equation. This means that the gradient for updating G's parameters is coming from the discriminator. The authors in [22] prove that for the generator to implicitly represent the distribution of the real training data, the discriminator must output 0.5 on any input. This means for the discriminator to be maximally confused on the class of any data point and hence for the generator finally able to produce data that the discriminator cannot distinguish from the real ones. The discriminator D can be understood as trying to maximize the log-likelihood for estimating the conditional probability of $p(Y = y|\mathbf{x})$ where Y is the random variable denoting whether $\mathbf{x}$ is coming from the real data or from the learned distribution of generator, p_g. Moreover the authors show that this optimization process from the discriminator is indeed the minimization of the **Jensen-Shannon divergence** between the learned distribution p_g and the real data distribution. The training process is visualized in figure 2.5.

Despite the theoretical grounds, training GANs is very difficult [16].

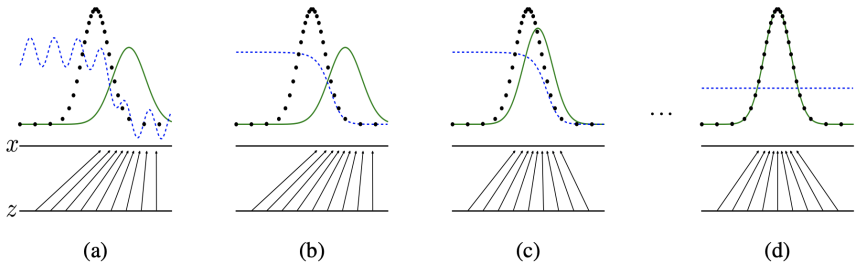


Figure 1: Generative adversarial nets are trained by simultaneously updating the **d**iscriminative distribution (D, blue, dashed line) so that it discriminates between samples from the data generating distribution (black, dotted line) $p_{\boldsymbol{x}}$ from those of the **g**enerative distribution p_g (G) (green, solid line). The lower horizontal line is the domain from which $\boldsymbol{z}$ is sampled, in this case uniformly. The horizontal line above is part of the domain of $\boldsymbol{x}$. The upward arrows show how the mapping $\boldsymbol{x} = G(\boldsymbol{z})$ imposes the non-uniform distribution p_g on transformed samples. G contracts in regions of high density and expands in regions of low density of p_g. (a) Consider an adversarial pair near convergence: p_g is similar to p_{data} and D is a partially accurate classifier. (b) In the inner loop of the algorithm D is trained to discriminate samples from data, converging to $D^*(\boldsymbol{x}) = \frac{p_{\text{data}}(\boldsymbol{x})}{p_{\text{data}}(\boldsymbol{x})+p_g(\boldsymbol{x})}$. (c) After an update to G, gradient of D has guided $G(\boldsymbol{z})$ to flow to regions that are more likely to be classified as data. (d) After several steps of training, if G and D have enough capacity, they will reach a point at which both cannot improve because $p_g = p_{\text{data}}$. The discriminator is unable to differentiate between the two distributions, i.e. $D(\boldsymbol{x}) = \frac{1}{2}$.

Figure 2.5: GAN training process [22].

Some of the major problems involve:

- Discriminator loss quickly vanishing and hence providing no gradients to the generator for the training process to progress.
- Mode collapse, the state where the generator collapses to only a minority of the modes of the distribution and hence generating the same (or very similar) data point(s).
- The convergence of the zero-sum game is very brittle and often the models diverge. The converge is also very susceptible to hyperparameters.

Multiple works [57, 46, 5, 13] have been introduced that present different ways to handle these problems in training GANs. These solutions range from simple tricks in the training process to changes of the objective function for the discriminator and generator alike.

2.5.2 Conditional GAN

GANs as introduced by [22] allow sampling from a learned distribution that mimicks that of the real training data. Mirza et. al changed the structure of both models in GAN to allow for conditional generation of data points [36]. They base their work on problems that are posed as one-to-many mapping.

What this means is that for some problems, for a single input, multiple different outputs are desirable. As an example, consider a GAN able to generate images of different types of clothes. If we were to use vanilla GAN then for different noise vectors $\mathbf{z}$ sampled from $p_{\mathbf{z}}$ we would get a different type of cloth. If we want to generate images of t-shirts then we would have to search the latent space until the right vector noise would produce an image of a t-shirt. However if we were to condition the GAN on the type of cloth we want (e.g. t-shirts) then the produced images would belong to that class but enough variations, like different fit, colors, etc., to make it look like a real image of a t-shirt.

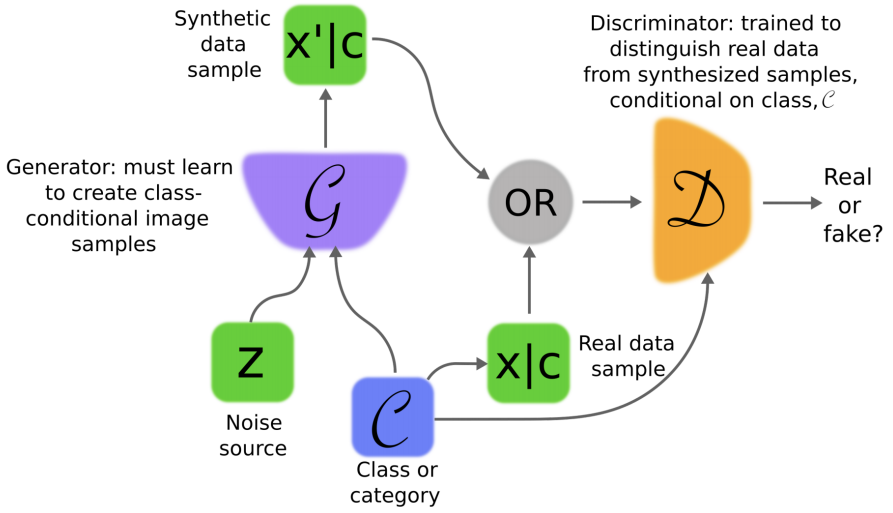


Figure 2.6: Conditional GAN architecture [16].

Figure 2.6 shows the architecture for the Conditional GAN (cGAN). In cGAN, a conditioning vector c is concatenated to the noise vector of the generator in order for the latter to generate a data point representative of the class denoted by c. c is also used to sample real training data $\mathbf{x}$ and concatenated together are inputted to the discriminator. The objective function in a cGAN is changed to account for the conditioning:

$$\min_G \max_D \mathbb{E}_{\mathbf{x}\sim p_{data}}[log\, D(\mathbf{x}|c)] + \mathbb{E}_{\mathbf{z}\sim p_{\mathbf{z}}}\left[log\big(1 - D(G(\mathbf{z}|c))\big)\right] \tag{2.4}$$

2.6 GAN-based RS

Despite various works introducing GANs to different application domains, they are still not very present in the RS literature. In this section we present

two of the most important techniques that combine GANs with RS.

2.6.1 IRGAN

IRGAN [51] is the first to bring GANs in the fields of IR and RS. The authors provide a unification of two schools of thinking for modelling in the information retrieval process. The generative school of thinking interprets the information retrieval process as an underlying generative model that describes how a relevant document can be generated from a specific query:

$$q \rightarrow d$$

where q is the query, d is a document and $\rightarrow$ denotes the generation direction. The discriminative school of thinking on the other hand assumes that for a specific query we already have a set of documents and their relevancy score and the resulting model is trying to understand this mapping by assuming both the query and the documents as features:

$$q + d \rightarrow r$$

where r is the relevancy score and $+$ denotes concatenation of features. The authors point out that the generative school of thinking of queries and documents provides theoretically sound modelling of their features but nevertheless suffers from the inability to use relevancy scores that can be retrieved from various sources. In this context they give a unification of both schools of thinking in a *minimax* game inspired by GANs.

Given a set of queries $\{q_1, \ldots, q_N\}$ and a set of documents $\{d_1, \ldots, d_M\}$, the true query relevancy distribution can be expressed as the conditional distribution $p_{\text{true}}(d|q, r)$ over the set of candidate documents (catalogue of items in the RS). With samples from $p_{\text{true}}(d|q, r)$ acting as training data (ratings in RS) the authors construct two separate models:

- **Generative retrieval model** $p_\theta(d|q, r)$ (G) whose aim is to select documents from a pool of documents for the given query q and approximate $p_{\text{true}}(d|q, r)$.
- **Discriminative retrieval model** $f_\phi(q, d)$ (D) whose aim is to distinguish relevant query-document pairs (q, d) from irrelevant ones. This is a binary classifier just like the discriminator in a vanilla GAN.

These two models play the minimax game with the following common objective function:

$$J^{G^*,D^*} = \min_G \max_D \sum_{n=1}^{N} \Big(\mathbb{E}_{d\sim p_{true}(d|q_n,r)}[log\, D(d|q_n)] + \mathbb{E}_{d\sim p_\theta(d|q_n,r)} \left[log\big(1 - D(d|q_n)\big)\right] \Big)$$

where

$$D(d|q) = \sigma(f_\phi(d,q)) = \frac{\exp(f_\phi(d,q))}{1+\exp(f_\phi(d,q))}$$

The objective function can be understood in the following way. The discriminative model D is trying to maximize the probability of seeing relevant document d for query q_n when d is sampled from the true underlying relevance distribution of query. At the same time, if the document for query q_n is generated (sampled) by the generative model, D is trying to minimize the probability of such document to be relevant. On the other hand G can only minimize the second term of the objective function. In order to minimize J^{G^*,D^*}, G must select documents for q_n such that D classifies them as relevant.

IRGAN is an instance of a conditional GAN (both D and G are conditioned on the query) and as such both models of IRGAN are neural networks. Assuming a differentiable function f_ϕ, D can be trained by SGD. The generative model $p_\theta(d|q,r)$ performs a discrete sampling from the pool of documents and for this reason it is not differentiable. Authors of IRGAN make use of the policy gradient based reinforcement learning algorithm REINFORCE [54] to update parameters θ of G. The selection done by G from the pool of documents is based on a softmax function:

$$p_\theta(d_k|q,r) = \frac{\exp(g_\theta(q,d_k))}{\sum_d \exp(g_\theta(q,d_k))}$$

where $g_\theta(\cdot)$ is a scoring function that reflects how relevant document d_k is for query q.

IRGAN was tested in [51] in 3 different domains. One of them is also RS. The 3 different domains differ in the way the scoring functions g_θ and f_ϕ are defined. For RS domain both functions are implemented as a MF approach:

$$g_\theta(u,i) = f_\phi(u,i) = s(u,i) = b_i + v_u^T v_i$$

where $v_u, v_i \in \mathbb{R}^k$ are the latent factors for user u and item i and b_i is the bias term for item i.

Finally the training process of IRGAN follows that of vanilla GAN where each of the players take turn in optimizing the objective function until convergence. The authors point out that this convergence is domain-specific.

2.6.2 CFGAN

CFGAN [11] takes a totally different approach in introducing GANs into RS. CFGAN is presented as a solution for Top-N recommendation problem. The authors of CFGAN provide evidence that the training process of IRGAN is flawed and empirically prove it (we summarize it in section 3.1 as a natural step in deriving our model). To bypass this flawed training process the authors of CFGAN introduce *vector-wise* training for GANs applied to RS. The main idea is to consider a historical user profile as a vector and make the generator of a GAN generate plausible deterministic historical profile when conditioned on a specific user, instead of discrete sampling for relevant items. The discriminator then is tasked with distinguishing between generated user profiles and real user profiles. This also helps avoid using the REINFORCE algorithm for updating the generator since with vector-wise training both models can be trained with SGD. We give a visualization of CFGAN in figure 2.7.

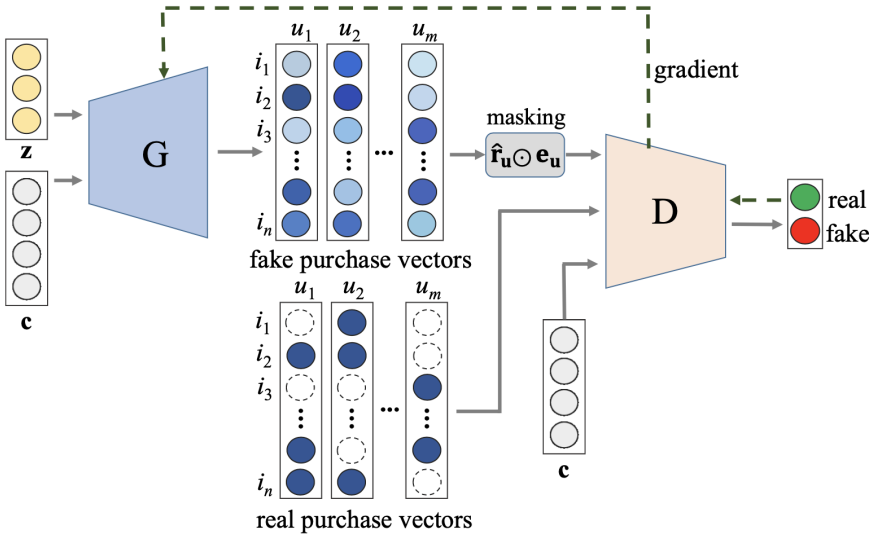


Figure 2.7: CFGAN model [11].

We explain CFGAN model focusing on the components on figure 2.7. CFGAN is also an instance of cGAN just like IRGAN. The generator network G takes as input a user conditioning vector c_u, that uniquely identifies

the user. cGAN generator takes as input also a random noise vector $\mathbf{z}$ but since the user profile generation must be deterministic, this noise vector is dropped. The generated user profiles are then used as input for the discriminator alongside the real user profiles with which the discriminator D is trained to label the ones coming from the G as fake and the ones coming from the URM as real.

The CFGAN minimax objective is given as:

$$
\begin{aligned}
J^D &= -\mathbb{E}_{\mathbf{x}\sim P_{data}}[\log D(\mathbf{x}|c)] - \mathbb{E}_{\hat{\mathbf{x}}\sim P_\phi}[\log(1 - D(\hat{\mathbf{x}}|c))] \\
&= -\sum_u \log D(r_u|c_u) - \sum_u \log\Big(1 - D\big((\hat{r}_u \odot e_u)|c_u\big)\Big) \\
&= -\sum_u \Bigg(\log D(r_u|c_u) + \log\Big(1 - D\big((\hat{r}_u \odot e_u)|c_u\big)\Big)\Bigg) \\
J^G &= \sum_u \log\Big(1 - D\big((\hat{r}_u \odot e_u)|c_u\big)\Big)
\end{aligned}
$$

where J^D is the objective that D minimizes, J^G is the objective G minimizes, $\hat{r}_u$ is the generated profile for user u, c_u is the user conditioning vector, e_u is a masking vector with $e_u = r_u$ and $\odot$ is the element-wise multiplication operator.

Before being used as input for the discriminator, the generated profile $\hat{r}_u$ is masked with a masking vector e_u (the real user profile). The reason for this stems from other CF approaches where only the observed interactions are used to make predictions about future interactions [11]. However this masking operation brings a trivial solution according to the authors of CFGAN; when dealing with implicit feedback the user profile is binary and sparse hence the resulting vector from the multiplication of the generated user profile $\hat{r}_u$ with the masking vector e_u will contain zero values exactly on the items with which the user has not interacted. Thus the generator could learn the trivial solution of generating $\overrightarrow{\mathbf{1}}$ (the vector containing only ones) since it is the identity element for element-wise multiplication.

To counteract the trivial solution CFGAN introduces 3 novel CF methods. The idea behind them is the following: at the beginning of each training iteration a portion of the non-interacted items are selected for each user and they are assumed to be irrelevant for the user, instead of missing. Then G is trained to output zero for these negative items in the generated user profile. The presented methods are:

- **Zero-reconstruction**: non-interacted items are selected at random

with a specific ratio and are assumed to be irrelevant. Then the generator objective is changed to:

$$J^G = \sum_u \left(\log\Big(1 - D\big((\hat{r}_u \odot e_u)|c_u\big)\Big) + \alpha \sum_j (x_{uj} - \hat{x}_{uj})^2 \right)$$
$$j \in N_u^{ZR(t)} \implies x_{uj} = 0$$

where $\hat{x}_{uj}$ denotes a specific element of the generated user profile, x_{uj} denotes a specific element of the real user profile, α is a coefficient that weights the regularization term and $N_u^{ZR(t)}$ denotes the set of items selected as irrelevant for user u during training iteration t under zero-reconstruction method. This new objective makes G deviate from the trivial solution by forcing the output for irrelevant items to be zero.

- **Partial masking**: during each training iteration at random non-interacted items are selected with a specific ratio. These items are assumed to be relevant for the user and both D and G are forced to consider them during the training with the following changed objective functions:

$$J^D = -\sum_u \left(\log D(r_u|c_u) + \log\Big(1 - D\big(\hat{r}_u \odot (e_u + k_u)|c_u\big)\Big) \right)$$
$$J^G = \sum_u \log\Big(1 - D\big(\hat{r}_u \odot (e_u + k_u)|c_u\big)\Big)$$
$$k_{uj} = \begin{cases} 1 & \text{if item } j \in N_u^{PM(t)} \\ 0 & \text{otherwise} \end{cases}$$

where $N_u^{PM(t)}$ denotes the set of items selected as irrelevant for user u during training iteration t under partial masking method.

- **Combination of zero-reconstruction and partial-masking**: both previous methods are applied. The objective function of D remains the same as for partial masking whereas the function of G changes as follows:

$$J^G = \sum_u \left(\log\Big(1 - D\big(\hat{r}_u \odot (e_u + k_u)|c_u\big)\Big) + \alpha \sum_j (x_{uj} - \hat{x}_{uj})^2 \right)$$

Chapter 3

Model

In this chapter we describe the derivation of our proposed model. We give again the original formulation of GAN for completeness and then proceed with the related work that lay the necessary steps to our model. We indicate the conditions and assumptions necessary for the model which we test through several research questions presented and addressed in chapter 5.

3.1 From GAN to RS

Generative Adversarial Networks [22] are now ubiquitous in the Computer Vision domain due to their generative modeling abilities. GANs learn implicitly a probability distribution by being able to generate samples in the data space from a noise input $\mathbf{z}$ sampled from a predefined distribution $p_{\mathbf{z}}$. As introduced in section 2.5.1, GANs are composed of two players, usually two neural networks that play a *minimax* zero-sum game in order to learn a target distribution. The objective function that these two networks optimize as given by [22] is shown below:

$$\min_G \max_D \mathbb{E}_{\mathbf{x}\sim p_{data}}[log\, D(\mathbf{x})] + \mathbb{E}_{\mathbf{z}\sim p_{\mathbf{z}}}\left[log\big(1 - D(G(\mathbf{z}))\big)\right]$$

where the discriminator D is a binary classifier with the well-known *sigmoid* activation function:

$$D(x) = \frac{1}{1 + \exp(-x)}$$

and the generator G is any neural network that maps noise input $\mathbf{z}$ to the data space. As indicated in section 2.5.1 the generator network uses the gradient from the discriminator to update its weights and generate better samples to fool the discriminator. In Computer Vision the generator is usually tasked with producing complete images where each composing pixel of the image can take continuous values ranging from 0 to 255.

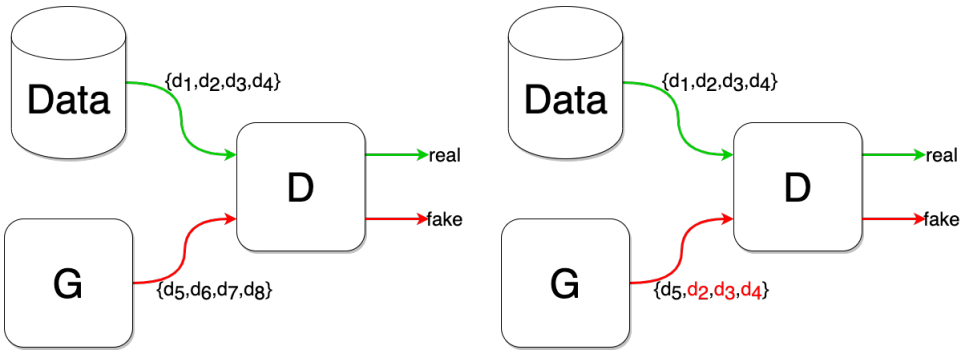
IRGAN [51] was first to propose the application of GAN for IR and RS. Differently from the original formulation, IRGAN uses the generator network to generate/select item d (e.g. item ID) from a pool of items as the most relevant for a specific user. This generation process is different from applications of GANs in Computer vision because picking items consists in a *discrete* sampling procedure. The objective function in IRGAN (in the context of information retrieval) is given as:

$$J^{G^*,D^*} = \min_G \max_D \sum_{n=1}^{N} \Big(\mathbb{E}_{d\sim p_{true}(d|q_n,r)}[log\, D(d|q_n)] + \mathbb{E}_{d\sim p_\theta(d|q_n,r)} \big[log\big(1 - D(d|q_n)\big)\big] \Big)$$

where q_n is a submitted query, r is the relevance distribution of a user over items and $p_\theta(d|q_n, r)$ is the generative model G characterized by parameters θ that learns to select document d that are most relevant for the submitted query. Since the generator samples discrete values its update cannot be done through gradient descent so for this reason IRGAN uses the policy-gradient based reinforcement learning [54].

CFGAN [11] brings forward a potential issue in the optimization problem of the discriminator of IRGAN. The aim of the algorithm is for the generator to be able to sample items that are most likely relevant for the user. Consider the beginning of the training procedure of IRGAN. Initially the discriminator has to differentiate between real item IDs and fake item IDs selected by the generator. This is easy in the beginning since the generator is not yet optimal so it will generate IDs that the discriminator can easily detect as not being relevant for the user. However, when approaching the optimality of the generator, the discriminator will be presented with item IDs that are identical to the ones already in the historical profile of the user but are presented to the discriminator labelled as *real* (when the item ID is coming from the real training data) and *fake* (when the item ID is coming from the generator) at the same time. CFGAN empirically shows that training IRGAN up to this point confuses the discriminator and deteriorates the accuracy of the algorithm.

To solve this problem CFGAN proposes a vector-wise training for RS where the generator G outputs *real-valued* vectors. In this way D can discriminate between real user historical interactions and plausible historical interactions produced by G.



(a) IRGAN model in the first epochs of training. The items picked by the generator are different from the ground truth.

(b) IRGAN model near/at optimal generator. The generator picks items with the label *fake* identical to those in the training set which the discriminator has seen as *real*.

Figure 3.1: Discrete item generation issue with IRGAN.

3.2 GANMF

Our model, which we denote as GANMF, attemps to solve the generic Top-N recommendation problem presented in chapter 1. We take a model-based approach for the recommendations and structure GANMF as a matrix factorization model. We task our model to learn the distribution of historical interactions of each respective user utilizing the *vector-training* introduced by CFGAN. This means for the generator to be able to produce historical profiles that are very similar to the training data but differ in such a way that can provide recommendations better than (comparable to) other traditional MF-based baselines. GANMF follows the original GAN formulation and is composed of two players, a generator network G and a discriminator network D. GAN allow the modeling of *multi-modal* outputs [21] where for a specific input there might be multiple correct outputs or labels. For this reason the input of the generator in a GAN is usually some noise from a predefined distribution. By using this noise GANs can produce diverse samples that resemble the training data. However, in our case the recommendation must be deterministic and unique in order for it to provide the best user experience and make it feel personalized for the users. This urges the conditioning of the generation process on each user available in our training data.

3.2.1 Discriminator D

The discriminator in a GAN is used to differentiate the source of the data it takes as input. As mentioned in 2.5.1 the discriminator is usually a binary classifier and in the original formulation it outputs the probability of the input being real and not generated by G. For GANMF we take another approach and model D according to the discriminator in EBGAN [57]. EBGAN was first to introduce the discriminator as an energy function where is assigns low energy to samples in the data manifold and high energy elsewhere. In this context we can think of the energy function as a hyperplane which takes the shape of a valley near real data that come from the training set and takes the shape of mountains near data generated by G. Just like EBGAN, GANMF uses an autoencoder [32] as the discriminator where the reconstruction loss acts as the energy function:

$$D(x) = \big\|Dec\big(Enc(x)\big) - x\big\| \tag{3.1}$$

where $Enc(\cdot)$ and $Dec(\cdot)$ are the *encoder* and *decoder* functions and $||\cdot||$ is the Euclidean norm.

The rationale for using an autoencoder model as discriminator for GANMF falls in the same line of the discussion by the authors of EBGAN. In the original formulation the discriminator's output is a scalar value squashed in the range $[0-1]$ by a *sigmoid* activation indicating a probability. The output of the generator in the case of GANMF is very high dimensional; specifically it is the length of a user historical profile $|I|$, which for some datasets might be in thousands or even millions of dimensions. Updating the weights of the generator through the gradient of a single scalar value in the discriminator output poses difficulties for learning the generator. Consider the case when two generated historical profiles for the same user, $\hat{I}_u^1$ and $\hat{I}_u^2$, differ between each other a lot but for the discriminator they are both fake profiles. The gradient propagated back to update the weights is going to be more or less the same for both generated profiles. Assuming I_u^* to be the optimal generated profile for user u then under some distance metric (e.g. Euclidean distance) we have:

$$||I_u^* - \hat{I}_u^1|| \leq ||I_u^* - \hat{I}_u^2|| \quad \bigvee \quad ||I_u^* - \hat{I}_u^2|| \leq ||I_u^* - \hat{I}_u^1||$$

yet the gradient coming from the discriminator will not make this distinction very clear. Also when training through Mini Batch Gradient Descent, having a single value output makes the gradient for the batch highly unlikely to be orthogonal for the individual batch samples [57]. We further explore the

effect of the autoencoder as the discriminator through experiments in section 5.3.

Given a real data x and a user conditioning vector y (more on this in 3.2.2) the loss function for GANMF discriminator is given by the *hinge loss* [57]:

$$\mathcal{L}_D(x, y) = D(x) + \big[m - D\big(G(y)\big)\big]^+ + \lambda_D \|\Omega^D\|_2^2 \tag{3.2}$$

where $[\cdot]^+ = \max(0, \cdot)$, m is a positive margin, $D(\cdot)$ is the autoencoder reconstruction loss as defined in equation 3.1, $G(\cdot)$ is the generator function, λ_D is a regularization constant and Ω^D is the set of parameters of the discriminator. D (the discriminator, not the $D(\cdot)$ function) is trained to minimize 3.2. The first term denotes the reconstruction error of the autoencoder on real user profiles. Since the reconstruction error is always positive the minimum of this term is 0 meaning a perfect reconstruction. The second term involving the max operator can be summarized as:

$$\big[m - D\big(G(y)\big)\big]^+ = \max\Big(0, D\big(G(y)\big)\Big) = \begin{cases} 0 & D\big(G(y)\big) \geq m \\ m - D\big(G(y)\big) & D\big(G(y)\big) < m \end{cases}$$

This term tries to keep the reconstruction loss of generated user profiles above the margin value. Since the discriminator is trying to minimize this term also, the autoencoder's weights are updated in such a way as to prevent the reconstruction error falling below the margin otherwise it is penalized by how much the error violates it. If the reconstruction loss is more than the margin, the second term reaches its minimum at 0. Using the max operator we achieve a higher energy value for the generated user profiles. This operator is common in training SVM where optimal class-separating hyperplane is the one that does not violate the margin from the support vectors [14]. In GANMF m is a hyperparameter of the model which we tune through a hold-out validation set.

The last term of $\mathcal{L}_D$ is a L_2 *regularization term* on the parameters of the discriminator that helps prevent overfitting.

3.2.2 Generator G

We now detail the generator network. As stated in section 3.2, we construct G as a conditional generator by using condition attributes that are unique to each user. These attributes serve as the only input to a fully-connected neural network. In [47] the authors show that a single layer autoencoder with linear activation function in the output layer is very similar to a low-rank MF approach. Instead of following this approach we make use of embedding layers present in deep learning frameworks like Tensorflow, PyTorch,

Keras, etc. An embedding layers is used as part of a bigger neural network to map integer values to real-valued vectors. They are usually the first layer in a multi-layer neural network and the parameters that they learn are task-specific. We use two such embedding layers for the user latent factors and item latent factors. In figure 3.2 we give a visual representation of the generator network.

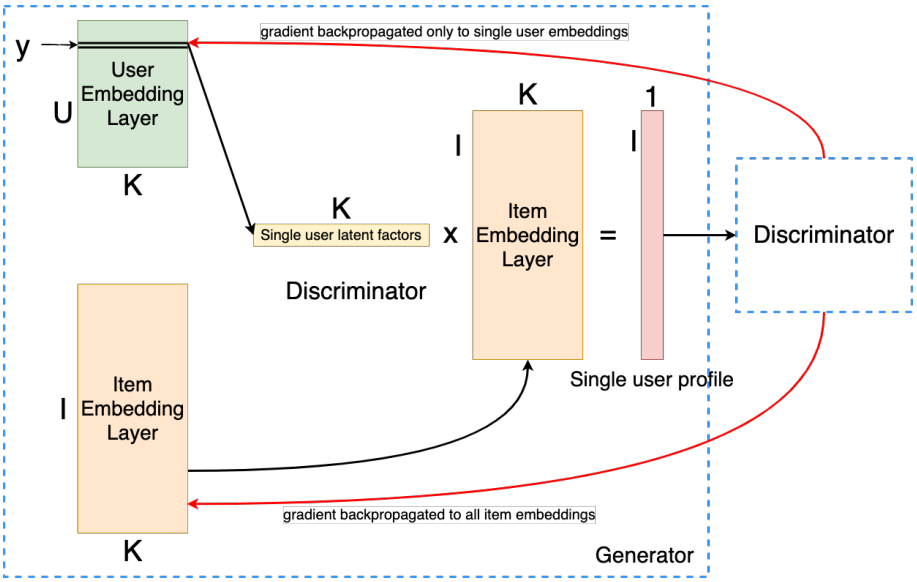


Figure 3.2: Generator network casted as MF-based approach with embedding layers.

Embedding layers are implemented as matrix of weights. They take as input a row-identifier and return the weights at that row in the matrix. Since we are utilizing only the URM and we have opted for embedding layers we are limited to one option in regards to possible conditioning vectors. We can use the row number in the URM respective to the user. In this case the generator input is just an integer value in the range $[1 - |U|]$. In the forward propagation phase of the training, when taking this user identifier the network first retrieves the embeddings at that row in the user embedding layer and then performs a vector-matrix multiplication of these embeddings with the matrix of item embedding layer. This operation results in a vector of dimension $|I|$, the length of a user profile. A clear advantage of using embedding layers is that the number of parameters to be learned by the generator is $\Theta(K \times (|I| + |U|))$, similar to baselines like WRMF. Finally the generator produces a user profile as follows:

$$G(y) = \Sigma[y, :]V$$

where Σ is the matrix denoting the user latent factors and V is the matrix denoting the item latent factors.

The generator's job is to fool the discriminator of GANMF. While the discriminator's job is to increase the reconstruction loss for generated user profiles, the generator tries to minimize the reconstruction loss:

$$\mathcal{L}_G(y) = D\big(G(y)\big) + \lambda_G \|\Omega^G\|_2^2 \tag{3.3}$$

where $G(\cdot)$ and $D(\cdot)$ are the generator and discriminator functions respectively, y is the user conditioning vector, λ_G is the L_2 regularization coefficient and Ω^G is the set of parameters of the generator. In our experiments we set $\lambda_G = 0$ because the generator is never trained directly with the real data but with gradients coming from the discriminator. Not being able to see the real data makes overfitting the generator difficult.

3.2.3 Single-sample class conditioning

As stated in section 3.2, GANMF at its core is a cGAN of the type presented in [36]. Usually the condition input in a cGAN is a label or class associated with the data sample passed to the generator in order to condition the generation process but also to the discriminator so it can judge the realness of the data sample given the label. Applications of cGAN usually involve datasets with multiple classes where for each single class there are hundreds or thousands of samples from the training set. This allows the discriminator to learn not only the source of the samples it receives but also a relation between the sample and its label.

We experimented with a binary classifier discriminator (as per the original formulation of GAN) where the input was the concatenation of a real/generated profile with the conditioning vector representing the user of the profile. We faced a peculiar problem with this version of GANMF; the user latent factor resulting from the training where very similar with one another and the quality (see evaluations) of the recommendations was very poor. However, considering that for every single user we only have one real historical profile this finding is not a surprise. Generalizing the input-target relation from a single data point per label is highly unlikely.

To alleviate this problem and further improve the accuracy of the recommendations we follow an approach called *feature matching* from [46]. Feature matching is presented as a technique to stabilize the training procedure of GANs and also avoid *mode collapse* (see section 2.5.1). During training, the generator might trick the discriminator very easily by finding a sample that the discriminator cannot distinguish from the real data and always generate

that particular sample or minor variations of it. Usually this single sample need not even be recognizable as a meaningful data point in comparison with the dataset at hand (e.g. if generating images of handwritten digits the output of the generator could be a tower in a black background which might resemble the digit one and hence the reason the discriminator cannot distinguish it but is entirely not related to what we are expecting from the generator). Feature matching changes the objective of the generator to not deceive the discriminator but match real data statistics by using the following loss term:

$$\Big\| \mathbb{E}_{\mathbf{x} \sim p_{data}} \mathbf{f}(\mathbf{x}) - \mathbb{E}_{\mathbf{z} \sim p_z(\mathbf{z})} \mathbf{f}\big(G(\mathbf{z})\big) \Big\|_2^2$$

where $\mathbf{f}(\cdot)$ is the activation of an intermediate layer of the discriminator, $G(\cdot)$ is the generator function, and $|| \cdot ||_2^2$ is the Euclidean norm squared.

We incorporate this technique in GANMF as a way to enforce conditioning the generating process. The conditioning vector in GANMF is necessary for the generator in order to retrieve the user latent factors. However, the generating process could still be stuck by discarding the information in the conditioning vector and by finding a specific user profile that deceives the GANMF discriminator (when reconstructed in the discriminator, the loss is the same as the average loss of all real user profiles). Such profile can very easily be a historical profile with interactions on only the most popular items given the popularity bias of RS datasets (see section 4.6). To force the generator to learn user-specific latent features we change the previous loss of the generator to the following:

$$\mathcal{L}_G(x, y) = \beta\, D\big(G(y)\big) \;+\; (1 - \beta) \Big\| l^1(x) - l^1\big(G(y)\big) \Big\|_2^2 + \lambda_G \|\Omega^G\|_2^2 \quad (3.4)$$

where l^1 is the activation of the hidden layer of the autoencoder (discriminator, see section 3.2.1), $D(\cdot)$ and $G(\cdot)$ are the discriminator and generator functions respectively, x and y are a real user profile and the corresponding conditioning vector respectively and β is a weighting term that decides the balance of the generator between deceiving the generator and matching the hidden layer activations. The last terms is the L_2 regularization term of parameters Ω^G of the generator controlled by the coefficient λ_G.

The second term of $\mathcal{L}_G$ takes the form of another autoencoder incorporating the generator and the encoder part of the discriminator. Given real user profile x and its corresponding conditioning vector y we first retrieve the hidden layer activation of the discriminator for x, $l^1(x)$. Then we generate a plausible user profile for y, $G(y)$. We retrieve the hidden layer activation of the discriminator for $G(y)$, $l^1(G(y))$. If the generated profile

is indeed suitable for the user denoted by y then we should have that $l^1(x)$ and $l^1(G(y)$ must be close to each other by some distance metric (here we use the Euclidean distance squared as [46]).

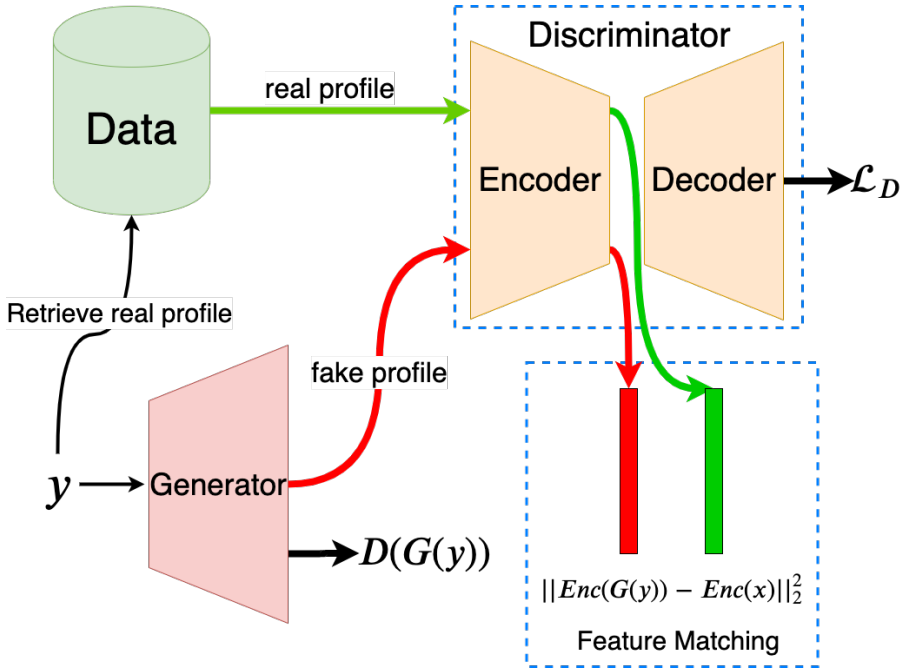


Figure 3.3: Architecture of GANMF.

3.3 Differences to IRGAN and CFGAN

We briefly summarize the differences between GANMF and the two GAN-based models presented in section 2.6. Compared to IRGAN, GANMF does not use the discrete sampling approach for selecting relevant items during training and inference phase but relies on the vector-wise training procedure of CFGAN. Different from CFGAN, GANMF uses an autoencoder as discriminator in order to provide richer gradients to the generator. In constrast with both CFGAN and IRGAN, GANMF uses a discriminator not conditioned on the user profile it tries to reconstruct. However it enforces conditioning through the application of feature matching loss that is added to the generator. Finally the generator of GANMF still performs a linear MF operation whereas the profile generation process in CFGAN is by construction non-linear. Nevertheless GANMF still provides better performance than CFGAN, as shown in chapter 5.

3.4 Training

3.4.1 Update rules

Figure 3.3 depicts the full GANMF model. Since we are using the vector-training approach from CFGAN, we can bypass the policy-gradient based reinforcement learning used by IRGAN and train the complete model end-to-end by Gradient Descent (GD) [45]. The update rule for GD is given by:

$$w^{k+1} = w^k - \mu \frac{\partial \mathcal{L}}{\partial w^k} \tag{3.5}$$

where w^k are the parameters of the model at time k, μ is the *learning rate* and $\mathcal{L}$ is the loss function the model is trying to optimize. The update rule of GD is identical for every neural network, what changes is the partial derivative of the loss function. Among the variations of gradient descent we use *minibatch* GD since it allows for higher degree of parallelism compared to *stochastic* GD and in this way we can make use of graphical processing units (GPUs) for training.

The discriminator network of GANMF is composed of an autoencoder incorporating two parts, the encoder and decoder models. The functions denoted by each of them for a user profile $\mathbf{x}$ (a column vector) are the followings (in vector notation):

$$\begin{aligned} Enc(\mathbf{x}) &= h(\mathbf{b}^E + \Theta^E \mathbf{x}) \\ Dec\big(Enc(\mathbf{x})\big) &= g\big(\mathbf{b}^D + \Theta^D Enc(\mathbf{x})\big) \\ &= g\Big(\mathbf{b}^D + \Theta^D \big(h(\mathbf{b}^E + \Theta^E \mathbf{x})\big)\Big) \end{aligned}$$

where $\mathbf{b}^E$ and $\mathbf{b}^D$ are the encoder and decoder bias vectors respectively, Θ^E and Θ^D are the encoder and decoder parameters respectively and $h\cdot$ and $g(\cdot)$ are activation functions.

Given as input a batch of real profiles $\mathbf{x}$ and conditioning vectors $\mathbf{y}$, the value of the loss $\mathcal{L}_D$ (3.2) is:

$$\begin{aligned} \mathcal{L}_D(\mathbf{x}, \mathbf{y}) = \frac{1}{|B|} \sum_{i \in B} \Bigg[& \Big\| Dec\big(Enc(\mathbf{x}_i)\big) - \mathbf{x}_i \Big\|_2^2 + \\ & \max\left(0, m - \Big\| Dec\Big(Enc\big(G(\mathbf{y}_i)\big)\Big) - G(\mathbf{y}_i) \Big\|_2^2\right) \Bigg] \end{aligned}$$

In order to update the parameters of the discriminator we have to compute the partial derivatives of the loss with respect to each of its parameters. For ease of reading we write $\mathcal{L}_D$ as the sum of two terms and calculate the partial derivatives separately for each of them:

$$\mathcal{L}_D = \frac{1}{|B|} \sum_{i \in B} A + C$$

$$\frac{\partial \mathcal{L}_D}{\partial \{\mathbf{b}^D, \mathbf{b}^E, \Theta^D, \Theta^E\}} = \frac{1}{|B|} \sum_{i \in B} \frac{\partial}{\partial \{\mathbf{b}^D, \mathbf{b}^E, \Theta^D, \Theta^E\}} (A + C)$$

We leave the full derivation steps of the above partial derivatives in the appendix A.1 for the interested reader.

We focus now on the update rules for the generator network. The loss function $\mathcal{L}_G$ (3.4) is also composed of two terms so we split this loss too:

$$\begin{aligned}
G(\mathbf{y}) &= \Sigma[\mathbf{y}, :]V = \Sigma[\mathbf{y}]V \\
\mathcal{L}_G(\mathbf{x}, \mathbf{y}) &= \beta\, D\big(G(\mathbf{y})\big) \;+\; (1-\beta) \left\| l^1(\mathbf{x}) - l^1\big(G(\mathbf{y})\big) \right\|_2^2 \\
&= \beta \left\| Dec\Big(Enc\big(G(\mathbf{y})\big)\Big) - G(\mathbf{y}) \right\|_2^2 + (1-\beta) \left\| Enc(\mathbf{x}) - Enc\big(G(\mathbf{y})\big) \right\|_2^2 \\
&= \beta\, A + (1-\beta)\, B \\
\frac{\partial \mathcal{L}_G}{\partial \{\Sigma, V\}} &= \frac{1}{|B|} \sum_{i \in B} \beta \frac{\partial A}{\partial \{\Sigma, V\}} + (1-\beta) \frac{\partial C}{\partial \{\Sigma, V\}}
\end{aligned}$$

Again, we leave the full derivation steps of the partial derivatives of $\mathcal{L}_G$ in appendix A.2 for the interested reader.

3.4.2 Early Stopping

Early stopping is a regularization technique that helps prevent *overfitting* [55]. When training machine learning models, given enough expressiveness the model can learn the training data perfectly thus being able to reduce the loss to its minimum value. However when such fully trained models are used in inference phase their performance drops. This phenomenon is termed overfitting and when it occurs the model's generalization ability suffers since it learns to memorize the training data and their noise. Early stopping is a simple mechanism used to stop the training operation before it reaches the overfitting of the model. The metrics by which the training is stopped are

the sames ones used for evaluating the final model but they are computed during the training phase on a validation set. If the loss on the training set is dropping and hence the performance is increasing but in the same time the performance of the model on the validation set starts to decrease then we have reached the stopping point of the training. A by-product of early stopping is a reduction on the total wall time of the training process (especially during hyperparameter tuning) if the evaluation process is not very expensive. Suppose we were to run training for 100 epochs with early stopping executed every 5 epochs with 2 consecutive worse validation scores and the process is stopped after 40 epochs. This means that on epoch 30 the validation score has registered its highest value. On epoch 35 the validation score has been lower than the one registered on epoch 30. On epoch 40, for a second time, the validation score has been lower than the best registered one and the training is stopped. The final model is the one denoted by the parameters of the highest validation score. In this way we stopped the training process midway and saved the time it would take to train fully and with worse performance (figure 3.4). We incorporate early stopping in our training procedure in the way described above; we run early stopping evaluation every 5 epochs and allow 2 consecutive worse evaluations on the validation set.

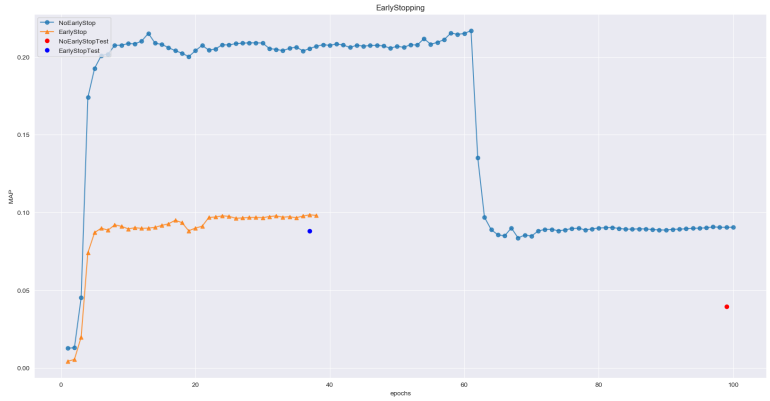


Figure 3.4: A depiction of the effect of early stopping on training. The dataset used is Movielens 100K, trained fully for 100 epochs and evaluated with MAP@5. The single blue dot represents the test set performance of the model trained with early stopping and the single red dot the test set performance without early stopping. We can see that the performance of training w/o early stopping continues improving until around epoch 60. However, if we were to train the model with early stopping we would have stopped before epoch 40 because the performance on the validation set after this point ceases to improve thus saving wall time when training and also making sure the final model is one that most probably will have higher performance on a holdout set.

The updating of the discriminator and generator is done according to the vanilla GAN implementation; first we update the parameters of the discriminator and then the parameters of the generator. Algorithm 1 details the steps of the training procedure.

Algorithm 1: GANMF Training

Input : set of users U, training URM_{tr}, validation URM_{val}, margin m, reconstruction coefficient β, D learning rate μ_D, G learning rate μ_G, batch size B, early stopping allowedWorse

Output: trained G model that can generate historical user profiles

initialize($\mathbf{b}^E$, $\mathbf{b}^D$, Θ^E, Θ^D, Σ, V)
noWorse $\leftarrow 0$
numIterations $\leftarrow \frac{|U|}{|B|}$
while *stopping condition not met* **do**
  for iter *in* numIterations **do**
    // Discriminator learning
    $u \leftarrow$ sampleBatch(U)
    fakeProfiles $\leftarrow$ generateBatchProfiles(u)
    realProfiles $\leftarrow URM_{tr}[u]$
    $\mathcal{L}_D \leftarrow$ compute(realProfiles, fakeProfiles) (eq. 3.2)
    $\frac{\partial \mathcal{L}_D}{\partial \mathbf{b}^E}, \frac{\partial \mathcal{L}_D}{\partial \mathbf{b}^D}, \frac{\partial \mathcal{L}_D}{\partial \Theta^E}, \frac{\partial \mathcal{L}_D}{\partial \Theta^D} \leftarrow$ compute($\mathcal{L}_D$, $\mathbf{b}^E$, $\mathbf{b}^D$, Θ^E, Θ^D)
    $\mathbf{b}^E, \mathbf{b}^D, \Theta^E, \Theta^D \leftarrow$ update($\mathbf{b}^E$, $\mathbf{b}^D$, Θ^E, Θ^D, $\frac{\partial \mathcal{L}_D}{\partial \mathbf{b}^E}$, $\frac{\partial \mathcal{L}_D}{\partial \mathbf{b}^D}$, $\frac{\partial \mathcal{L}_D}{\partial \Theta^E}$, $\frac{\partial \mathcal{L}_D}{\partial \Theta^D}$, μ_D) (section 3.4.1)

    // Generator learning
    $u \leftarrow$ sampleBatch(U)
    fakeProfiles $\leftarrow$ generateBatchProfiles(u)
    realProfiles $\leftarrow URM_{tr}[u]$
    $\mathcal{L}_G \leftarrow$ compute(realProfiles, fakeProfiles) (eq. 3.4)
    $\frac{\partial \mathcal{L}_G}{\partial \Sigma}, \frac{\partial \mathcal{L}_G}{\partial V} \leftarrow$ compute($\mathcal{L}_G$, Σ, V)
    $\Sigma, V \leftarrow$ update(Σ, V, $\frac{\partial \mathcal{L}_G}{\partial \Sigma}$, $\frac{\partial \mathcal{L}_G}{\partial V}$, μ_G) (section 3.4.1)
  end

  // Early Stopping
  performance $\leftarrow$ evaluate(G, URM_{val})
  if isWorse(bestMAP, performance) **then**
    noWorse ++
  else
    bestMAP = performance
    noWorse $\leftarrow 0$
  end
  if noWorse $>$ allowedWorse **then**
    break
  end
end

Chapter 4

Datasets

In this chapter we describe the datasets we used to perform the experiments detailed in chapter 5. Since all of the considered algorithms make use of only the URM we consider only parts of the datasets that are composed of interactions and disregard additional information like user/item information. We detail in the next section the procedure we followed to standardize the datasets. The following sections show the statistical information of each dataset along with the distribution of all user historical profiles and their 95^{th} percentile. Finally we report popularity bias of each dataset through metrics like Gini Index since it is an important characteristic when considering CF approaches.

4.1 Dataset preparation

All datasets are hosted in URLs of the research lab that gathered them. The preparation procedure is identical for each dataset. Upon downloading the data we transform them in a CSV file where each row denotes a user-item interaction with 3 mandatory elements – user ID, item ID and rating in dataset-specific range – if the data is not already in this format. Then we convert these interactions into a sparse URM matrix of implicit ratings of dimensions $|U| \times |I|$ where U is the set of users and I is the set of items read from the CSV file. We follow the approach used in [27] to binarize the URM by setting a cell of the matrix to 1 if there is an interaction between the user and the item denoted by the matrix cell and 0 otherwise.

Recommender systems, just like machine learning more generally, have two steps towards their evaluation. The first one is the training phase, the second is the testing phase. In order to perform both phases we divide each dataset into two mutually exclusive sets for training and testing. In order for

the evaluation to be as unbiased as possible we perform only the final scoring of the algorithms with the test set. During the training phase we have to also account for the tuning of various hyperparameters of the algorithms so an additional set is required, different from the training and test set. We secure this *validation* set from the training set and we detail this process in section 5.1.

Each row of the URM represents user preferences over the items. Each user interacts with a different number of items and extreme cases of interaction with only 1 or 2 items make it difficult to be allocated in the 3 splits. For this reason we remove from the initial URM all users that have interacted with only one item. We call this resulting matrix the full URM. For the users that have interacted with 2 items, we include one interaction in the training set and one in the test set. Users with more than 2 interactions are separated into training and test set following the ratio 4:1 respectively. This separation is done per user profile, e.g. if a user has interacted with 10 items, at random we assign 8 of those interactions to the training set and leave 2 in the test set. The statistics in subsequent sections are based on the full unprocessed datasets.

4.2 MovieLens

MovieLens[24] datasets are released by GroupLens Research lab from the University of Minnesota. They come in several versions identified by the number of user-item interactions available in each one. These versions are MovieLens 100K, 1M, 10M, 20M and 25M. In this thesis work we consider only MovieLens 100K and 1M due to computational constraints.

4.2.1 MovieLens 100K

Movielens 100K is the smallest dataset from MovieLens and the most dense containing 100000 interactions from 943 users on 1682 movies putting its density at 6.3%. The dataset has been constructed in such a way as to contain only users that have rated at least 20 items. The most active user has rated a maximum of 737 movies whereas all users on average have rated 106 movies. Ratings are in the range $[1 - 5]$ with a step of 1. Table 4.1 summarizes the dataset.

4.2.2 MovieLens 1M

MovieLens 1M is another version of MovieLens that we used to evaluate the algorithms. As the name suggests it has around 1 million interactions

Interactions	100000
Density	6.3%
Users	943
Items	1682
Avg. interactions	106.04
Min. interactions	20
Max. interactions	737

Table 4.1: MovieLens 100K dataset statistics

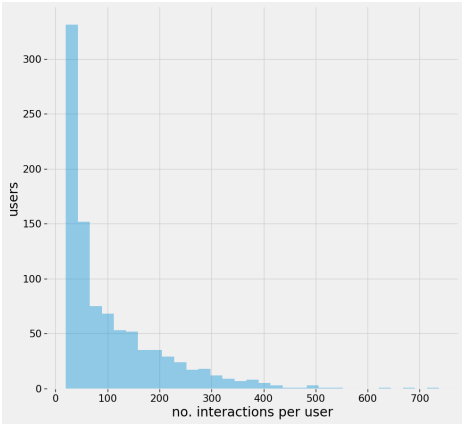


Figure 4.1: MovieLens 100K: distribution of per-user number of interactions

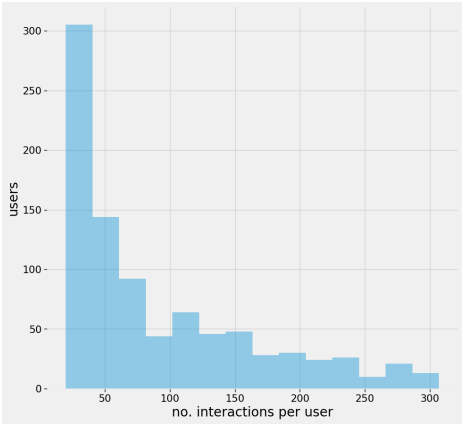


Figure 4.2: MovieLens 100K: distribution of 95th percentile number of interactions

from 6040 users on 3706 items. This translates in a matrix density of 4.47%. These interactions are explicit ratings of users given to different movies in the range $[0-5]$ with a step of 1. The average number of interactions per user is 165.6. The dataset has been put together following MovieLens 100K; only users with at least 20 rated movies have been included. The maximum number of interactions is 2314. The dataset is summarized in Table 4.2.

Interactions	1000209
Density	4.47%
Users	6040
Items	3706
Avg. interactions	165.6
Min. interactions	20
Max. interactions	2314

Table 4.2: MovieLens 1M dataset statistics

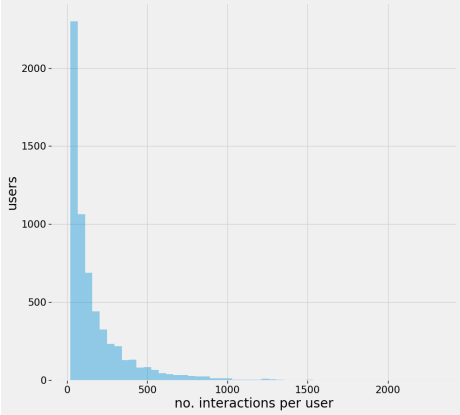


Figure 4.3: MovieLens 1M: distribution of per-user number of interactions

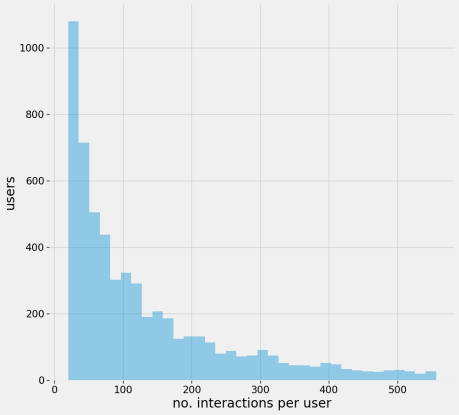


Figure 4.4: MovieLens 1M: distribution of 95th percentile number of interactions

4.3 CiaoDVD

CiaoDVD dataset[23] is hosted by LibRec[1] which is a Java-based library for Recommender Systems. The authors have crawled the entire catalogue of DVDs from the website `dvd.ciao.co.uk`. The dataset contains ratings in the range $[1-5]$ by 17615 users on 16121 for a total of 72345 ratings. The dataset has a density of 0.025%, much sparser than all the other datasets we

[1]https://www.librec.net/datasets.html

considered. The average number of ratings' per user is 4.11, the minimum is 1 and the maximum number of ratings is 1106.

Interactions	72345
Density	0.025%
Users	17615
Items	16121
Avg. interactions	4.11
Min. interactions	1
Max. interactions	1106

Table 4.3: CiaoDVD dataset statistics

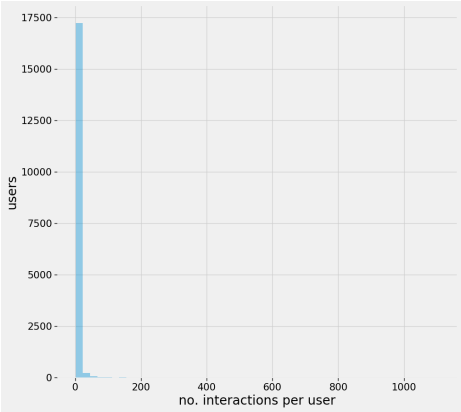


Figure 4.5: CiaoDVD: distribution of per-user number of interactions

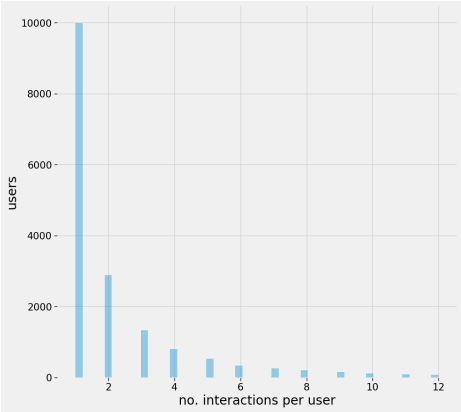


Figure 4.6: CiaoDVD: distribution of 95th percentile per-user number of interactions

4.4 Delicious

Delicious[9] dataset is obtained by the Delicious social bookmarking system and was released in the 2nd International Workshop on Information Heterogeneity and Fusion in Recommender Systems. It contains tuples of the form (user, bookmark, tag) denoting tags users put to bookmarked URLs. To construct the URM required to test our algorithms we keep only the pairs (user, bookmark) as *implicit* interactions. There are in total 104799 interactions between 1867 users and 69223 bookmarks. The density of the URM matrix is 0.081% and the mean number of tagged bookmarks per user is 56.13. This dataset also has users with only one interaction as the minimum number of interactions per user. Table 4.4 summarizes the dataset

statistics.

Interactions	104799
Density	0.081%
Users	1867
Items	69223
Avg. interactions	56.13
Min. interactions	1
Max. interactions	95

Table 4.4: Delicious dataset statistics

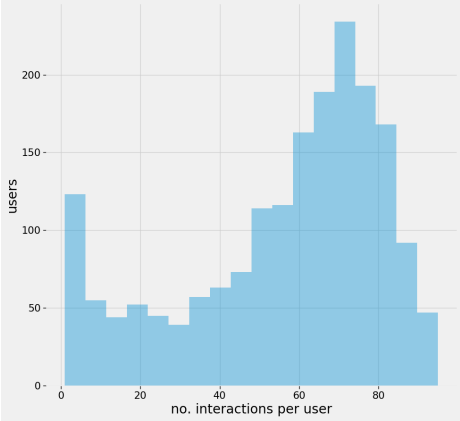


Figure 4.7: Delicious: distribution of per-user number of interactions

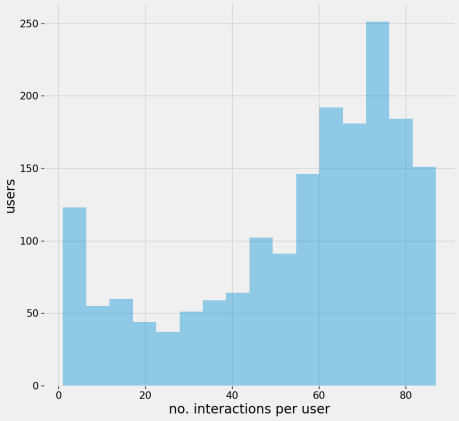


Figure 4.8: Delicious: distribution of 95th percentile per-user number of interactions

4.5 LastFM

LastFM[9] is another dataset made available in the 2nd International Workshop on Information Heterogeneity and Fusion in Recommender Systems. It contains music artist listening information from almost 1900 users for 17632 unique artists in the format (user, artist, listeningCount) where *listeningCount* represents how many times the user listened to the specific artist. For our evaluations we consider only the pairs (user, artist) as *implicit* feedback. The URM from this dataset is 0.28% dense with 92834 pairs. On average a single user has listened to 49.07 different artists with a maximum of 50 and every user has listened to at least 1 artist. Table 4.5 summarizes the statistical information of LastFM dataset.

Interactions	92834
Density	0.28%
Users	1900
Items	17632
Avg. interactions	49.07
Min. interactions	1
Max. interactions	50

Table 4.5: LastFM dataset statistics

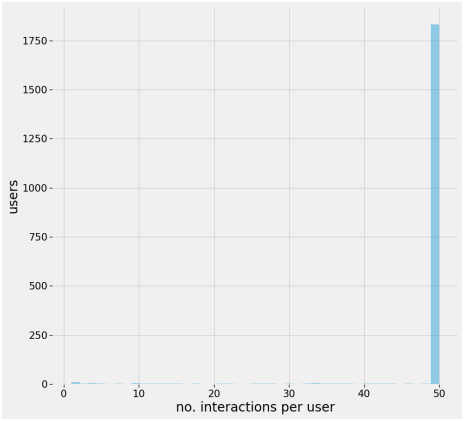


Figure 4.9: LastFM: distribution of per-user number of interactions

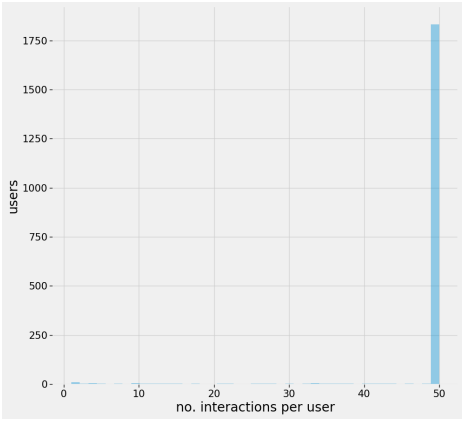


Figure 4.10: LastFM: distribution of 95th percentile per-user number of interactions

4.6 Dataset popularity bias

The effect item popularity has on RS is well known in the RS and IR communities [7]. In [15] the authors show that a very small percentage of the total items in datasets like MovieLens 1M and Netflix [8] are attributed a large portion of all interactions. More specifically they show that 33% of the ratings are on 1.7% and on 5.5% of the items in Netflix and MovieLens 1M datasets, respectively. Such distribution of the total ratings of a datasets is also known by the name *long-tail* distribution [3, 15] and we follow [15] in denoting the most rated items (the ones that account for 33% of the total ratings) as *short-head* items. Figure 4.11 depicts the long tail distribution of dataset MovieLens 1M.

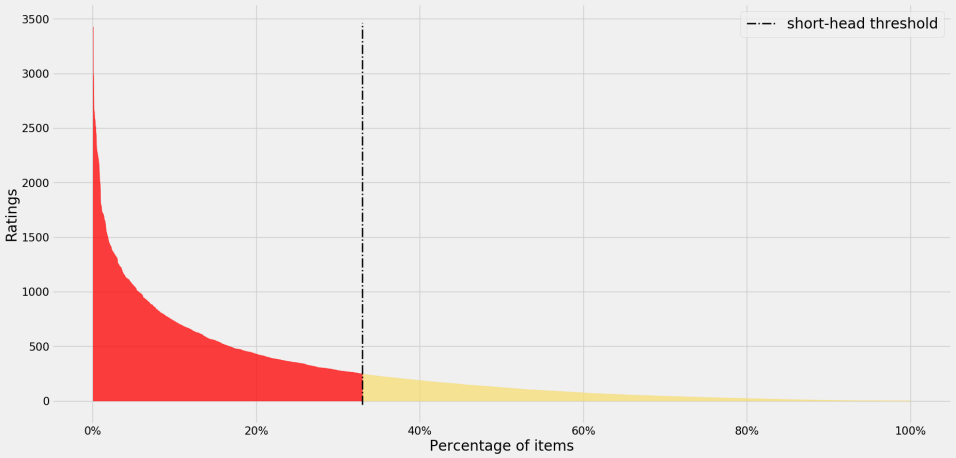


Figure 4.11: The long tail distribution of the ratings of MovieLens 1M. The x-axis shows the percentage of items in decreasing order of number of ratings per item.

In order to compare the popularity bias between datasets we can use the Gini coefficient/index [20]. It was initially introduced as a way to quantify the income (or wealth) distribution in a population sample. We can adapt this metric to our domain by assuming the number of ratings to be the amount of wealth of each item. In this way Gini coefficient can give us an insight on the distribution of the ratings among the items. This index is tighly related to a graphical representation of the inequality in income/wealth, the Lorenz curve. This curve maps the cumulative percentage of people from the population sample to their cumulative wealth. A uniformly distributed income is obtained by the line $y = x$, the 45°line with respect to the horizontal axis. A deviation from this line indicates a skewed distribution of income and more informally an inequality between the population income. When referring to the Gini index in relation to the Lorenz curve [34], this index

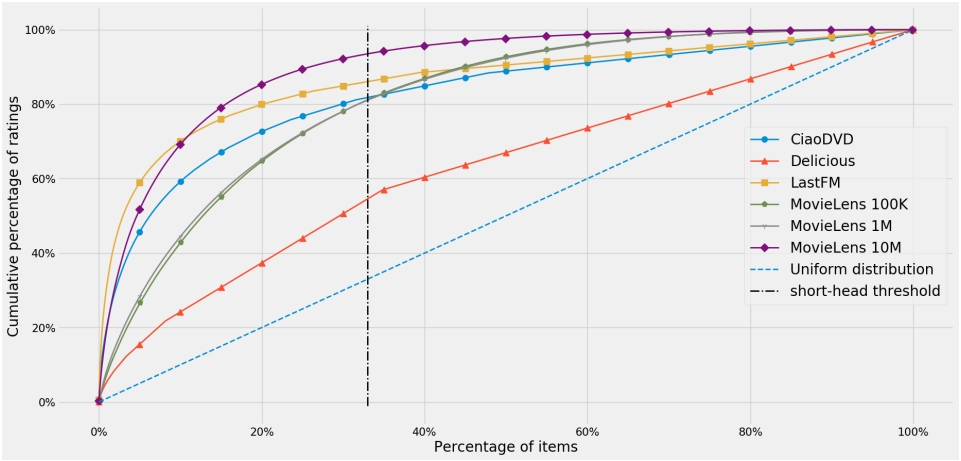


Figure 4.12: The inverse Lorenz curve for each of the considered datasets. The x-axis shows the percentage of items ordered in decreasing order according to the number of ratings. The y-axis shows the cumulative number of ratings for a specific portion of items. The blue dashed line is the curve of equal distribution of ratings among the items in any of datasets. The dash-dotted black vertical line shows the threshold (33%) for the short-head items. For 5 out of 6 datasets, items in this threshold account for more than 80% of the total ratings.

measures the area between the Lorenz curve of the population sample and the curve indicating a uniformly distributed income. The range of the index is in $[0 - 1]$ with low values showing a tendency toward equal distribution and high values a high inequality.

We use the Gini index as a way to quantify how skewed is the distribution of ratings [38] in each of our datasets and depict this with the inverse Lorenz curve in figure 4.12. We can clearly see from the figure that a small portion of items holds a great majority of total ratings. 20% of items in both LastFM and CiaoDVD account for more than 75% of the ratings and for 4 out of 5 datasets, 33% of the items account for more than 80% of the total ratings with only Delicious dataset not showing a high degree of popularity bias. Finally we give the Gini index for each dataset in table 4.6.

Dataset	ML100K	ML1M	Ciao	LastFM	Delicious
Gini Index	0.63	0.63	0.65	0.73	0.25

Table 4.6: Gini index for the considered datasets.

Chapter 5

Experiments

In this chapter we detail the experimental setup along with the experiments we ran to evaluate our model. In the first experiment we compare GANMF with baseline models on every dataset presented on chapter 4. As a second experiment we perform an ablation study on some parts of the model with the aim of understanding whether the design choices laid out in chapter 3 actually describe its performance. The experiments are designed to answer the following research questions:

1. How does GANMF compare to other traditional MF and neighborhood-based baselines?
2. How does GANMF compare to CFGAN which also uses vector-wise training within the GAN framework?
3. How does the choice of using an autoencoder as a discriminator affect the performance of GANMF?
4. How effective is feature matching in enforcing conditional generation of user historical profiles?
5. What is the effect of using Deep Neural Networks (DNN) for both discriminator and generator of GANMF?

5.1 Implementation details & Experimental Setup

We implemented our model using Python programming language [49], version 3.6. Since the input to a RS is usually a URM or UCM/ICM we had to deal with sparse matrices and sparse vectors. To help in computations involving such objects we used scientific libraries like *Numpy* [50] and *Scipy*

[30]. SVD computation for one of the baselines was calculated with the library *Scikit-learn* [41]. To implement GAN-based and neural network models we used the well-known *Tensorflow* framework [1], version 1.12. Tensorflow builds a computational graph for the neural network and performs automatic differentiation so we did not need to implement the complex gradients by hard coding them. For some of the baselines we used the framework written by Maurizio Ferrari Dacrema[1] in the context of the Recommender Systems course of Master of Science degree at Politecnico di Milano.

All the experiments were performed on a single machine with a 2-core Intel G4560 CPU, 8GB DDR4 Ram and an NVIDIA GTX 770 4GB GPU. The full code of the experiments and all the results can be found at `bitbucket.org/ervindervishaj/tesi`.

Hyperparameter Tuning

One of the experiments we perform is the comparison of GANMF with other baselines across a variety of metrics (see section 2.4.3) at different cutoffs. Our model and every baseline have additional parameters beside the ones that take part in the learning process. These special parameters are usually called *hyperparameters* and for every different value of each of them the learning process is a complete separate problem. We use bayesian optimization [29, 4] to find the hyperparameters of each model. Bayesian optimization is a technique used to optimize black-box functions; functions whose first and second derivatives we cannot obtain and thus cannot make use of approaches like gradient descent and quasi-Newton methods [17] and are very expensive in terms of time and obtaining evaluations at different points. Bayesian optimization builds its own internal model, a surrogate, of the function we are trying to optimize. This surrogate takes the form of a Gaussian Process (GP) Regressor and its parameters are updated with each new evaluation of the function.

In this thesis we applied bayesian optimization through the Python library *Scikit-optimize* [25] which provides functions for GP and random-search function optimization. Both these functions take in a set of hyperparameters along with their respective value ranges, e.g. regularization coeffient α in the range $[10^{-6}, 0]$. For each algorithm we optimized mAP@5 on a holdout set and we ran 50 evaluations where the first 10 were random evaluations in order for GP to construct a prior for the functions.

[1] `https://github.com/MaurizioFD/RecSys_Course_AT_PoliMi`

Dataset splitting

In order to achieve fair comparison among all involved algorithms it is important for them to be trained and evaluated on the same training, validation and test sets. Initially each full URM is built as explained in section 4.1, i.e. only users with at least 2 ratings are included in the full URM. Then we reserved 20% of the user-item interactions as a test set on which the final performance of the algorithms is evaluated. In order to assess how good each set of candidate values for the hyperparameters of a model is, we need a holdout validation set on which to evaluate the model with the selected hyperparameters. To accomodate this optimization process we reserved 25% of the remaining 80% of the user-item interactions for the validation set. At this point in the splitting of the initial dataset, we are left with 60% of the interactions. This would constitute our training set but we need to account for some of the algorithms that need early stopping mechanism (section 3.4.2). We reserved only 15% of the training set interactions as an early stopping validation set and all the remaining data is used to train the models for the optimization process. Figure 5.1 visualizes the whole splitting process.

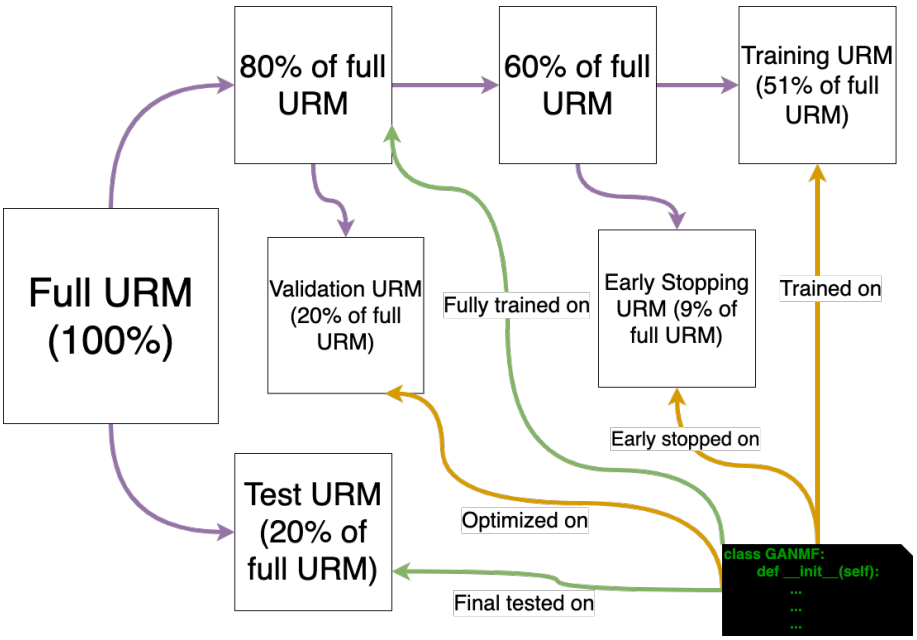


Figure 5.1: Dataset splitting process for hyperparameter optimization and final training and testing. Starting from the full URM the purple arrows indicate a split of the initial dataset. Orange arrows indicate an interaction of the algorithm with the datasets during hyperparameter optimization. Green arrows indicate an interaction of the algorithm with the datasets for the final training and testing.

5.2 Comparison with baselines

In this section we focus on our first and second research questions. In chapter 3 we presented our model GANMF as a MF approach which utilizes a discriminator to learn the latent factors of both users and items with the aim of solving the *top-N* recommendation problem. In this first experiment we are interested in comparing GANMF with more traditional approaches mainly built on top of MF technique, one GAN-based model and a nearest-neighbor machine learning technique, all of which are used for the same recommendation problem. As baselines we include the following algorithms:

- Top-popular, non-personalized approach that recommends the most popular items and that will serve as a bottom baseline.
- *PureSVD* [15].
- Matrix factorization with the *BPR* [42] training criterion.
- WRMF [27].
- SLIM [39] with *BPR* [42] training criterion.
- CFGAN [11] user-based.
- CFGAN [11] item-based.

We have intentionally omitted IRGAN [51] algorithm from our comparison because the authors of CFGAN show that it performs worse than CFGAN. Also, more importantly, IRGAN's training phase is characterized by the REINFORCE algorithm which is very expensive in terms of computation.

In analogy with CFGAN, our model also has two training modes; a user-based mode and an item-based mode. We evaluate both methods and separately optimize and compare each of them along with the other baselines. As a standard GANMF model we use the model with a **single layer autoencoder** with **linear activations** as discriminator and with **2 embedding layers** as generator. The final generated historical profile from the generator is the usual matrix multiplication of the embedding layers, a linear operation employed by the traditional MF technique.

In the subsequent sections we discuss the results of the algorithms evaluated on the test set of each dataset separately. We compare the algorithms on the basis of 5 metrics at 4 different cutoffs; we report the *precision*, *recall*, *nDCG*, *mAP* and *item coverage* on cutoffs of 5, 10, 20 and 50 (section

2.4.3). We chose these metrics since they provide a good analysis for the algorithms in both prediction accuracy (precision and recall) and ranking accuracy (nDCG and mAP). The different cutoffs give us an indication how the algorithms behave with increasing recommendation list length. Coverage metric can give us some insights on how capable is each algorithm in handling item popularity bias of the datasets.

5.2.1 MovieLens 100K

We present the results for MovieLens 100K in tables 5.1, 5.2, 5.3 and 5.4. The results for cutoff 5 show that MF-based approach PureSVD performs better than the other models. Interestingly both PureSVD and WRMF perform better than the machine learning neighborhood approach SLIM-BPR. The difference between PureSVD and SLIM-BPR and can be explained to some degree by the coverage metric; SLIM-BPR covers in its recommendations almost 5% of all the available items, thus providing a very narrow recommendation to all users. PureSVD and WRMF are able to surpass the precision of SLIM-BPR while at the same time offering a much higher coverage both at 16%, most probably finding the right items to recommend also to users that are not only interested in the most popular items given this dataset's popularity bias as indicated by a Gini index of 0.63 (section 4.6). This argument is further supported by the higher recall of both MF-based models. Somewhat disappointingly MF-BPR performs slightly better than the non-personalized approach on some of the metrics.

PureSVD perform also better than both GAN-based models with GANMF-i ranking 2$^{\text{nd}}$ in all the metrics. In table 5.1 we also give the relative change of GANMF-i to the best performing model. We can observe that GANMF is able to compete pretty well in all metrics, being within $[0.1\% - 1.6\%]$ of the results of the best model at cutoff 5. The other variant, GANMF-u, has a performance comparable to WRMF however providing less item coverage. Both GANMF model can surpass SLIM-BPR in all recommendation metrics while at the same time providing almost 3 times as much item coverage. Both models also are very close to one another with GANMF-i performing slightly better at the cost of 1% less coverage.

On the other hand, comparing the GAN-based models we see that GANMF variants provide better performance than CFGAN variants in all metrics beside item coverage. The difference in results between user-based variants is greater than between item-based variants. A large gap in performance can be seen also between CFGAN variants, something that is not observed for GANMF models. Something to be pointed out is the high coverage both

CFGAN models exhibit, twice as much when compared to GANMF and MF-based approaches. We believe this is due to the *CF methods* used by CFGAN. As mentioned in section 2.6.2, CFGAN uses zero reconstruction regularization and partial masking as a way for the generator to be able to reconstruct more accurate user/item profiles. Zero reconstruction, as implied by the name, aims to reconstruct a portion of the sparsity of the profile. Using this additional loss for the generator CFGAN learns to zero out in the generator output the items that the user has not interacted with. If we consider a user that has previously rated not so popular items, when his/her profile is processed by the generator, it learns to give a 0 score to popular items. Given enough users that like less popular items, the item coverage of the generator is increased.

Continuing the discussion with the results at cutoffs 10, 20 and 50 we can see that both PureSVD and GANMF-i perform almost identical to one another, also in terms of item coverage. Both methods have a slight advantage over WRMF we can see that the same arguments hold also for longer recommendation lists. WRMF shows stronger prediction accuracy compared to PureSVD while the latter has a slightly better ranking accuracy as measured by mAP. GANMF variants still perform better than CFGAN even in longer recommendation lists.

Overall in this dataset GANMF is able to provide comparable results with PureSVD and slightly better than both WRMF and SLIM-BPR. Morever both GANMF variants can surpass CFGAN models while the latter can provide much better coverage.

Algorithm	**Prec@5**	**Rec@5**	**nDCG@5**	**mAP@5**	**Cov@5**
Top popular	0.2165429	0.0708332	0.0977239	0.1449216	0.0172414
PureSVD	**0.4201485**	**0.1424799**	**0.2102587**	**0.3512042**	0.1599287
WRMF	0.4053022	0.1355566	0.2002147	0.3331628	0.1593341
MF-BPR	0.2021209	0.0859918	0.1155711	0.1426535	0.0368609
SLIM-BPR	0.366702	0.1293142	0.1872001	0.2996604	0.0487515
CFGAN-u	0.2462354	0.0815232	0.1177102	0.1778296	0.3061831
CFGAN-i	0.3546129	0.1253011	0.1759757	0.275205	**0.313912**
GANMF-u	0.3959703	0.1360042	0.1928307	0.3210631	0.146849
GANMF-i	0.4197243	0.1421869	0.2068968	0.3481637	0.137931
Relative Change	-0.10%	-0.21%	-1.60%	-0.87%	-53.22%

Table 5.1: Results for cutoff 5 on MovieLens 100K. Higher values are better. Best results are in bold. Relative change reports the difference between best GANMF variant and best performing baseline, with negative values denoting worse performance of GANMF.

Algorithm	Prec@10	Rec@10	nDCG@10	mAP@10	Cov@10
Top popular	0.1969247	0.1207682	0.1368754	0.1166896	0.029132
PureSVD	0.3551432	0.2275842	**0.2759503**	**0.2893783**	0.2098692
WRMF	0.3511135	0.2230234	0.2675774	0.2781397	0.2057075
MF-BPR	0.1686108	0.1322022	0.1496186	0.1124854	0.0618312
SLIM-BPR	0.2971368	0.196986	0.2385696	0.2360697	0.0766944
CFGAN-u	0.2174973	0.14153	0.1624927	0.1419956	**0.3929845**
CFGAN-i	0.3034995	0.207585	0.2365852	0.223579	0.3876338
GANMF-u	0.3371156	0.218114	0.255901	0.2628942	0.1967895
GANMF-i	**0.3565217**	**0.2282473**	0.2729096	0.288402	0.1920333
Relative Change	0.39%	0.29%	-1.10%	-0.34%	-49.92%

Table 5.2: Results for cutoff 10 on MovieLens 100K. Higher values are better. Best results are in bold. Relative change reports the difference between best GANMF variant and best performing baseline.

Algorithm	Prec@20	Rec@20	nDCG@20	mAP@20	Cov@20
Top popular	0.1549841	0.1703185	0.1706868	0.0959901	0.048157
PureSVD	**0.2859491**	**0.3480285**	**0.3500764**	**0.2532081**	0.2788347
WRMF	0.2821845	0.3445215	0.3414857	0.2424792	0.283591
MF-BPR	0.1384942	0.2085422	0.1929	0.1036458	0.1070155
SLIM-BPR	0.2373277	0.2993396	0.3008329	0.20232	0.1165279
CFGAN-u	0.181018	0.2238151	0.2125324	0.1238725	**0.4845422**
CFGAN-i	0.2461294	0.3198261	0.3043358	0.1973178	0.4583829
GANMF-u	0.2734358	0.3321748	0.3272267	0.2303143	0.2758621
GANMF-i	0.2827678	0.3406212	0.3432445	0.2487557	0.2526754
Relative Change	-1.11%	-2.13%	-1.95%	-1.76%	-43.07%

Table 5.3: Results for cutoff 20 on MovieLens 100K. Higher values are better. Best results are in bold. Relative change reports the difference between best GANMF variant and best performing baseline.

Algorithm	Prec@50	Rec@50	nDCG@50	mAP@50	Cov@50
Top popular	0.1164581	0.2954592	0.2357702	0.0923372	0.1010702
PureSVD	**0.1934252**	0.5372567	**0.4478853**	**0.2436833**	0.4203329
WRMF	0.1904348	0.532496	0.4380601	0.233905	0.4268728
MF-BPR	0.1003606	0.3389586	0.2548068	0.1074788	0.2128419
SLIM-BPR	0.1594486	0.4625969	0.3832263	0.1915255	0.1837099
CFGAN-u	0.1346978	0.3978162	0.296069	0.1243592	**0.6070155**
CFGAN-i	0.1710498	0.5021467	0.3971656	0.1967767	0.5648038
GANMF-u	0.1834571	0.5169792	0.4205572	0.2200193	0.4173603
GANMF-i	0.1931495	**0.5375202**	0.4434139	0.2400581	0.3989298
Relative Change	-0.14%	0.05%	-1.00%	-1.49%	-31.24%

Table 5.4: Results for cutoff 50 on MovieLens 100K. Higher values are better. Best results are in bold. Relative change reports the difference between best GANMF variant and best performing baseline.

5.2.2 MovieLens 1M

MovieLens 1M is the second MovieLens dataset that we test our algorithm on. We give the results on the test sets on tables 5.5, 5.6, 5.7 and 5.8. In this dataset our model performs better than all other baselines in all metrics with precision 4-5% better in all cutoffs and mAP approximately 8% on average better than the second best performing baseline, SLIM-BPR. Moreover GANMF-i beside the higher performance has also higher coverage than SLIM-BPR and is second only to WRMF. WRMF on the other hand performs better than PureSVD on all metrics and cutoffs. MF-BPR also in this dataset struggles to match the performance of top-popular approach.

Compared to CFGAN-u, GANMF-i's metrics at cutoff 5 are approximately 3 times better. In comparison with CFGAN-u all the metrics at cutoff 10 of GANMF-i are approximately one order of magnitude higher, even at a very similar coverage@5. GANMF-u on the other hand surpasses both CFGAN models at an even higher coverage@10. Differently from MovieLens 100K but in line with CF algorithms, CFGAN-i has much lower coverage than CFGAN-u indicating that the former is biasing its recommendations towards more popular items than the later. This is evident also when considering the performance of the non-personalized baseline where better results are observed on all metrics compared to both CFGAN-u and CFGAN-i. Also differently from GANMF variants, CFGAN-i and CFGAN-u results seems to differ a lot between them on the same metric and cutoff.

We bring to attention also the traditional MF-based approaches. MF-BPR as in MovieLens 100K performs on par with the non-personalized base-

line with only slight improvements on recall and nDCG metrics. PureSVD and WRMF perform similarly to one another with WRMF having better prediction accuracy and PureSVD better ranking according to mAP@5 and mAP@10 metrics. Also, the discrepancy between these two algorithms increases with increasing recommendation list cutoff. Compared to GANMF-i variants both models perform worse. The coverage at cutoff 5 of WRMF is the highest of all considered algorithms and that could explain the drop in performance. However its coverage is only 2% greater than that of GANMF-u whose performance is equally higher. In table 5.7 and 5.8, different from the previous cutoffs, we can see that coverage of GANMF-u is slightly higher than that of WRMF while still maintaining both higher prediction and ranking accuracy. Finally we see that both traditional MF-based models and GANMF perform similarly in MovieLens 1M as in MovieLens 100K. We recall from section 4.6 that both these datasets have the same Gini index and hence the same bias toward popular items in terms of ratings.

Algorithm	**Prec@5**	**Rec@5**	**nDCG@5**	**mAP@5**	**Cov@5**
Top popular	0.214404	0.0419873	0.0706967	0.1544478	0.0118726
PureSVD	0.3995364	0.0923932	0.1500131	0.3272192	0.1200756
WRMF	0.4009272	0.0997203	0.1572062	0.3202506	**0.2199137**
MF-BPR	0.2181457	0.0439298	0.074034	0.1574015	0.069347
SLIM-BPR	0.4140397	0.1068285	0.1668732	0.3346971	0.1775499
CFGAN-u	0.0417219	0.0104807	0.0152957	0.0202545	0.1173772
CFGAN-i	0.1834768	0.0352682	0.0621318	0.1331042	0.053157
GANMF-u	0.4320199	**0.1083758**	**0.1692069**	0.3539966	0.2026444
GANMF-i	**0.4323179**	0.1053124	0.1690019	**0.358333**	0.1392337
Relative Change	4.41%	1.45%	1.40%	7.06%	-7.85%

Table 5.5: Results for cutoff 5 on MovieLens 1M. Higher values are better. Best results are in bold. Relative change reports the difference between best GANMF variant and best performing baseline.

5.2.3 CiaoDVD

We present in this section the results on the CiaoDVD dataset. This is the sparsiest dataset used in this work with a density at 0.025%. Its full URM has a square-ish shape of 17615 users $\times$ 16121 items and the Gini index is 0.65, so ratings are biased toward popular items. Results on this dataset, tables 5.9, 5.10, 5.11 and 5.12, show WRMF as the top performer. SLIM-BPR comes as the best second algorithm. However we can see that SLIM-BPR's coverage is the greatest among all 4 different cutoffs with almost 5-6 times

Algorithm	Prec@10	Rec@10	nDCG@10	mAP@10	Cov@10
Top popular	0.1837417	0.0681797	0.0945611	0.1173533	0.017809
PureSVD	0.3434272	0.1488452	0.1995704	0.262581	0.1654074
WRMF	0.3472351	0.1624667	0.2104642	0.259208	**0.2701025**
MF-BPR	0.1861921	0.0739437	0.1000572	0.1177731	0.1214247
SLIM-BPR	0.3528808	0.1712493	0.2209655	0.2680273	0.2420399
CFGAN-u	0.0373013	0.0170446	0.0208279	0.0136246	0.1896924
CFGAN-i	0.1527649	0.0579876	0.0821061	0.0963071	0.0812196
GANMF-u	0.3702483	**0.1742579**	**0.2252712**	0.28728	0.264436
GANMF-i	**0.372351**	0.1692659	0.2243996	**0.2916971**	0.1826767
Relative Change	5.52%	1.76%	1.95%	8.83%	-2.10%

Table 5.6: Results for cutoff 10 on MovieLens 1M. Higher values are better. Best results are in bold. Relative change reports the difference between best GANMF variant and best performing baseline.

Algorithm	Prec@20	Rec@20	nDCG@20	mAP@20	Cov@20
Top popular	0.1561921	0.1151276	0.1271463	0.0918708	0.0302213
PureSVD	0.282798	0.2299902	0.2566088	0.2158214	0.2182947
WRMF	0.2898593	0.2517595	0.2722827	0.2187244	0.3297356
MF-BPR	0.1534354	0.1189983	0.1311217	0.0904318	0.220993
SLIM-BPR	0.2872765	0.2586859	0.2808407	0.2234922	0.3262277
CFGAN-u	0.0421275	0.0381151	0.0345674	0.0117824	0.293578
CFGAN-i	0.1230215	0.0908527	0.1050038	0.0721666	0.1219644
GANMF-u	**0.3053808**	**0.2653345**	**0.2885205**	0.2417735	**0.3416082**
GANMF-i	0.304404	0.2552169	0.2852354	**0.2422803**	0.2417701
Relative Change	5.35%	2.57%	2.73%	8.41%	3.60%

Table 5.7: Results for cutoff 20 on MovieLens 1M. Higher values are better. Best results are in bold. Relative change reports the difference between best GANMF variant and best performing baseline.

Algorithm	Prec@50	Rec@50	nDCG@50	mAP@50	Cov@50
Top popular	0.114394	0.2008153	0.1755463	0.0743705	0.0631409
PureSVD	0.2040563	0.3817655	0.3427192	0.1890541	0.3224501
WRMF	0.2114934	0.4140289	0.3642318	0.199202	0.4384781
MF-BPR	0.1126325	0.2121367	0.1819059	0.0740611	0.4541284
SLIM-BPR	0.2028411	0.410478	0.3664565	0.2000196	0.4589854
CFGAN-u	0.0392384	0.0873179	0.0599086	0.0123139	**0.5010793**
CFGAN-i	0.0904007	0.1565923	0.1422478	0.0584476	0.2363734
GANMF-u	**0.2184503**	**0.4275134**	**0.3809177**	**0.2164192**	0.4695089
GANMF-i	0.2166623	0.4125733	0.3749247	0.2142928	0.3537507
Relative Change	3.29%	3.26%	3.95%	8.20%	-6.30%

Table 5.8: Results for cutoff 50 on MovieLens 1M. Higher values are better. Best results are in bold. Relative change reports the difference between best GANMF variant and best performing baseline.

that of WRMF and hence provides more diverse recommendations. WRMF performs also better than PureSVD.

In regards to GANMF we can see from the results that GANMF-i is the third best model beating PureSVD in all metrics at all cutoffs with greater coverage for the item-based variant. GANMF-s coverage is also 2-3 times greater than that's of WRMF.

We compare now the GAN-inspired models. Both CFGAN variants have lower coverage than GANMF, which reaches almost twice the coverage on some cutoffs. Despite this we also see both GANMF versions beating all metrics of both CFGAN models. Interestingly the CFGAN variant with greater coverage, CFGAN-i, performs better than CFGAN-u. The reverse behavior is observed for GANMF where the best performing variant has the highest coverage.

5.2.4 LastFM

LastFM, different from the other datasets seen so far, is a music dataset. However not surprisingly, as a music datasets it also shows high item popularity bias [10] also indicated by its Gini index of 0.73. Its full URM has shape 1900 users $\times$ 17632 items, so almost 10 times more items than users. We were expecting this disparity between the length of user and item profiles to show on the performances of our GANMF variants. However as we can see from the results on tables 5.13, 5.14, 5.15 and 5.16 both GANMF-u and GANMF-i produce similarly accurate recommendations. The performance of our model is the highest among all baselines including WRMF and SLIM-

Algorithm	Prec@5	Rec@5	nDCG@5	mAP@5	Cov@5
Top popular	0.0093078	0.0301836	0.0218311	0.0173683	0.0008684
PureSVD	0.0122181	0.0352304	0.0259442	0.0206842	0.0714596
WRMF	**0.0169376**	**0.0504526**	**0.0363511**	**0.0284788**	0.0362881
MF-BPR	0.0079969	0.0251085	0.0188852	0.0151641	0.0012406
SLIM-BPR	0.015915	0.048574	0.0355339	0.0284408	**0.2934681**
CFGAN-u	0.0101206	0.0328725	0.0230227	0.0180462	0.0011786
CFGAN-i	0.0104615	0.0285493	0.0205241	0.0153599	0.0578128
GANMF-u	0.01387	0.0420354	0.0298209	0.0230707	0.0535947
GANMF-i	0.0128736	0.0356792	0.0267557	0.0214877	0.1059488
Relative Change	-18.11%	-16.68%	-17.96%	-18.99%	-63.90%

Table 5.9: Results for cutoff 5 on CiaoDVD. Higher values are better. Best results are in bold. Relative change reports the difference between best GANMF variant and best performing baseline.

Algorithm	Prec@10	Rec@10	nDCG@10	mAP@10	Cov@10
Top popular	0.0076691	0.0492416	0.0284671	0.0196393	0.0015508
PureSVD	0.0097273	0.0548152	0.0330951	0.022942	0.1050803
WRMF	**0.0132931**	**0.0788733**	**0.0465245**	**0.0317792**	0.0560759
MF-BPR	0.0062271	0.0382356	0.023579	0.0166721	0.0021711
SLIM-BPR	0.0119428	0.0703845	0.0434444	0.03079	**0.3987966**
CFGAN-u	0.0078526	0.0502189	0.0291485	0.0200167	0.0017989
CFGAN-i	0.0089932	0.0498985	0.02824	0.0179044	0.0869673
GANMF-u	0.011366	0.0648433	0.0381765	0.0258148	0.1134545
GANMF-i	0.0100026	0.0533817	0.0333071	0.0234155	0.1547671
Relative Change	-14.50%	-17.79%	-17.94%	-18.77%	-61.19%

Table 5.10: Results for cutoff 10 on CiaoDVD. Higher values are better. Best results are in bold. Relative change reports the difference between best GANMF variant and best performing baseline.

Algorithm	Prec@20	Rec@20	nDCG@20	mAP@20	Cov@20
Top popular	0.0060304	0.0748749	0.035647	0.0213848	0.0029155
PureSVD	0.0076691	0.0837869	0.0414224	0.024936	0.1387011
WRMF	**0.0103173**	**0.1162985**	**0.0573421**	**0.0344711**	0.0868432
MF-BPR	0.0047063	0.057897	0.0289933	0.0179722	0.004032
SLIM-BPR	0.008908	0.1020194	0.0524146	0.032958	**0.5131195**
CFGAN-u	0.0064696	0.0818359	0.0378881	0.0222032	0.0032256
CFGAN-i	0.0075905	0.0837763	0.0378648	0.0202608	0.1277216
GANMF-u	0.0085212	0.0945504	0.046718	0.0279302	0.2396253
GANMF-i	0.0076036	0.080545	0.0410068	0.0251971	0.2177284
Relative Change	-17.41%	-18.70%	-18.53%	-18.98%	-53.30%

Table 5.11: Results for cutoff 20 on CiaoDVD. Higher values are better. Best results are in bold. Relative change reports the difference between best GANMF variant and best performing baseline.

Algorithm	Prec@50	Rec@50	nDCG@50	mAP@50	Cov@50
Top popular	0.0044048	0.1385646	0.0494453	0.0235451	0.005955
PureSVD	0.0050944	0.1347948	0.0530028	0.0267462	0.2078035
WRMF	**0.0065836**	**0.1813807**	**0.0719227**	**0.0367931**	0.1476335
MF-BPR	0.0032853	0.1010213	0.0383544	0.0193877	0.0089945
SLIM-BPR	0.0059518	0.1600545	0.0656625	0.0349981	**0.6520687**
CFGAN-u	0.0044992	0.1408453	0.0506728	0.0241884	0.0066373
CFGAN-i	0.0054457	0.1482506	0.0523911	0.0226197	0.2256064
GANMF-u	0.0055926	0.1492362	0.0592024	0.0299301	0.4378761
GANMF-i	0.0050236	0.1251562	0.0513823	0.0267774	0.3358973
Relative Change	-15.05%	-17.72%	-17.69%	-18.65%	-32.85%

Table 5.12: Results for cutoff 50 on CiaoDVD. Higher values are better. Best results are in bold. Relative change reports the difference between best GANMF variant and best performing baseline.

BPR which on the previous datasets appeared to be the strongest among our selected baselines. GANMF achieves also higher coverage along all cut-offs compared to MF-based algorithms. PureSVD's coverage on this dataset is one of the lowest we have seen in this work by this algorithm, only 0.76% on a recommendation length of 5. Both variants of GANMF have higher coverage than the other baselines besides CFGAN-i and ranking accuracy according to mAP is consistently 7% higher than the next best performing baseline WRMF.

CFGAN models are both weaker compared to GANMF variants in all metrics and cutoffs. CFGAN-u has a slightly better performance compared to CFGAN-i but that comes at the cost of a quarter of the coverage of the former. Both CFGAN models are also less accurate in both prediction and ranking compared to traditional MF-approaches and also compared to SLIM-BPR. CFGAN-u has a very comparable performance with PureSVD and also a much higher coverage of at least a factor of 3.

Algorithm	**Prec@5**	**Rec@5**	**nDCG@5**	**mAP@5**	**Cov@5**
Top popular	0.0771762	0.0397706	0.0579687	0.0523638	0.0010209
PureSVD	0.2022293	0.1033268	0.1472857	0.1505213	0.0076565
WRMF	0.245966	0.1258021	0.1788593	0.1847563	0.0227427
MF-BPR	0.1130573	0.0584839	0.0846587	0.0855603	0.0068058
SLIM-BPR	0.220276	0.1124237	0.1615952	0.1664171	0.0352768
CFGAN-u	0.1954352	0.1009801	0.1432142	0.1405215	0.029662
CFGAN-i	0.1928875	0.0992035	0.1372855	0.1358475	**0.08201**
GANMF-u	**0.2607219**	**0.1338915**	**0.1914327**	**0.2009306**	0.038623
GANMF-i	0.2562633	0.1318541	0.1867868	0.1935747	0.044578
Relative Change	6.00%	6.43%	7.03%	8.75%	-45.64%

Table 5.13: Results for cutoff 5 on LastFM. Higher values are better. Best results are in bold. Relative change reports the difference between best GANMF variant and best performing baseline.

5.2.5 Delicious

We report the performances on the last dataset, Delicious, in the tables 5.17, 5.18, 5.19 and 5.20. We found these results peculiar because differently from the other datasets, WRMF performs substantially better than PureSVD; almost 7 times better across metrics Prec@5, Rec@5 and nDCG@5 and almost 10 times in mAP@5. WRMF performs better than all other baselines but at a less steep difference. We can see that SLIM-BPR provides higher coverage@5 than WRMF at 2% difference but it comes at the cost of 30% of

Algorithm	Prec@10	Rec@10	nDCG@10	mAP@10	Cov@10
Top popular	0.0662951	0.0680608	0.0777387	0.0382397	0.001588
PureSVD	0.1512739	0.1526964	0.1819721	0.1042772	0.0107191
WRMF	0.186518	0.1890005	0.2231374	0.1303702	0.0309097
MF-BPR	0.0841295	0.0860107	0.1039224	0.0582924	0.0126475
SLIM-BPR	0.1664013	0.1689499	0.2011444	0.1144614	0.0593807
CFGAN-u	0.1492569	0.1524582	0.1792991	0.0981531	0.0397573
CFGAN-i	0.1507431	0.1549922	0.1759522	0.097336	**0.1110481**
GANMF-u	**0.1937898**	0.1968786	**0.2356293**	**0.1391564**	0.0565449
GANMF-i	0.1937367	**0.1973378**	0.2326003	0.1357974	0.0623866
Relative Change	3.90%	4.41%	5.60%	6.74%	-43.82%

Table 5.14: Results for cutoff 10 on LastFM. Higher values are better. Best results are in bold. Relative change reports the difference between best GANMF variant and best performing baseline.

Algorithm	Prec@20	Rec@20	nDCG@20	mAP@20	Cov@20
Top popular	0.0488057	0.0993311	0.0946814	0.0391824	0.0024387
PureSVD	0.1052548	0.2123528	0.2145312	0.1063918	0.0161071
WRMF	0.129034	0.2608253	0.2623477	0.1335815	0.0426497
MF-BPR	0.059448	0.1206518	0.1229142	0.0587035	0.0223457
SLIM-BPR	0.1166932	0.2366589	0.2379893	0.1167282	**0.0952246**
CFGAN-u	0.1085456	0.2198224	0.2161836	0.1028526	0.0560912
CFGAN-i	0.1087049	0.2222522	0.2125373	0.1020133	**0.1487069**
GANMF-u	**0.1360934**	0.2754476	**0.278796**	**0.1437082**	0.085583
GANMF-i	0.1360138	**0.2757899**	0.2755811	0.1405968	0.0896098
Relative Change	5.47%	5.74%	6.27%	7.58%	-39.74%

Table 5.15: Results for cutoff 20 on LastFM. Higher values are better. Best results are in bold. Relative change reports the difference between best GANMF variant and best performing baseline.

Algorithm	Prec@50	Rec@50	nDCG@50	mAP@50	Cov@50
Top popular	0.0341083	0.1725694	0.1258099	0.0447256	0.0047074
PureSVD	0.0632272	0.3184067	0.2602294	0.1195382	0.0266561
WRMF	0.075414	0.3816908	0.3141909	0.1500941	0.0681715
MF-BPR	0.0395966	0.2003551	0.1569982	0.0660411	0.0498525
SLIM-BPR	0.0695223	0.3517537	0.2875025	0.1312701	0.1794465
CFGAN-u	0.0642994	0.3250405	0.2614986	0.1161981	0.0971529
CFGAN-i	0.0642887	0.3260986	0.257554	0.115265	**0.2247051**
GANMF-u	0.0795117	0.4014345	**0.3330374**	**0.1612095**	0.1620349
GANMF-i	**0.0796178**	**0.4019249**	0.3300618	0.1581174	0.1544918
Relative Change	5.57%	5.30%	6.00%	7.41%	-27.89%

Table 5.16: Results for cutoff 50 on LastFM. Higher values are better. Best results are in bold. Relative change reports the difference between best GANMF variant and best performing baseline.

the ranking accuracy according to mAP@5. MF-BPR achieves much better results than top-popular recommender but far from WRMF or SLIM-BPR.

GANMF-u, the best performing among GANMF variants, performs almost 4 times better than PureSVD at cutoffs 5 and 10 but has a lower advantage at cutoff 20 and 50. Even though it presents the highest coverage over all recommendation list lengths it performs at half the accuracy of WRMF in all metrics at cutoff 5. We came across another interesting result when comparing GANMF and SLIM-BPR. In table 5.17 we can see that SLIM-BPR provides better recommendation and ranking accuracy than GANMF-i at a comparable coverage. However in the results of the next cutoffs GANMF performs better and between cutoffs its results are more stable with the recall@10 close to double that of recall@5 given the increase in coverage. We can thus say that SLIM-BPR is more susceptible to the popularity bias because even though it experiences approximately the same increase in coverage, its performance decreases with longer recommendation lists.

Comparing GANMF to CFGAN we see for the first time than CFGAN variants perform better than GANMF's in some metrics like precision, recall and nDCG. However GANMF-i reports better ranking accuracy through mAP@5 and mAP@10 also considering the twice as high coverage.

5.3 Ablation Study

In this section we focus on research questions 3, 4 and 5. We dissect our model GANMF in order to understand whether there is some causal connection between its components and the results presented in section 5.2. We

Algorithm	Prec@5	Rec@5	nDCG@5	mAP@5	Cov@5
Top popular	0.0021786	0.0009668	0.0012767	0.0010004	0.00013
PureSVD	0.0360566	0.0158643	0.0216547	0.0249088	0.0551406
WRMF	**0.2565359**	**0.1174365**	**0.1504931**	**0.2165255**	0.0796412
MF-BPR	0.0139434	0.011767	0.0116898	0.0102059	0.0078442
SLIM-BPR	0.1556645	0.0711403	0.1023254	0.1486908	0.0999379
CFGAN-u	0.1496732	0.0660646	0.0867188	0.1144944	0.0482065
CFGAN-i	0.0004357	0.0001679	0.0002649	0.0002542	0.0191266
GANMF-u	0.1383442	0.0549645	0.0777383	0.1189397	**0.1041995**
GANMF-i	0.120915	0.0501615	0.0684077	0.1032746	0.0914436
Relative Change	-46.07%	-53.20%	-48.34%	-45.07%	4.26%

Table 5.17: Results for cutoff 5 on Delicious. Higher values are better. Best results are in bold. Relative change reports the difference between best GANMF variant and best performing baseline.

Algorithm	Prec@10	Rec@10	nDCG@10	mAP@10	Cov@10
Top popular	0.0020153	0.002054	0.0018988	0.0007785	0.0001878
PureSVD	0.0454248	0.0381012	0.0378445	0.0257323	0.1085766
WRMF	**0.2267974**	**0.1959225**	**0.2095796**	**0.1932031**	0.1413692
MF-BPR	0.0099129	0.0170848	0.0145766	0.0084955	0.0118602
SLIM-BPR	0.0964597	0.0898005	0.1149212	0.092658	0.1703191
CFGAN-u	0.1353486	0.1159111	0.1235349	0.099101	0.088121
CFGAN-i	0.0002179	0.0001679	0.0002649	0.0001452	0.0335727
GANMF-u	0.1251634	0.0952886	0.1096041	0.1021225	**0.180807**
GANMF-i	0.1150327	0.0901394	0.0996606	0.0922297	0.1641073
Relative Change	-44.81%	-51.36%	-47.70%	-47.14%	6.16%

Table 5.18: Results for cutoff 10 on Delicious. Higher values are better. Best results are in bold. Relative change reports the difference between best GANMF variant and best performing baseline.

Algorithm	Prec@20	Rec@20	nDCG@20	mAP@20	Cov@20
Top popular	0.001988	0.0034133	0.0027562	0.0006951	0.0004189
PureSVD	0.0639706	0.0974176	0.0741175	0.0390156	0.1980845
WRMF	**0.147195**	**0.2503321**	**0.2416519**	**0.1730376**	0.2203747
MF-BPR	0.0083878	0.0259708	0.0187067	0.008566	0.0196178
SLIM-BPR	0.0621187	0.1147779	0.1283582	0.0758196	0.2591769
CFGAN-u	0.104793	0.1754949	0.1583063	0.0992318	0.1482022
CFGAN-i	0.0001906	0.0004379	0.0003876	0.0001177	0.0616414
GANMF-u	0.0946351	0.1418546	0.1381279	0.0921853	**0.2819294**
GANMF-i	0.0958061	0.1456018	0.1337833	0.0889318	0.2610982
Relative Change	-34.91%	-41.84%	-42.84%	-46.73%	8.78%

Table 5.19: Results for cutoff 20 on Delicious. Higher values are better. Best results are in bold. Relative change reports the difference between best GANMF variant and best performing baseline.

Algorithm	Prec@50	Rec@50	nDCG@50	mAP@50	Cov@50
Top popular	0.0015033	0.0063785	0.0040758	0.0007879	0.0009101
PureSVD	0.0492702	0.1914322	0.1181371	0.0536321	0.3080046
WRMF	**0.0728105**	**0.308557**	**0.2677807**	**0.1853322**	0.3798737
MF-BPR	0.0074074	0.04619	0.0270562	0.0097744	0.0438583
SLIM-BPR	0.0289434	0.1308242	0.1364398	0.0777533	0.340826
CFGAN-u	0.0561002	0.2370797	0.1856272	0.1113201	0.2359331
CFGAN-i	0.0001743	0.0007427	0.000538	0.0001243	0.1159441
GANMF-u	0.055915	0.2201512	0.1728492	0.1032774	**0.4228797**
GANMF-i	0.0566231	0.2326523	0.1707571	0.0998884	0.3675946
Relative Change	-22.23%	-24.60%	-35.45%	-44.27%	11.32%

Table 5.20: Results for cutoff 50 on Delicious. Higher values are better. Best results are in bold. Relative change reports the difference between best GANMF variant and best performing baseline.

recall from chapter 3 that two defining components of GANMF are the autoencoder acting as the discriminator and incorporation of feature matching loss for the generator with the aim of helping class conditioning. We perform an ablation study on these two components which we detail in the next two sections. Finally on section 5.3.3 we compare the standard GANMF model with a version of it where we use DNN for the discriminator and generator.

5.3.1 GANMF with binary classifier discriminator

For this experiment we drop the autoencoder discriminator and replace it with the binary classifier network used in vanilla GAN and in the CFGAN model. We call this model biGANMF. This new discriminator takes as input a user/item profile and outputs a single value in the range $[0-1]$ denoting the probability that the seen input is coming from the real training data. We perform the experiment with unchanged generator architecture coupled with feature matching loss. As user/item features we use the learned features in the last layer of the discriminator before the binary output (see figure 5.2). BiGANMF is optimized just like the standard GANMF model; bayesian optimization is used to determine the number of layers, the number of units per layer and the activation function of each layer of the new discriminator. The optimization is applied to the whole model and not only the discriminator. The discriminator loss function is now defined as in the vanilla GAN:

$$\max_{D} \mathbb{E}_{\mathbf{x}\sim p_{data}}[log\, D(\mathbf{x})] + \mathbb{E}_{\mathbf{z}\sim p_{\mathbf{z}}}\left[log\big(1 - D(G(\mathbf{z}))\big)\right]$$

MovieLens 100K

We present the results of biGANMF versus the standard GANMF model on MovieLens 100K in tables 5.21, 5.22, 5.23 and 5.24. As we can see, biGANMF performs worse than the autoencoder GANMF by a significant margin in all metrics and all cutoffs, at around 40% worse. After 50 bayesian optimization evaluations the best performing biGANMF-u was the model whose discriminator had 1 layer with 1024 units and a linear activation function. An interesting result is the relation between the coverage and the performance of biGANMF variants. In the standard GANMF the variant with the best performance is the one with the lowest coverage, albeit a small difference. The reverse is seen for biGANMF where biGANMF-u has approximately 4 times the coverage of biGANMF-i yet still performing better. Thus biGANMF-u

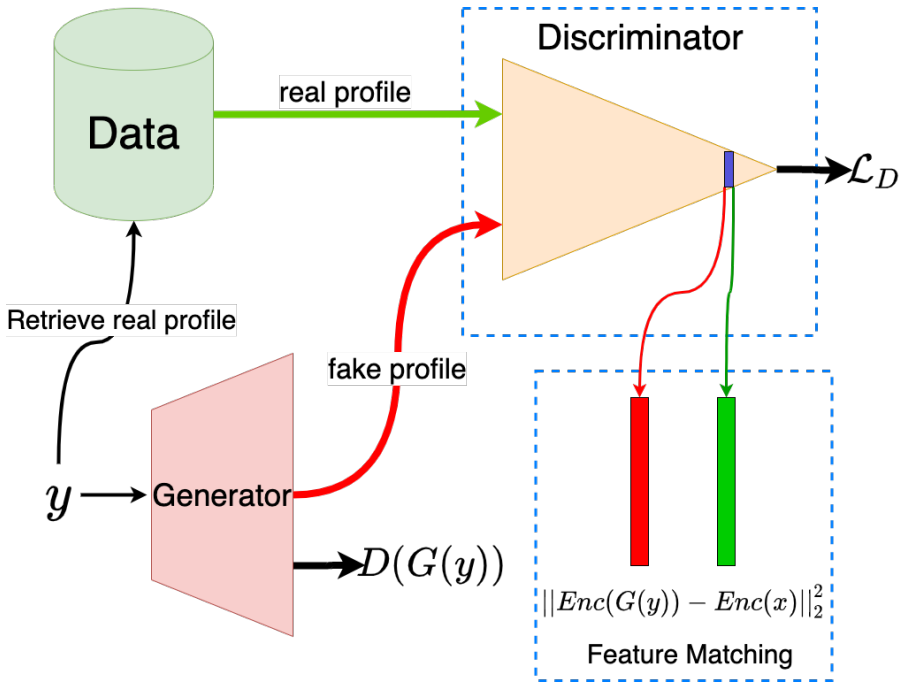


Figure 5.2: GANMF with binary classifier discriminator. The feature matching loss is optimized with features coming from the last fully connected layer of the discriminator before the final output.

is able to recommend a larger pool of items and still recommend those items to the right users compared to biGANMF-i.

Algorithm	Prec@5	Rec@5	nDCG@5	mAP@5	Cov@5
GANMF-u	0.3959703	0.1360042	0.1928307	0.3210631	**0.146849**
GANMF-i	**0.4197243**	**0.1421869**	**0.2068968**	**0.3481637**	0.137931
biGANMF-u	0.2434783	0.0782224	0.1124793	0.1722632	0.0802616
biGANMF-i	0.1978791	0.0494344	0.0800853	0.1410166	0.020214
Relative Change	-41.99%	-44.99%	-45.64%	-50.52%	-45.34%

Table 5.21: Results for cutoff 5 on MovieLens 100K. Higher values are better. Best results are in bold. Relative change reports the difference between best biGANMF variant and best performing GANMF variant.

Algorithm	Prec@10	Rec@10	nDCG@10	mAP@10	Cov@10
GANMF-u	0.3371156	0.218114	0.255901	0.2628942	**0.1967895**
GANMF-i	**0.3565217**	**0.2282473**	**0.2729096**	**0.288402**	0.1920333
biGANMF-u	0.2067869	0.1273371	0.1508587	0.1323419	0.1224732
biGANMF-i	0.1615058	0.0792526	0.1048993	0.101738	0.0356718
Relative Change	-42.00%	-44.21%	-44.72%	-54.11%	-37.76%

Table 5.22: Results for cutoff 10 on MovieLens 100K. Higher values are better. Best results are in bold. Relative change reports the difference between best biGANMF variant and best performing GANMF variant.

Algorithm	Prec@20	Rec@20	nDCG@20	mAP@20	Cov@20
GANMF-u	0.2734358	0.3321748	0.3272267	0.2303143	**0.2758621**
GANMF-i	**0.2827678**	**0.3406212**	**0.3432445**	**0.2487557**	0.2526754
biGANMF-u	0.1685578	0.1996495	0.1950765	0.1105033	0.1884661
biGANMF-i	0.1343584	0.1276293	0.1374063	0.08097	0.0612366
Relative Change	-40.39%	-41.39%	-43.17%	-55.58%	-31.68%

Table 5.23: Results for cutoff 20 on MovieLens 100K. Higher values are better. Best results are in bold. Relative change reports the difference between best biGANMF variant and best performing GANMF variant.

MovieLens 1M

We give in tables 5.25, 5.26, 5.27 and 5.28 the results of biGANMF on MovieLens 1M. We see a similar pattern to Movielens 100K with the GANMF performing again substantially better. The best results are split between the variants of GANMF since both of them are performing quite similar with

Algorithm	Prec@50	Rec@50	nDCG@50	mAP@50	Cov@50
GANMF-u	0.1834571	0.5169792	0.4205572	0.2200193	**0.4173603**
GANMF-i	**0.1931495**	**0.5375202**	**0.4434139**	**0.2400581**	0.3989298
biGANMF-u	0.1180276	0.3185704	0.2564102	0.1033315	0.3258026
biGANMF-i	0.1001697	0.2393149	0.194118	0.0728119	0.1254459
Relative Change	-38.89%	-40.73%	-42.17%	-56.96%	-21.94%

Table 5.24: Results for cutoff 50 on MovieLens 100K. Higher values are better. Best results are in bold. Relative change reports the difference between best biGANMF variant and best performing GANMF variant.

negligible differences albeit GANMF-u's higher item coverage. biGANMF performs on average 35% than GANMF on all metrics and cutoffs. The resulting best biGANMF model is the one with 1 hidden layer of 581 units and a ReLU hidden activation function. We note here the tendency of biGANMF model to work better with shallow discriminator just like in MovieLens 100K. Different from the previous dataset, the disparity in item coverage between the biGANMF variants is bigger at a factor of approximately 5.

Algorithm	Prec@5	Rec@5	nDCG@5	mAP@5	Cov@5
GANMF-u	0.4320199	**0.1083758**	**0.1692069**	0.3539966	**0.2026444**
GANMF-i	**0.4323179**	0.1053124	0.1690019	**0.358333**	0.1392337
biGANMF-u	0.1977815	0.0351879	0.0614358	0.1434128	0.0132218
biGANMF-i	0.3024503	0.061469	0.1051042	0.2323213	0.1168376
Relative Change	-30.04%	-43.28%	-37.88%	-35.17%	-42.34%

Table 5.25: Results for cutoff 5 on MovieLens 1M. Higher values are better. Best results are in bold. Relative change reports the difference between best biGANMF variant and best performing GANMF variant.

Algorithm	Prec@10	Rec@10	nDCG@10	mAP@10	Cov@10
GANMF-u	0.3702483	**0.1742579**	**0.2252712**	0.28728	**0.264436**
GANMF-i	**0.372351**	0.1692659	0.2243996	**0.2916971**	0.1826767
biGANMF-u	0.1741722	0.0598876	0.0843955	0.11023	0.0205073
biGANMF-i	0.2624834	0.1029214	0.1420329	0.1815342	0.1621695
Relative Change	-29.51%	-40.94%	-36.95%	-37.77%	-38.67%

Table 5.26: Results for cutoff 10 on MovieLens 1M. Higher values are better. Best results are in bold. Relative change reports the difference between best biGANMF variant and best performing GANMF variant.

Algorithm	Prec@20	Rec@20	nDCG@20	mAP@20	Cov@20
GANMF-u	**0.3053808**	**0.2653345**	**0.2885205**	0.2417735	**0.3416082**
GANMF-i	0.304404	0.2552169	0.2852354	**0.2422803**	0.2417701
biGANMF-u	0.1474172	0.0998068	0.1131748	0.0857512	0.0364274
biGANMF-i	0.2203725	0.1665832	0.1871483	0.1458388	0.2309768
Relative Change	-27.84%	-37.22%	-35.14%	-39.81%	-32.39%

Table 5.27: Results for cutoff 20 on MovieLens 1M. Higher values are better. Best results are in bold. Relative change reports the difference between best biGANMF variant and best performing GANMF variant.

Algorithm	Prec@50	Rec@50	nDCG@50	mAP@50	Cov@50
GANMF-u	**0.2184503**	**0.4275134**	**0.3809177**	**0.2164192**	**0.4695089**
GANMF-i	0.2166623	0.4125733	0.3749247	0.2142928	0.3537507
biGANMF-u	0.1083609	0.1741139	0.1567217	0.0681944	0.0769023
biGANMF-i	0.1616821	0.2857345	0.2557555	0.1242929	0.3602267
Relative Change	-25.99%	-33.16%	-32.86%	-42.57%	-23.28%

Table 5.28: Results for cutoff 50 on MovieLens 1M. Higher values are better. Best results are in bold. Relative change reports the difference between best biGANMF variant and best performing GANMF variant.

CiaoDVD

Tables 5.29, 5.30, 5.31 and 5.32 summarize the results of biGANMF on CiaoDVD dataset. In this dataset the performance of both variants of biGANMF falls behind a lot compared to standard GANMF. We observe at least 40% worse performance compared to the best GANMF variant in all metrics. A peculiar second observation and a possible explanation for the performance of biGANMF is the very low coverage. Coverage@5 is only 0.1% of the total item catalogue. Our explanation for this is a possible mode collapse of the generator of both variants to recommending only most popular items. This is supported also by the coverage of the non-personalized top-popular approach which is very close to the coverage of biGANMF. We note here that CiaoDVD is the dataset with the lowest density and a Gini index of 0.65. The best biGANMF-i's discriminator had 5 hidden layers with linear activation function and 1024 units each.

LastFM

We summarize the results of biGANMF on LastFM dataset on the tables 5.33, 5.34, 5.35 and 5.36. Also on LastFM, biGANMF has much lower performance than standard GANMF. We also include the top-popular rec-

Algorithm	Prec@5	Rec@5	nDCG@5	mAP@5	Cov@5
Top popular	0.0093078	0.0301836	0.0218311	0.0173683	0.0008684
GANMF-u	**0.01387**	**0.0420354**	**0.0298209**	**0.0230707**	0.0535947
GANMF-i	0.0128736	0.0356792	0.0267557	0.0214877	**0.1059488**
biGANMF-u	0.0064761	0.0213594	0.0170405	0.0141438	0.0009925
biGANMF-i	0.001075	0.0033913	0.0019349	0.0013056	0.0008064
Relative Change	-53.31%	-49.19%	-42.86%	-38.69%	-99.06%

Table 5.29: Results for cutoff 5 on CiaoDVD. Higher values are better. Best results are in bold. Relative change reports the difference between best biGANMF variant and best performing GANMF variant.

Algorithm	Prec@10	Rec@10	nDCG@10	mAP@10	Cov@10
Top popular	0.0076691	0.0492416	0.0284671	0.0196393	0.0015508
GANMF-u	**0.011366**	**0.0648433**	**0.0381765**	**0.0258148**	0.1134545
GANMF-i	0.0100026	0.0533817	0.0333071	0.0234155	**0.1547671**
biGANMF-u	0.005021	0.0327005	0.0210368	0.0154138	0.0017989
biGANMF-i	0.0006948	0.0037171	0.0020925	0.0013258	0.0016748
Relative Change	-55.82%	-49.57%	-44.90%	-40.29%	-98.84%

Table 5.30: Results for cutoff 10 on CiaoDVD. Higher values are better. Best results are in bold. Relative change reports the difference between best biGANMF variant and best performing GANMF variant.

Algorithm	Prec@20	Rec@20	nDCG@20	mAP@20	Cov@20
Top popular	0.0060304	0.0748749	0.035647	0.0213848	0.0029155
GANMF-u	**0.0085212**	**0.0945504**	**0.046718**	**0.0279302**	**0.2396253**
GANMF-i	0.0076036	0.080545	0.0410068	0.0251971	0.2177284
biGANMF-u	0.0040115	0.0508702	0.0260647	0.0165922	0.0035978
biGANMF-i	0.0005768	0.0061517	0.002785	0.0014816	0.0031015
Relative Change	-52.92%	-46.20%	-44.21%	-40.59%	-98.50%

Table 5.31: Results for cutoff 20 on CiaoDVD. Higher values are better. Best results are in bold. Relative change reports the difference between best biGANMF variant and best performing GANMF variant.

Algorithm	Prec@50	Rec@50	nDCG@50	mAP@50	Cov@50
Top popular	0.0044048	0.1385646	0.0494453	0.0235451	0.005955
GANMF-u	**0.0055926**	**0.1492362**	**0.0592024**	**0.0299301**	**0.4378761**
GANMF-i	0.0050236	0.1251562	0.0513823	0.0267774	0.3358973
biGANMF-u	0.0025485	0.0770503	0.0318737	0.017419	0.008064
biGANMF-i	0.0005794	0.0131116	0.004367	0.0016863	0.0077539
Relative Change	-54.43%	-48.37%	-46.16%	-41.80%	-98.16%

Table 5.32: Results for cutoff 50 on CiaoDVD. Higher values are better. Best results are in bold. Relative change reports the difference between best biGANMF variant and best performing GANMF variant.

ommender in the above tables. We can see that both biGANMF-i and biGANMF-u fall short on the performance compared to top-popular recommender, however they have higher coverage. Between best GANMF and best biGANMF we can see a factor of 2-3 better performance for GANMF. The best biGANMF-u model used a discriminator with 1 hidden 4-unit layer with linear activation.

Algorithm	Prec@5	Rec@5	nDCG@5	mAP@5	Cov@5
Top popular	0.0777125	0.0400469	0.0583715	0.0527276	0.0010209
GANMF-u	**0.2607219**	**0.1338915**	**0.1914327**	**0.2009306**	0.038623
GANMF-i	0.2562633	0.1318541	0.1867868	0.1935747	**0.044578**
biGANMF-u	0.0720807	0.0367051	0.0488659	0.040436	0.0012477
biGANMF-i	0.0820594	0.0418584	0.0563792	0.0494882	0.0070327
Relative Change	-68.53%	-68.74%	-70.55%	-75.37%	-84.22%

Table 5.33: Results for cutoff 5 on LastFM. Higher values are better. Best results are in bold. Relative change reports the difference between best biGANMF variant and best performing GANMF variant.

Algorithm	Prec@10	Rec@10	nDCG@10	mAP@10	Cov@10
Top popular	0.0662951	0.0680608	0.0777387	0.0382397	0.001588
GANMF-u	**0.1937898**	0.1968786	**0.2356293**	**0.1391564**	0.0565449
GANMF-i	0.1937367	**0.1973378**	0.2326003	0.1357974	**0.0623866**
biGANMF-u	0.0619427	0.0631484	0.0673429	0.0296739	0.001985
biGANMF-i	0.0620488	0.0636028	0.0714914	0.0340902	0.0119102
Relative Change	-67.98%	-67.77%	-69.66%	-75.50%	-80.91%

Table 5.34: Results for cutoff 10 on LastFM. Higher values are better. Best results are in bold. Relative change reports the difference between best biGANMF variant and best performing GANMF variant.

Algorithm	Prec@20	Rec@20	nDCG@20	mAP@20	Cov@20
Top popular	0.0488057	0.0993311	0.0946814	0.0391824	0.0024387
GANMF-u	**0.1360934**	0.2754476	**0.278796**	**0.1437082**	0.085583
GANMF-i	0.1360138	**0.2757899**	0.2755811	0.1405968	**0.0896098**
biGANMF-u	0.0474257	0.0959869	0.0852117	0.0313648	0.0036865
biGANMF-i	0.0460456	0.0936367	0.087913	0.0352168	0.0191697
Relative Change	-65.15%	-65.20%	-68.47%	-75.49%	-78.61%

Table 5.35: Results for cutoff 20 on LastFM. Higher values are better. Best results are in bold. Relative change reports the difference between best biGANMF variant and best performing GANMF variant.

Algorithm	Prec@50	Rec@50	nDCG@50	mAP@50	Cov@50
Top popular	0.0341083	0.1725694	0.1258099	0.0447256	0.0047074
GANMF-u	0.0795117	0.4014345	**0.3330374**	**0.1612095**	0.1620349
GANMF-i	**0.0796178**	**0.4019249**	0.3300618	0.1581174	0.1544918
biGANMF-u	0.0333439	0.1683531	0.1161281	0.0370809	0.0075431
biGANMF-i	0.031518	0.16029	0.1163866	0.0406224	0.0374319
Relative Change	-58.12%	-58.11%	-65.05%	-74.80%	-76.90%

Table 5.36: Results for cutoff 50 on LastFM. Higher values are better. Best results are in bold. Relative change reports the difference between best biGANMF variant and best performing GANMF variant.

Delicious

We see a repeating pattern on the performance of biGANMF also on Delicious dataset. biGANMF's performance is several orders of magnitude worse than GANMF and almost 0 across all metrics. The coverage is also very low and the fact that this dataset has a Gini index of 0.25, thus a more moderate long tail distribution, makes it difficult for biGANMF with a low coverage to perform well because there is not a steep bias toward popular items. Both biGANMF models perform worse even than top-popular recommendations.

Algorithm	**Prec@5**	**Rec@5**	**nDCG@5**	**mAP@5**	**Cov@5**
Top popular	0.0021786	0.0009668	0.0012767	0.0010004	0.00013
GANMF-u	**0.1383442**	**0.0549645**	**0.0777383**	**0.1189397**	**0.1041995**
GANMF-i	0.120915	0.0501615	0.0684077	0.1032746	0.0914436
biGANMF-u	0.0006536	0.0002183	0.0003706	0.0003758	0.0001878
biGANMF-i	0.0007625	0.000345	0.0004438	0.0005792	0.0002311
Relative Change	-99.45%	-99.37%	-99.43%	-99.51%	-99.78%

Table 5.37: Results for cutoff 5 on Delicious. Higher values are better. Best results are in bold. Relative change reports the difference between best biGANMF variant and best performing GANMF variant.

Algorithm	**Prec@10**	**Rec@10**	**nDCG@10**	**mAP@10**	**Cov@10**
Top popular	0.0020153	0.002054	0.0018988	0.0007785	0.0001878
GANMF-u	**0.1251634**	**0.0952886**	**0.1096041**	**0.1021225**	**0.180807**
GANMF-i	0.1150327	0.0901394	0.0996606	0.0922297	0.1641073
biGANMF-u	0.0004902	0.0003036	0.0004467	0.0002095	0.0003612
biGANMF-i	0.0003813	0.000345	0.0004438	0.0002896	0.0004334
Relative Change	-99.61%	-99.64%	-99.59%	-99.72%	-99.76%

Table 5.38: Results for cutoff 10 on Delicious. Higher values are better. Best results are in bold. Relative change reports the difference between best biGANMF variant and best performing GANMF variant.

In this experiment we compared GANMF with other strong and well-known CF-based baselines. GANMF performed on par with and even exceeded the baselines on selected datasets. This shows that we can use GANs to learn better latent factors for users and items while still using a linear MF approach. Morever, GANMF scored better than CFGAN on all datasets except Delicious.

Algorithm	Prec@20	Rec@20	nDCG@20	mAP@20	Cov@20
Top popular	0.001988	0.0034133	0.0027562	0.0006951	0.0004189
GANMF-u	0.0946351	0.1418546	**0.1381279**	**0.0921853**	**0.2819294**
GANMF-i	**0.0958061**	**0.1456018**	0.1337833	0.0889318	0.2610982
biGANMF-u	0.0002996	0.0003944	0.0004992	0.0001392	0.0006645
biGANMF-i	0.0003813	0.0005645	0.0005872	0.0002735	0.0007512
Relative Change	-99.60%	-99.61%	-99.57%	-99.70%	-99.73%

Table 5.39: Results for cutoff 20 on Delicious. Higher values are better. Best results are in bold. Relative change reports the difference between best biGANMF variant and best performing GANMF variant.

Algorithm	Prec@50	Rec@50	nDCG@50	mAP@50	Cov@50
Top popular	0.0015033	0.0063785	0.0040758	0.0007879	0.0009101
GANMF-u	0.055915	0.2201512	**0.1728492**	**0.1032774**	**0.4228797**
GANMF-i	**0.0566231**	**0.2326523**	0.1707571	0.0998884	0.3675946
biGANMF-u	0.0002505	0.0010525	0.0007765	0.0001594	0.0016035
biGANMF-i	0.0003813	0.0013978	0.0009869	0.0003445	0.0017624
Relative Change	-99.33%	-99.40%	-99.43%	-99.67%	-99.58%

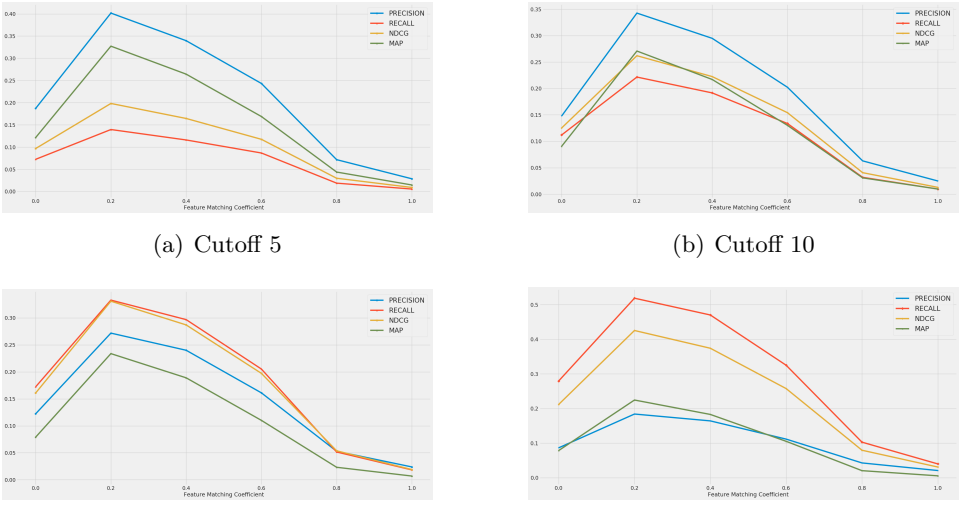
Table 5.40: Results for cutoff 50 on Delicious. Higher values are better. Best results are in bold. Relative change reports the difference between best biGANMF variant and best performing GANMF variant.

5.3.2 Effect of feature matching loss

In this section we investigate how much effect does the added feature matching loss have on the performance and eventually on the conditioning of the generator. The way we perform this experiment is by using the standard GANMF and fixing the hyperparameter values to the ones found during the optimization phase (see section 5.1). The only hyperparameter we modify is the feature matching coefficient in equation 3.4; we train again both GANMF variants with feature matching coefficient ranging from $[0-1]$ with a step of 0.2. A value of 0 means that we are turning off feature matching loss whereas a value of 1 means we are only using feature matching loss and disregarding gradients coming from the discriminator. Finally we visualize the similarity between the generated profiles of every user in order to understand how much feature matching helps in reducing this similarity.

MovieLens 100K

We give in figures 5.3 and 5.4 the plots of the metrics at every cutoff of GANMF trained with different values for the feature matching coefficient. We see clearly that a value 0.2 provides the best recommendation accuracy. An important insight is the sharp decrease of the performance of GANMF for the value 1, which is the case when we neglect completely the discriminator.



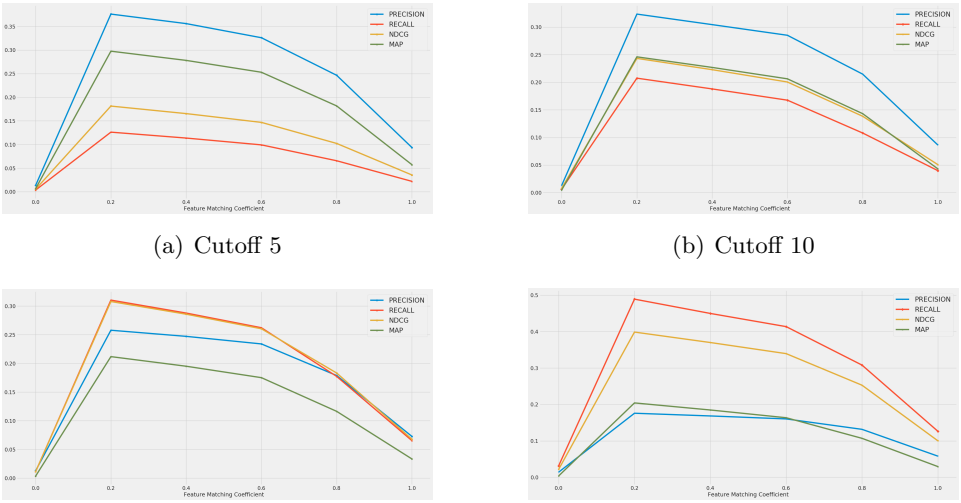
(a) Cutoff 5

(b) Cutoff 10

(c) Cutoff 20

(d) Cutoff 50

Figure 5.3: Effect of feature matching loss on GANMF-u performance on MovieLens 100K.



(a) Cutoff 5

(b) Cutoff 10

(c) Cutoff 20

(d) Cutoff 50

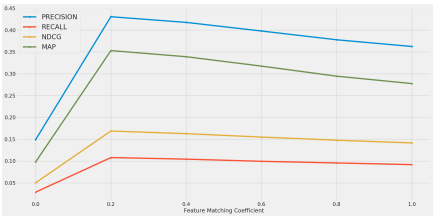
Figure 5.4: Effect of feature matching loss on GANMF-i performance on MovieLens 100K.

MovieLens 1M

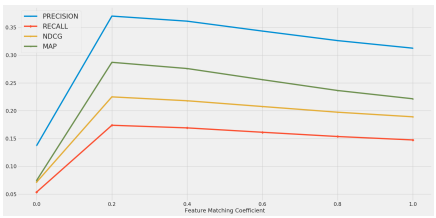
We see again a similar situation for the dataset MovieLens 1M, figures 5.5 and 5.6. For the item-based variant of GANMF depending only on the gradient coming from the discriminator makes it difficult to learn to generate plausible item profiles. We recall here the shape of the full URM of MovieLens 10, 6040 users $\times$ 3706 items. Without the help from feature matching the generator needs to rely on only 3706 item profiles to learn to generate 6040 dimensional "fake" profiles. This is a limited number of profiles for the discriminator to give the right amount of information to the generator when considering that, as mentioned in 3.2.2, each generated profile should be unique to each user.

CiaoDVD

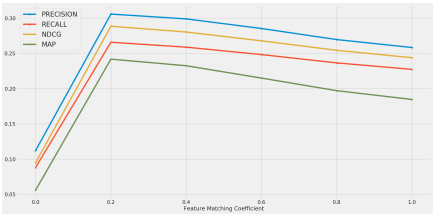
On CiaoDVD dataset, figures 5.7, 5.8, we see an interesting pattern on the effect of feature matching. First, not using feature matching at all deteriorates the performance of both variants of GANMF. Secondly, for greater cutoffs the effect of this additional loss on most of the metrics seems to be very low. From section 5.2 we saw that GANMF does not score very high on CiaoDVD and we believe the problem of the performance and also the behavior of feature matching is the high sparsity of CiaoDVD. Nonetheless, we still can observe that a combination of both discriminator and feature



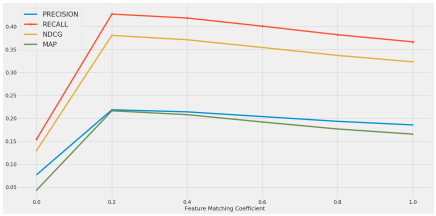
(a) Cutoff 5



(b) Cutoff 10

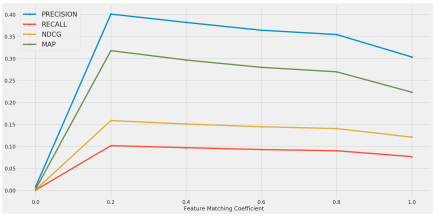


(c) Cutoff 20

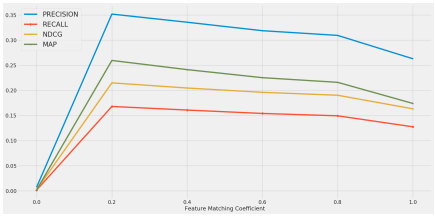


(d) Cutoff 50

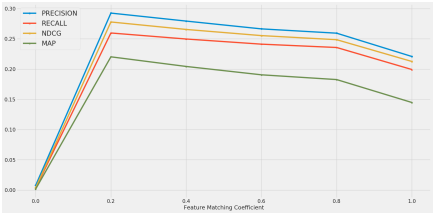
Figure 5.5: Effect of feature matching loss on GANMF-u performance on MovieLens 1M.



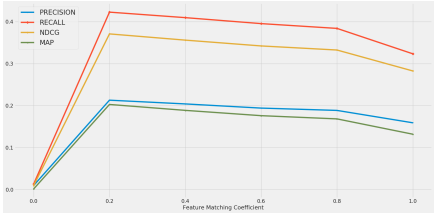
(a) Cutoff 5



(b) Cutoff 10



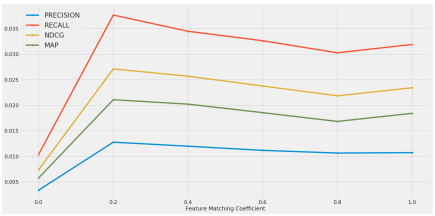
(c) Cutoff 20



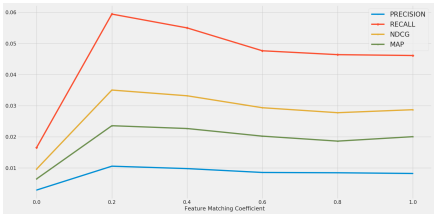
(d) Cutoff 50

Figure 5.6: Effect of feature matching loss on GANMF-i performance on MovieLens 1M.

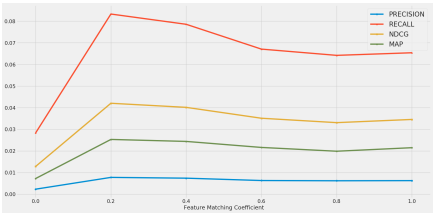
matching loss provides the best accuracy.



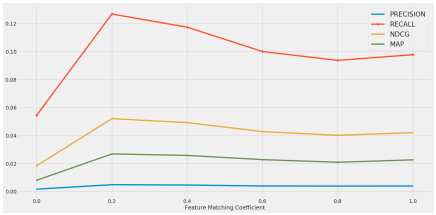
(a) Cutoff 5



(b) Cutoff 10

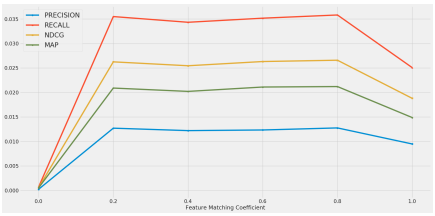


(c) Cutoff 20

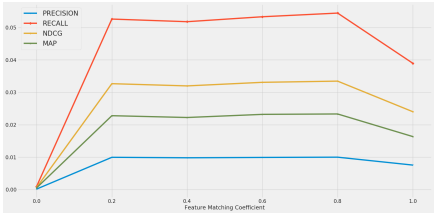


(d) Cutoff 50

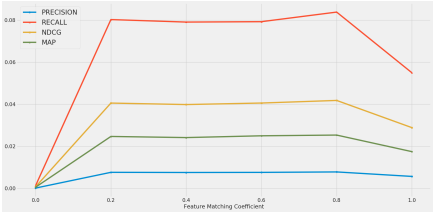
Figure 5.7: Effect of feature matching loss on GANMF-u performance on CiaoDVD.



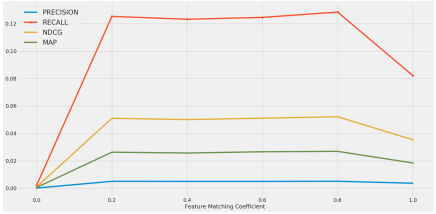
(a) Cutoff 5



(b) Cutoff 10



(c) Cutoff 20



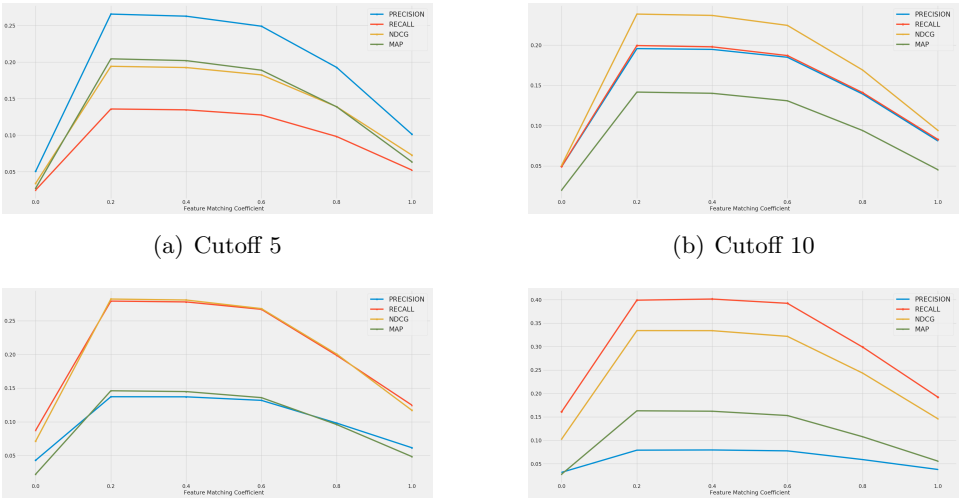
(d) Cutoff 50

Figure 5.8: Effect of feature matching loss on GANMF-i performance on CiaoDVD.

LastFM

In figures 5.9 and 5.10 we can see that the behavior of feature matching on LastFM supports our claims on feature matching for MovieLens 1M. The

full URM of LastFM has dimensions 1900 users × 17632 items. GANMF-u's discriminator is trained with 1900 real profiles where each profile is approximately 10 times as long as the available profiles. Adding featuring matching drastically helps recommendations. However considering the reverse learning problem that GANMF-i is used for. Its discriminator is trained on 17632 real profiles with each profile 1900 users long. Even in this case feature matching helps but after a certain point it does not affect anymore learning.



(a) Cutoff 5

(b) Cutoff 10

(c) Cutoff 20

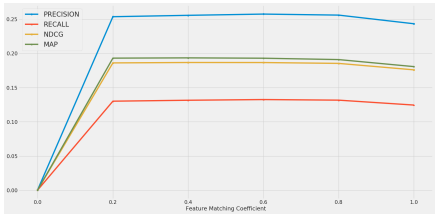
(d) Cutoff 50

Figure 5.9: Effect of feature matching loss on GANMF-u performance on LastFM.

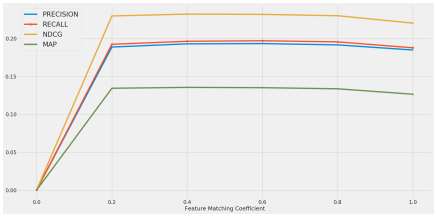
Delicious

Delicious dataset, in figures 5.11 and 5.12, shows us in some way the same underlying pattern of the combination of feature matching with the GAN adversarial loss seen in LastFM. Delicious dataset has a shape of 1892 users × 69223 items, a difference by a factor of approximately 36. On GANMF-u, adding feature matching helps the recommendation accuracy to the point that, for all cutoffs, a better performance is achieved with only feature matching. However the effect of increasing the weight of feature matching beyond the value 0.2 is small compared to going from without feature matching to feature matching weighted by 0.2. On the other hand for GANMF-i a combination between both losses brings the best performance especially in the shorter recommendation list lengths.

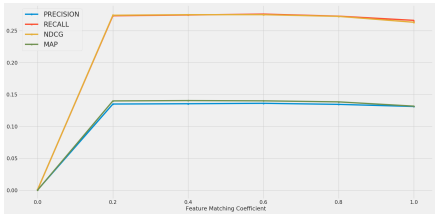
We have shown empirically that joining the GAN adversarial loss func-



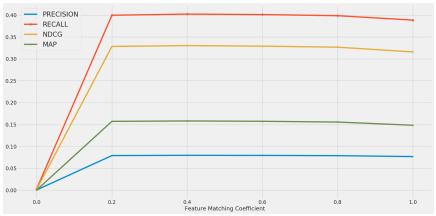
(a) Cutoff 5



(b) Cutoff 10

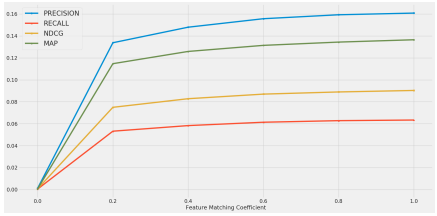


(c) Cutoff 20

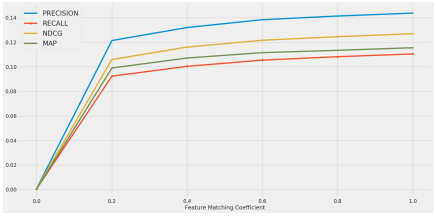


(d) Cutoff 50

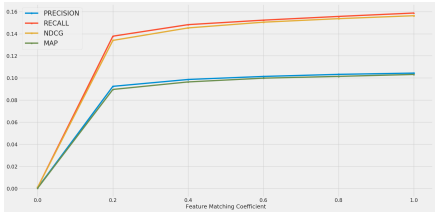
Figure 5.10: Effect of feature matching loss on GANMF-i performance on LastFM.



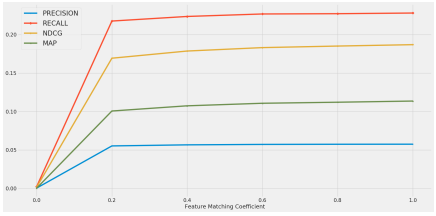
(a) Cutoff 5



(b) Cutoff 10

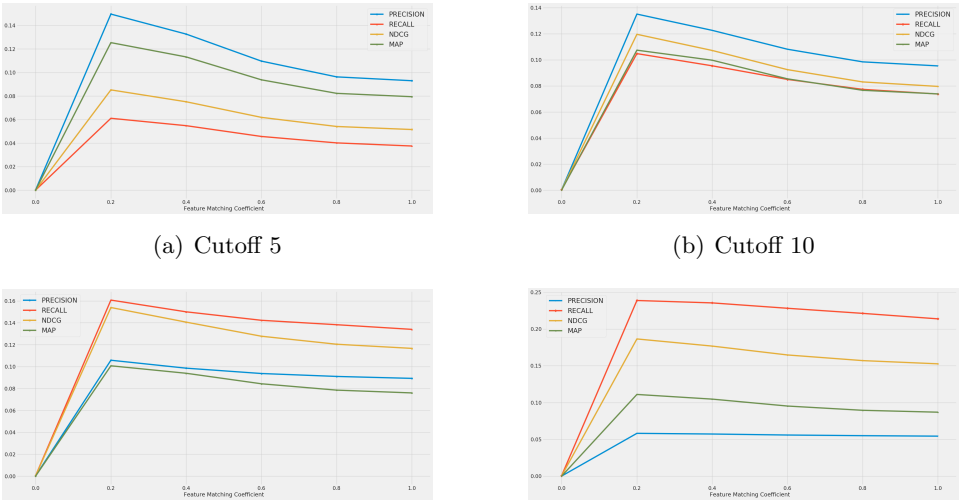


(c) Cutoff 20



(d) Cutoff 50

Figure 5.11: Effect of feature matching loss on GANMF-u performance on Delicious.



(a) Cutoff 5

(b) Cutoff 10

(c) Cutoff 20

(d) Cutoff 50

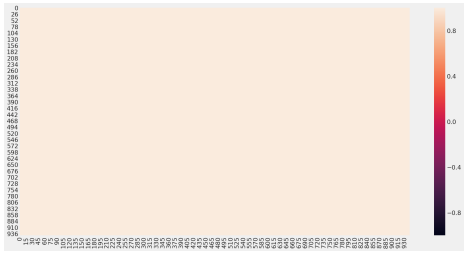
Figure 5.12: Effect of feature matching loss on GANMF-i performance on Delicious.

tion with feature matching loss for the generator helps the generator learn to produce more plausible user/item profiles and eventually increases the recommendation accuracy. Finally we are also interested in how successful is feature matching in conditioning the generator to produce user/item-tailored, "fake" profiles. In order to investigate this, we first optimized the standard GANMF-u without feature matching. Then generated user profiles and computed the cosine similarity between each pair of users. We give in figure 5.13 the heatmaps of the similarity for each dataset and the respective mean and standard deviation of the similarities.

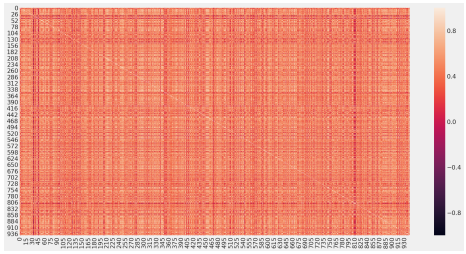
We can clearly see from the heatmaps that for all the datasets we are able to reduce the average similarity between the generated profiles of any two users when we incorporate feature matching loss for the generator. Moreover, not including feature matching loss on some of the datasets, we are presented with an average similarity of almost 1, meaning that the generator is producing the same "fake" profile for all users.

5.3.3 GANMF with DNN components

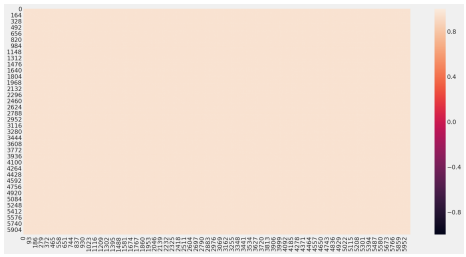
In the previous sections of this chapter we have shown that our standard GANMF model, with a linear MF operation performed by the generator, can outperform traditional RS approaches on some datasets and CFGAN on almost all datasets we tested them on. As a final experiment we want to understand whether extending the standard GANMF model to use DNN



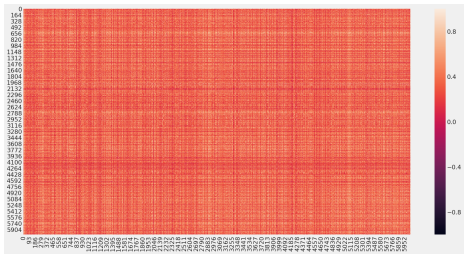
(a) MovieLens 100K w/o feature matching. Mean: 0.9995351, Std: 10^{-3}.



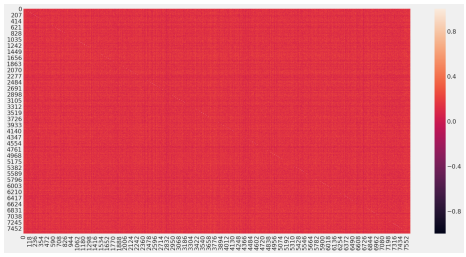
(b) MovieLens 100Kw/ feature matching. Mean: 0.48735228, Std: 0.2251461



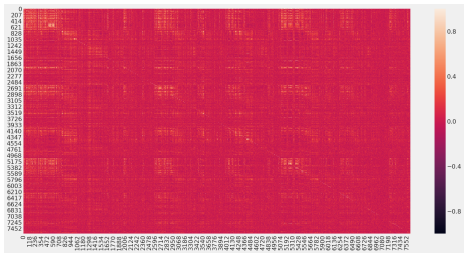
(c) MovieLens 1M w/o feature matching. Mean: 0.9563655, Std: 0.0037879287.



(d) MovieLens 1M w/ feature matching. Mean: 0.37644428, Std: 0.19592425

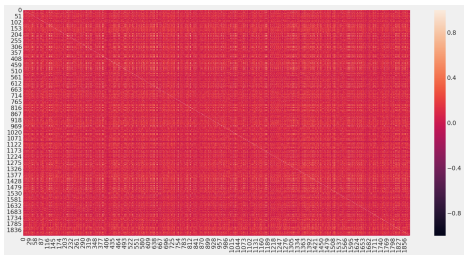


(e) CiaoDVD w/o feature matching. Mean: 0.14036298, Std: 0.07149107.

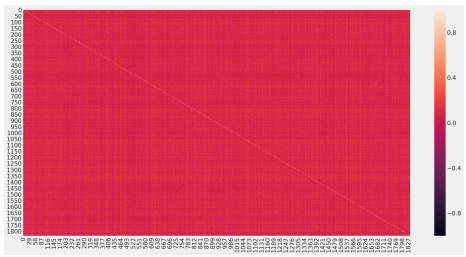


(f) CiaoDVD w/ feature matching. Mean: 0.04443381, Std: 0.1101416

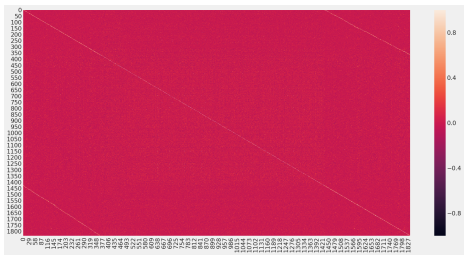
(g) LastFM w/o feature matching. Mean: 0.99140006, Std: 0.00073062413.



(h) LastFM w/ feature matching. Mean: 0.12330822, Std: 0.17331722



(i) Delicious w/o feature matching. Mean: 0.0840853, Std: 0.07278927.



(j) Delicious w/ feature matching. Mean: 0.0017516246, Std: 0.07453684

Figure 5.13: Feature matching conditioning on the user generated profiles.

components for the discriminator and generator, can provide further performance increments compared to the standard version. To conduct this experiment we substitute the single-layer-linearly-activated autoencoder with a deeper autoencoder. We also substitute the generator with an MLP. In this new version, which we call DeepGANMF, the generator does not use anymore embedding layers so we are constrained to use another type of conditioning vector for the generator. We take here the approach of CFGAN and utilize the complete real user profiles as conditions for the generator. With this two new components, we optimize again each variant (user and item based) of DeepGANMF for each of the datasets. In the following sections we give the comparison of DeepGANMF with the standard GANMF on all datasets.

MovieLens 100K

We give in tables 5.41, 5.42, 5.43 and 5.44 the results for DeepGANMF. We can observe that both standard GANMF variants perform much better than DeepGANMF on all metrics and on all cutoffs. A very interesting result though is the coverage of DeepGANMF-u. On cutoffs 5 it is 56% of the total items, the highest seen by any model in this thesis at this cutoff. At cutoff 50, the coverage reaches 99%. almost the complete catalogue of items. Also the other variant shows higher coverage than GANMF on all cutoffs. Also the performance of DeepGANMF-i is not that much lower when compared with GANMF, considering the higher coverage. At cutoff 50, coverage for DeepGANMF-i is 54% whereas GANMF-i covers only 40% of the items.

Algorithm	**Prec@5**	**Rec@5**	**nDCG@5**	**mAP@5**	**Cov@5**
GANMF-u	0.3959703	0.1360042	0.1928307	0.3210631	0.146849
GANMF-i	**0.4197243**	**0.1421869**	**0.2068968**	**0.3481637**	0.137931
DeepGANMF-u	0.1005302	0.0214772	0.0352384	0.0659971	**0.559453**
DeepGANMF-i	0.3230117	0.0997668	0.1495498	0.2539148	0.1872771

Table 5.41: Results for cutoff 5 on MovieLens 100K. Higher values are better. Best results are in bold.

MovieLens 1M

We give the results of DeepGANMF on MovieLens 1M in tables 5.41, 5.42, 5.43 and 5.44. Also in this version of MovieLens, GANMF variants perform much better than DeepGANMF in all metrics and cutoffs. The same observation can be made for DeepGANMF-u, still achieving great coverage at

Algorithm	Prec@10	Rec@10	nDCG@10	mAP@10	Cov@10
GANMF-u	0.3371156	0.218114	0.255901	0.2628942	0.1967895
GANMF-i	**0.3565217**	**0.2282473**	**0.2729096**	**0.288402**	0.1920333
DeepGANMF-u	0.0846235	0.0359723	0.0476395	0.0472484	**0.7419738**
DeepGANMF-i	0.2836691	0.167003	0.2025917	0.2088889	0.2431629

Table 5.42: Results for cutoff 10 on MovieLens 100K. Higher values are better. Best results are in bold.

Algorithm	Prec@20	Rec@20	nDCG@20	mAP@20	Cov@20
GANMF-u	0.2734358	0.3321748	0.3272267	0.2303143	**0.2758621**
GANMF-i	**0.2827678**	**0.3406212**	**0.3432445**	**0.2487557**	0.2526754
DeepGANMF-u	0.0721633	0.0633654	0.0648731	0.0355798	0.89239
DeepGANMF-i	0.2326087	0.2587212	0.2614043	0.1798248	0.3418549

Table 5.43: Results for cutoff 20 on MovieLens 100K. Higher values are better. Best results are in bold.

Algorithm	Prec@50	Rec@50	nDCG@50	mAP@50	Cov@50
GANMF-u	0.1834571	0.5169792	0.4205572	0.2200193	0.4173603
GANMF-i	**0.1931495**	**0.5375202**	**0.4434139**	**0.2400581**	0.3989298
DeepGANMF-u	0.0553128	0.1190691	0.0945475	0.0280326	**0.9869203**
DeepGANMF-i	0.1622481	0.4150829	0.3433792	0.1692178	0.5410226

Table 5.44: Results for cutoff 50 on MovieLens 100K. Higher values are better. Best results are in bold.

all cutoffs. Different from GANMF, where both user and item variants have very close scores across metrics, DeepGANMF shows quite some difference among variants.

Algorithm	Prec@5	Rec@5	nDCG@5	mAP@5	Cov@5
GANMF-u	0.4320199	**0.1083758**	**0.1692069**	0.3539966	0.2026444
GANMF-i	**0.4323179**	0.1053124	0.1690019	**0.358333**	0.1392337
DeepGANMF-u	0.1804305	0.0436618	0.0733999	0.1237536	**0.6799784**
DeepGANMF-i	0.1030132	0.0189744	0.0322978	0.0716291	0.0752833

Table 5.45: Results for cutoff 5 on MovieLens 1M. Higher values are better. Best results are in bold.

Algorithm	Prec@10	Rec@10	nDCG@10	mAP@10	Cov@10
GANMF-u	0.3702483	**0.1742579**	**0.2252712**	0.28728	0.264436
GANMF-i	**0.372351**	0.1692659	0.2243996	**0.2916971**	0.1826767
DeepGANMF-u	0.142202	0.0652266	0.0922406	0.0833942	**0.8367512**
DeepGANMF-i	0.090447	0.0328512	0.0447229	0.054566	0.1082029

Table 5.46: Results for cutoff 10 on MovieLens 1M. Higher values are better. Best results are in bold.

Algorithm	Prec@20	Rec@20	nDCG@20	mAP@20	Cov@20
GANMF-u	**0.3053808**	**0.2653345**	**0.2885205**	0.2417735	0.3416082
GANMF-i	0.304404	0.2552169	0.2852354	**0.2422803**	0.2417701
DeepGANMF-u	0.106697	0.0933383	0.1116552	0.0599598	**0.9368591**
DeepGANMF-i	0.0760844	0.0543757	0.0599065	0.0420953	0.1737723

Table 5.47: Results for cutoff 20 on MovieLens 1M. Higher values are better. Best results are in bold.

Algorithm	Prec@50	Rec@50	nDCG@50	mAP@50	Cov@50
GANMF-u	**0.2184503**	**0.4275134**	**0.3809177**	**0.2164192**	0.4695089
GANMF-i	0.2166623	0.4125733	0.3749247	0.2142928	0.3537507
DeepGANMF-u	0.0719338	0.1452663	0.1407886	0.0499875	**0.9905559**
DeepGANMF-i	0.1616821	0.2857345	0.2557555	0.1242929	0.3602267

Table 5.48: Results for cutoff 50 on MovieLens 1M. Higher values are better. Best results are in bold.

CiaoDVD

The results of DeepGANMF on CiaoDVD, the sparsiest dataset, are shown on tables 5.49, 5.50, 5.51 and 5.52. Even on this dataset we see GANMF having better results than DeepGANMF. Also the coverage of the best performing GANMF variant, GANMF-u is the highest. The best scoring DeepGANMF is the item-based version with performance at 50% that of GANMF-u. However the user-based version of DeepGANMF performs much worse, more than an order of magnitude worse than the item-based version. Also its coverage is more than 2 orders of magnitude smaller than GANMF-i.

Algorithm	**Prec@5**	**Rec@5**	**nDCG@5**	**mAP@5**	**Cov@5**
GANMF-u	**0.01387**	**0.0420354**	**0.0298209**	**0.0230707**	0.0535947
GANMF-i	0.0128736	0.0356792	0.0267557	0.0214877	**0.1059488**
DeepGANMF-u	0.0004195	0.0012839	0.0012123	0.0010728	0.0006823
DeepGANMF-i	0.0057944	0.0190632	0.0136923	0.0109967	0.0104832

Table 5.49: Results for cutoff 5 on CiaoDVD. Higher values are better. Best results are in bold.

Algorithm	**Prec@10**	**Rec@10**	**nDCG@10**	**mAP@10**	**Cov@10**
GANMF-u	**0.011366**	**0.0648433**	**0.0381765**	**0.0258148**	0.1134545
GANMF-i	0.0100026	0.0533817	0.0333071	0.0234155	**0.1547671**
DeepGANMF-u	0.000236	0.0012899	0.0012194	0.0010616	0.0014267
DeepGANMF-i	0.0045621	0.0310584	0.0177842	0.0123537	0.0206563

Table 5.50: Results for cutoff 10 on CiaoDVD. Higher values are better. Best results are in bold.

Algorithm	**Prec@20**	**Rec@20**	**nDCG@20**	**mAP@20**	**Cov@20**
GANMF-u	**0.0085212**	**0.0945504**	**0.046718**	**0.0279302**	**0.2396253**
GANMF-i	0.0076036	0.080545	0.0410068	0.0251971	0.2177284
DeepGANMF-u	0.0002032	0.0018843	0.0013991	0.0010962	0.0026673
DeepGANMF-i	0.0036117	0.046804	0.0222357	0.0134368	0.0446002

Table 5.51: Results for cutoff 20 on CiaoDVD. Higher values are better. Best results are in bold.

LastFM

Tables 5.53, 5.54, 5.55 and 5.56 show the results of DeepGANMF on LastFM, the only music dataset in this thesis. GANMF variants in this dataset perform very similar, something that we found interesting when comparing

Algorithm	Prec@50	Rec@50	nDCG@50	mAP@50	Cov@50
GANMF-u	**0.0055926**	**0.1492362**	**0.0592024**	**0.0299301**	**0.4378761**
GANMF-i	0.0050236	0.1251562	0.0513823	0.0267774	0.3358973
DeepGANMF-u	0.0005244	0.0148079	0.0043573	0.001566	0.006017
DeepGANMF-i	0.0024489	0.0764839	0.0286444	0.0143492	0.1111594

Table 5.52: Results for cutoff 50 on CiaoDVD. Higher values are better. Best results are in bold.

GANMF with the traditional MF baelines. Also DeepGANMF variants score close to one another but with the user variant coming on top. However its performance is still almost 3-4 times worse than that of GANMF-u, the best model among the 4. However DeepGANMF-u's coverage is the highest, with a ratio of 3 compared to GANMF-u's coverage. We also observe that increasing the recommendation list length, also the coverage of DeepGANMF changes a lot whereas GANMF-s coverage increases very little.

Algorithm	Prec@5	Rec@5	nDCG@5	mAP@5	Cov@5
GANMF-u	**0.2607219**	**0.1338915**	**0.1914327**	**0.2009306**	0.038623
GANMF-i	0.2562633	0.1318541	0.1867868	0.1935747	0.044578
DeepGANMF-u	0.0992569	0.0504045	0.0697229	0.067152	0.0359006
DeepGANMF-i	0.0677282	0.035634	0.0513143	0.0522775	**0.0924456**

Table 5.53: Results for cutoff 5 on LastFM. Higher values are better. Best results are in bold.

Algorithm	Prec@10	Rec@10	nDCG@10	mAP@10	Cov@10
GANMF-u	**0.1937898**	0.1968786	**0.2356293**	**0.1391564**	0.0565449
GANMF-i	0.1937367	**0.1973378**	0.2326003	0.1357974	0.0623866
DeepGANMF-u	0.0697983	0.0710499	0.0841033	0.0441018	0.1079855
DeepGANMF-i	0.0483546	0.0506163	0.0616444	0.0359391	**0.1600499**

Table 5.54: Results for cutoff 10 on LastFM. Higher values are better. Best results are in bold.

Delicious

We report the results of DeepGANMF on Delicious dataset on tables 5.57, 5.58, 5.59 and 5.60. We see a peculiar result from DeepGANMF-u, a score of almost 0 in all cutoffs. We suspect this is due to a mode collapse of the network or a divergence of the minimax game played by the two models. Still when compared to the other variant, GANMF scores higher in all metrics

Algorithm	Prec@20	Rec@20	nDCG@20	mAP@20	Cov@20
GANMF-u	**0.1360934**	0.2754476	**0.278796**	**0.1437082**	0.085583
GANMF-i	0.1360138	**0.2757899**	0.2755811	0.1405968	0.0896098
DeepGANMF-u	0.0437367	0.0887454	0.0938563	0.0423348	0.2682623
DeepGANMF-i	0.0327229	0.067643	0.0709883	0.0358842	**0.2684324**

Table 5.55: Results for cutoff 20 on LastFM. Higher values are better. Best results are in bold.

Algorithm	Prec@50	Rec@50	nDCG@50	mAP@50	Cov@50
GANMF-u	0.0795117	0.4014345	**0.3330374**	**0.1612095**	0.1620349
GANMF-i	**0.0796178**	**0.4019249**	0.3300618	0.1581174	0.1544918
DeepGANMF-u	0.0214013	0.1080092	0.1022785	0.0438043	**0.5764519**
DeepGANMF-i	0.0203185	0.1048153	0.0866545	0.039331	0.4769737

Table 5.56: Results for cutoff 50 on LastFM. Higher values are better. Best results are in bold.

and cutoffs, with a ratio of almost 5 on precision and almost 6 on recall. The coverage of DeepGANMF-i is also much lower than that of GANMF at all cutoffs.

Algorithm	Prec@5	Rec@5	nDCG@5	mAP@5	Cov@5
GANMF-u	**0.1383442**	**0.0549645**	**0.0777383**	**0.1189397**	**0.1041995**
GANMF-i	0.120915	0.0501615	0.0684077	0.1032746	0.0914436
DeepGANMF-u	0	0	0	0	0.0011268
DeepGANMF-i	0.009695	0.0034395	0.0047121	0.0062981	0.0247461

Table 5.57: Results for cutoff 5 on Delicious. Higher values are better. Best results are in bold.

Algorithm	Prec@10	Rec@10	nDCG@10	mAP@10	Cov@10
GANMF-u	**0.1251634**	**0.0952886**	**0.1096041**	**0.1021225**	**0.180807**
GANMF-i	0.1150327	0.0901394	0.0996606	0.0922297	0.1641073
DeepGANMF-u	0.0000545	0.0000363	0.0000269	0.0000054	0.0017769
DeepGANMF-i	0.0147059	0.0104842	0.0101901	0.0076004	0.0486977

Table 5.58: Results for cutoff 10 on Delicious. Higher values are better. Best results are in bold.

The previous experiment showed some interesting results that persisted across metrics and datasets. Given the added expressiveness that non-linearities bring into a model, especially with DNN, we were surprised to see empirically that substituting the single-layer autoencoder with a deeper autoen-

Algorithm	Prec@20	Rec@20	nDCG@20	mAP@20	Cov@20
GANMF-u	0.0946351	0.1418546	**0.1381279**	**0.0921853**	**0.2819294**
GANMF-i	**0.0958061**	**0.1456018**	0.1337833	0.0889318	0.2610982
DeepGANMF-u	0.0000272	0.0000363	0.0000269	0.0000036	0.0019647
DeepGANMF-i	0.0187636	0.025334	0.0197045	0.011027	0.0850873

Table 5.59: Results for cutoff 20 on Delicious. Higher values are better. Best results are in bold.

Algorithm	Prec@50	Rec@50	nDCG@50	mAP@50	Cov@50
GANMF-u	0.055915	0.2201512	**0.1728492**	**0.1032774**	**0.4228797**
GANMF-i	**0.0566231**	**0.2326523**	0.1707571	0.0998884	0.3675946
DeepGANMF-u	0.0000109	0.0000363	0.0000269	0.0000036	0.0019936
DeepGANMF-i	0.0125708	0.0415393	0.0281336	0.0139134	0.1201624

Table 5.60: Results for cutoff 50 on Delicious. Higher values are better. Best results are in bold.

coder and also changing the linear MF of the standard GANMF generator to a more complex and deeper generator did not bring any improvement in the score of GANMF. On the contrary, it worsened the performance for all the datasets. A second interesting result was the almost 100% coverage of DeepGANMF on the densiest datasets. A possible further investigation would be exploring a greater number of hyperparameter optimizations given the greater number of hyperparameters to explore.

Chapter 6

Conclusions

In the previous chapters we presented a new MF approach, GANMF, based on the GAN framework. While relatively new in the RS community, we showed that learning the user and item latent factors in a MF setting through GANs is possible and such approach can provide comparable and in some cases even better results than other, more traditional and established MF approaches. We also compared this new approach with one of state of the art techniques model-based techniques that builds a model for the item-item similarity matrix, SLIM.

Together with the baselines, PureSVD, WRMF, MF-BPR and SLIM-BPR, we tested GANMF on different well-known datasets in the RS community and evaluated all models' performances with accuracy and ranking metrics. GANMF was positioned as a strong candidate technique for building MF-based CF recommenders. Moreover on datasets like MovieLens 1M and LastFM, GANMF outperforms all other baselines providing even greater item coverage. This indicates that GANMF can provide richer recommendations by suggesting a broader spectrum of items.

We compared GANMF also with CFGAN, another GAN-based recommender that uses the same vector-wise training as GANMF. Even though CFGAN uses nonlinearities in its generator to produce plausible user/item profiles, GANMF with its linear MF operation performed better and, at selected datasets, led with substantial improvement in both accuracy and ranking metrics. The only dataset where CFGAN proved to be better than GANMF was Delicious.

We introduced GANMF as a GAN where the discriminator role is played by a shallow autoencoder taking queues from EBGAN. As explained in section 3.2.2, cGANs are trained with multiple data samples for each category with which the model is conditioned. This is not feasible in RS where for

each user/item we only have one historical profile. To account for this we included an additional loss term for the generator, the feature matching loss. Differently from the rationale of [46], we used feature matching to enforce conditioning on the generation process. We performed an ablation study to understand the importance of these component in the overall architecture and we empirically showed that they are important for achieving the best performance out of GANMF.

We can conclude that GANs and in particular our new approach, can be used to build robust RS models. We expect that in the future, generative modelling will play a more substantial role in the field of RS.

Bibliography

[1] Abadi, M., Agarwal, A., Barham, P., Brevdo, E., Chen, Z., Citro, C., Corrado, G. S., Davis, A., Dean, J., Devin, M., Ghemawat, S., Goodfellow, I., Harp, A., Irving, G., Isard, M., Jia, Y., Jozefowicz, R., Kaiser, L., Kudlur, M., Levenberg, J., Mané, D., Monga, R., Moore, S., Murray, D., Olah, C., Schuster, M., Shlens, J., Steiner, B., Sutskever, I., Talwar, K., Tucker, P., Vanhoucke, V., Vasudevan, V., Viégas, F., Vinyals, O., Warden, P., Wattenberg, M., Wicke, M., Yu, Y., and Zheng, X. TensorFlow: Large-scale machine learning on heterogeneous systems, 2015. Software available from tensorflow.org.

[2] Aggarwal, C. C., et al. *Recommender systems*, vol. 1. Springer, 2016.

[3] Anderson, C., and Andersson, M. P. Long tail.

[4] Antenucci, S., Boglio, S., Chioso, E., Dervishaj, E., Kang, S., Scarlatti, T., and Dacrema, M. F. Artist-driven layering and user's behaviour impact on recommendations in a playlist continuation scenario. In *Proceedings of the ACM Recommender Systems Challenge 2018*. 2018, pp. 1–6.

[5] Arjovsky, M., Chintala, S., and Bottou, L. Wasserstein gan. *arXiv preprint arXiv:1701.07875* (2017).

[6] Bell, R. M., and Koren, Y. Scalable collaborative filtering with jointly derived neighborhood interpolation weights. In *Seventh IEEE International Conference on Data Mining (ICDM 2007)* (2007), IEEE, pp. 43–52.

[7] Bellogín, A., Castells, P., and Cantador, I. Statistical biases in information retrieval metrics for recommender systems. *Information Retrieval Journal 20*, 6 (2017), 606–634.

[8] Bennett, J., Lanning, S., et al. The netflix prize. In *Proceedings of KDD cup and workshop* (2007), vol. 2007, Citeseer, p. 35.

[9] Cantador, I., Brusilovsky, P., and Kuflik, T. 2nd workshop on information heterogeneity and fusion in recommender systems (hetrec 2011). In *Proceedings of the 5th ACM conference on Recommender systems* (New York, NY, USA, 2011), RecSys 2011, ACM.

[10] Celma, Ò., and Cano, P. From hits to niches? or how popular artists can bias music recommendation and discovery. In *Proceedings of the 2nd KDD Workshop on Large-Scale Recommender Systems and the Netflix Prize Competition* (2008), pp. 1–8.

[11] Chae, D.-K., Kang, J.-S., Kim, S.-W., and Lee, J.-T. Cfgan: A generic collaborative filtering framework based on generative adversarial networks. In *Proceedings of the 27th ACM international conference on information and knowledge management* (2018), pp. 137–146.

[12] Chen, C.-W., Lamere, P., Schedl, M., and Zamani, H. Recsys challenge 2018: automatic music playlist continuation. In *Proceedings of the 12th ACM Conference on Recommender Systems* (2018), pp. 527–528.

[13] Chen, X., Duan, Y., Houthooft, R., Schulman, J., Sutskever, I., and Abbeel, P. Infogan: Interpretable representation learning by information maximizing generative adversarial nets. In *Advances in neural information processing systems* (2016), pp. 2172–2180.

[14] Cortes, C., and Vapnik, V. Support-vector networks. *Machine learning 20*, 3 (1995), 273–297.

[15] Cremonesi, P., Koren, Y., and Turrin, R. Performance of recommender algorithms on top-n recommendation tasks. In *Proceedings of the fourth ACM conference on Recommender systems* (2010), pp. 39–46.

[16] Creswell, A., White, T., Dumoulin, V., Arulkumaran, K., Sengupta, B., and Bharath, A. A. Generative adversarial networks: An overview. *IEEE Signal Processing Magazine 35*, 1 (2018), 53–65.

[17] Frazier, P. I. A tutorial on bayesian optimization. *arXiv preprint arXiv:1807.02811* (2018).

[18] Funk, S. Netflix Update: Try This at Home. `https://sifter.org/~simon/journal/20061211.html`, 2006. [Online; accessed 1-April-2020].

[19] Ge, M., Delgado-Battenfeld, C., and Jannach, D. Beyond accuracy: evaluating recommender systems by coverage and serendipity. In *Proceedings of the fourth ACM conference on Recommender systems* (2010), pp. 257–260.

[20] Gini, C. Variabilità e mutabilità (variability and mutability). *Tipografia di Paolo Cuppini, Bologna, Italy* (1912), 156.

[21] Goodfellow, I. Nips 2016 tutorial: Generative adversarial networks. *arXiv preprint arXiv:1701.00160* (2016).

[22] Goodfellow, I., Pouget-Abadie, J., Mirza, M., Xu, B., Warde-Farley, D., Ozair, S., Courville, A., and Bengio, Y. Generative adversarial nets. In *Advances in neural information processing systems* (2014), pp. 2672–2680.

[23] Guo, G., Zhang, J., Thalmann, D., and Yorke-Smith, N. Etaf: An extended trust antecedents framework for trust prediction. In *Proceedings of the 2014 International Conference on Advances in Social Networks Analysis and Mining (ASONAM)* (2014), pp. 540–547.

[24] Harper, F. M., and Konstan, J. A. The movielens datasets: History and context. *Acm transactions on interactive intelligent systems (tiis) 5*, 4 (2015), 1–19.

[25] Head, T., MechCoder, L., Shcherbatyi, I., et al. scikit-optimize/scikit-optimize: v0. 5.2, 2018.

[26] Herlocker, J. L., Konstan, J. A., Terveen, L. G., and Riedl, J. T. Evaluating collaborative filtering recommender systems. *ACM Transactions on Information Systems (TOIS) 22*, 1 (2004), 5–53.

[27] Hu, Y., Koren, Y., and Volinsky, C. Collaborative filtering for implicit feedback datasets. In *2008 Eighth IEEE International Conference on Data Mining* (2008), Ieee, pp. 263–272.

[28] Jebara, T. *Machine learning: discriminative and generative*, vol. 755. Springer Science & Business Media, 2012.

[29] Jones, D. R., Schonlau, M., and Welch, W. J. Efficient global optimization of expensive black-box functions. *Journal of Global optimization 13*, 4 (1998), 455–492.

[30] Jones, E., Oliphant, T., and Peterson, P. Scipy: Open source scientific tools for python.

[31] Koren, Y., Bell, R., and Volinsky, C. Matrix factorization techniques for recommender systems. *Computer*, 8 (2009), 30–37.

[32] Kramer, M. A. Nonlinear principal component analysis using autoassociative neural networks. *AIChE journal 37*, 2 (1991), 233–243.

[33] Linden, G., Smith, B., and York, J. Amazon. com recommendations: Item-to-item collaborative filtering. *IEEE Internet computing*, 1 (2003), 76–80.

[34] Lorenz, M. O. Methods of measuring the concentration of wealth. *Publications of the American statistical association 9*, 70 (1905), 209–219.

[35] Manning, C., Raghavan, P., and Schütze, H. Introduction to information retrieval. *Natural Language Engineering 16*, 1 (2010), 100–103.

[36] Mirza, M., and Osindero, S. Conditional generative adversarial nets. *arXiv preprint arXiv:1411.1784* (2014).

[37] Ng, A. Y., and Jordan, M. I. On discriminative vs. generative classifiers: A comparison of logistic regression and naive bayes. In *Advances in neural information processing systems* (2002), pp. 841–848.

[38] Nikolov, D., Lalmas, M., Flammini, A., and Menczer, F. Quantifying biases in online information exposure. *Journal of the Association for Information Science and Technology 70*, 3 (2019), 218–229.

[39] Ning, X., and Karypis, G. Slim: Sparse linear methods for top-n recommender systems. In *2011 IEEE 11th International Conference on Data Mining* (2011), IEEE, pp. 497–506.

[40] Oard, D. W., Kim, J., et al. Implicit feedback for recommender systems. In *Proceedings of the AAAI workshop on recommender systems* (1998), vol. 83, WoUongong.

[41] Pedregosa, F., Varoquaux, G., Gramfort, A., Michel, V., Thirion, B., Grisel, O., Blondel, M., Prettenhofer, P., Weiss, R., Dubourg, V., Vanderplas, J., Passos, A., Cournapeau, D., Brucher, M., Perrot, M., and Duchesnay, E. Scikit-learn: Machine learning in Python. *Journal of Machine Learning Research 12* (2011), 2825–2830.

[42] Rendle, S., Freudenthaler, C., Gantner, Z., and Schmidt-Thieme, L. Bpr: Bayesian personalized ranking from implicit feedback. *arXiv preprint arXiv:1205.2618* (2012).

[43] Ricci, F., Rokach, L., and Shapira, B. Introduction to recommender systems handbook. In *Recommender systems handbook*. Springer, 2011, pp. 1–35.

[44] Rogers, D. J., and Tanimoto, T. T. A computer program for classifying plants. *Science 132*, 3434 (1960), 1115–1118.

[45] Rumelhart, D. E., Hinton, G. E., and Williams, R. J. Learning representations by back-propagating errors. *nature 323*, 6088 (1986), 533–536.

[46] Salimans, T., Goodfellow, I., Zaremba, W., Cheung, V., Radford, A., and Chen, X. Improved techniques for training gans. In *Advances in neural information processing systems* (2016), pp. 2234–2242.

[47] Strub, F., Mary, J., and Gaudel, R. Hybrid collaborative filtering with autoencoders. *arXiv preprint arXiv:1603.00806* (2016).

[48] Tanimoto, T. T. Elementary mathematical theory of classification and prediction.

[49] Van Rossum, G., and Drake Jr, F. L. *Python tutorial*, vol. 620. Centrum voor Wiskunde en Informatica Amsterdam, 1995.

[50] Walt, S. v. d., Colbert, S. C., and Varoquaux, G. The numpy array: a structure for efficient numerical computation. *Computing in Science & Engineering 13*, 2 (2011), 22–30.

[51] Wang, J., Yu, L., Zhang, W., Gong, Y., Xu, Y., Wang, B., Zhang, P., and Zhang, D. Irgan: A minimax game for unifying generative and discriminative information retrieval models. In *Proceedings of the 40th International ACM SIGIR conference on Research and Development in Information Retrieval* (2017), pp. 515–524.

[52] Wang, K., Gou, C., Duan, Y., Lin, Y., Zheng, X., and Wang, F.-Y. Generative adversarial networks: introduction and outlook. *IEEE/CAA Journal of Automatica Sinica 4*, 4 (2017), 588–598.

[53] Wang, Y., Wang, L., Li, Y., He, D., and Liu, T.-Y. A theoretical analysis of ndcg type ranking measures. In *Conference on Learning Theory* (2013), pp. 25–54.

[54] Williams, R. J. Simple statistical gradient-following algorithms for connectionist reinforcement learning. *Machine learning 8*, 3-4 (1992), 229–256.

[55] Yao, Y., Rosasco, L., and Caponnetto, A. On early stopping in gradient descent learning. *Constructive Approximation 26*, 2 (2007), 289–315.

[56] Zhang, S., Yao, L., Sun, A., and Tay, Y. Deep learning based recommender system: A survey and new perspectives. *ACM Computing Surveys (CSUR) 52*, 1 (2019), 5.

[57] Zhao, J., Mathieu, M., and LeCun, Y. Energy-based generative adversarial network. *arXiv preprint arXiv:1609.03126* (2016).

Appendix A

Parameter Update Rules

A.1 Update rules for the discriminator

We continue in this section the derivations for the update rules for the parameters of the discriminator of GANMF as described in 3.2.1.

$$\mathcal{L}_D = \frac{1}{|B|}\sum_{i\in B} A + B$$

$$\frac{\partial \mathcal{L}_D}{\partial\{\mathbf{b}^D, \mathbf{b}^E, \Theta^D, \Theta^E\}} = \frac{1}{|B|}\sum_{i\in B} \frac{\partial}{\partial\{\mathbf{b}^D, \mathbf{b}^E, \Theta^D, \Theta^E\}}(A + C)$$

$$\begin{aligned}\frac{\partial}{\partial \mathbf{b}^D} A &= 2\Big(Dec\big(Enc(\mathbf{x}_i)\big) - \mathbf{x}_i\Big)\frac{\partial}{\partial \mathbf{b}^D} Dec\big(Enc(\mathbf{x}_i)\big) \\ &= 2\Big(Dec\big(Enc(\mathbf{x}_i)\big) - \mathbf{x}_i\Big)\, g'\Big(\mathbf{b}^D + \Theta^D\big(Enc(\mathbf{x}_i)\big)\Big)\end{aligned}$$

$$\frac{\partial}{\partial \mathbf{b}^D} C = \begin{cases} 0 & D\big(G(y)\big) \geq m \\ -2\Big(Dec\Big(Enc\big(G(\mathbf{y}_i)\big)\Big) - G(\mathbf{y}_i)\Big) & \\ \cdot \frac{\partial}{\partial \mathbf{b}^D} Dec\Big(Enc\big(G(\mathbf{y}_i)\big)\Big) & otherwise \end{cases}$$

$$= \begin{cases} 0 & D\big(G(y)\big) \geq m \\ -2\Big(Dec\Big(Enc\big(G(\mathbf{y}_i)\big)\Big) - G(\mathbf{y}_i)\Big) & \\ \cdot g'\Big(\mathbf{b}^D + \Theta^D\Big(Enc\big(G(\mathbf{y}_i)\big)\Big)\Big) & otherwise \end{cases}$$

$$\frac{\partial}{\partial \mathbf{b}^E} A = 2\Big(Dec\big(Enc(\mathbf{x}_i)\big) - \mathbf{x}_i\Big) \frac{\partial}{\partial \mathbf{b}^E} Dec\big(Enc(\mathbf{x}_i)\big)$$

$$= 2\Big(Dec\big(Enc(\mathbf{x}_i)\big) - \mathbf{x}_i\Big)\, g'\Big(\mathbf{b}^D + \Theta^D\big(Enc(\mathbf{x}_i)\big)\Big) \frac{\partial}{\partial \mathbf{b}^E} \Theta^D Enc(\mathbf{x}_i)$$

$$= 2\Big(Dec\big(Enc(\mathbf{x}_i)\big) - \mathbf{x}_i\Big)\, g'\Big(\mathbf{b}^D + \Theta^D\big(Enc(\mathbf{x}_i)\big)\Big) \Theta^D h'(\mathbf{b}^E + \Theta^E \mathbf{x}_i)$$

$$\frac{\partial}{\partial \mathbf{b}^E} C = \begin{cases} 0 & D\big(G(y)\big) \geq m \\ -2\Big(Dec\Big(Enc\big(G(\mathbf{y}_i)\big)\Big) - G(\mathbf{y}_i)\Big) \frac{\partial}{\partial \mathbf{b}^E} Dec\Big(Enc\big(G(\mathbf{y}_i)\big)\Big) & otherwise \end{cases}$$

$$= \begin{cases} 0 & D\big(G(y)\big) \geq m \\ -2\Big(Dec\Big(Enc\big(G(\mathbf{y}_i)\big)\Big) - G(\mathbf{y}_i)\Big) & \\ \cdot\, g'\Big(\mathbf{b}^D + \Theta^D\Big(Enc\big(G(\mathbf{y}_i)\big)\Big)\Big) \frac{\partial}{\partial \mathbf{b}^E} \Theta^D Enc\big(G(\mathbf{y}_i)\big) & otherwise \end{cases}$$

$$= \begin{cases} 0 & D\big(G(y)\big) \geq m \\ -2\Big(Dec\Big(Enc\big(G(\mathbf{y}_i)\big)\Big) - G(\mathbf{y}_i)\Big) & \\ \cdot\, g'\Big(\mathbf{b}^D + \Theta^D\Big(Enc\big(G(\mathbf{y}_i)\big)\Big)\Big) \Theta^D h'\big(\mathbf{b}^E + \Theta^E G(\mathbf{y}_i)\big) & otherwise \end{cases}$$

The partial derivative of both A and B (both column vectors) with respect to matrices Θ^D and Θ^E cannot be written in a matrix form so we show the partial derivatives of each element of A and B:

$$\frac{\partial}{\partial \Theta^D_{kl}} A_j = 2\Big(Dec\big(Enc(\mathbf{x}_i)\big) - \mathbf{x}_i\Big)_j \; \frac{\partial}{\partial \Theta^D_{kl}} \Big(Dec\big(Enc(\mathbf{x}_i)\big)\Big)_j$$

$$= 2\Big(Dec\big(Enc(\mathbf{x}_i)\big) - \mathbf{x}_i\Big)_j \; g'\Big(\mathbf{b}^D + \Theta^D\big(Enc(\mathbf{x}_i)\big)\Big)_j \, \delta_{jk} Enc(\mathbf{x}_i)_l$$

where δ_{ij} is the Kronecker delta with:

$$\delta_{ij} = \begin{cases} 0 & i \neq j \\ 1 & i = j \end{cases}$$

In a similar fashion we have:

$$\frac{\partial}{\partial \Theta^D_{kl}} C_j = \begin{cases} 0 & D\big(G(y)\big) \geq m \\ -2\Big(Dec\Big(Enc\big(G(\mathbf{y}_i)\big)\Big) - G(\mathbf{y}_i)\Big)_j & \\ \cdot \frac{\partial}{\partial \Theta^D_{kl}} Dec\Big(Enc\big(G(\mathbf{y}_i)\big)\Big)_j & otherwise \end{cases}$$

$$= \begin{cases} 0 & D\big(G(y)\big) \geq m \\ -2\Big(Dec\Big(Enc\big(G(\mathbf{y}_i)\big)\Big) - G(\mathbf{y}_i)\Big)_j & \\ \cdot g'\Big(\mathbf{b}^D + \Theta^D\Big(Enc\big(G(\mathbf{y}_i)\big)\Big)\Big)_j \ \delta_{jk} Enc\big(G(\mathbf{y}_i)\big)_l & otherwise \end{cases}$$

$$\frac{\partial}{\partial \Theta^E_{kl}} A_j = 2\Big(Dec\big(Enc(\mathbf{x}_i)\big) - \mathbf{x}_i\Big)_j \ \frac{\partial}{\partial \Theta^E_{kl}} \Big(Dec\big(Enc(\mathbf{x}_i)\big)\Big)_j$$

$$= 2\Big(Dec\big(Enc(\mathbf{x}_i)\big) - \mathbf{x}_i\Big)_j \ g'\Big(\mathbf{b}^D + \Theta^D\big(Enc(\mathbf{x}_i)\big)\Big)_j \ \frac{\partial}{\partial \Theta^E_{kl}} \Theta^D Enc(\mathbf{x}_i)$$

$$= 2\Big(Dec\big(Enc(\mathbf{x}_i)\big) - \mathbf{x}_i\Big)_j \ g'\Big(\mathbf{b}^D + \Theta^D\big(Enc(\mathbf{x}_i)\big)\Big)_j \ \Theta^D h'(\mathbf{b}^E + \Theta^E \mathbf{x}_i)_j \ \delta_{jk} (\mathbf{x}_i)_l$$

$$\frac{\partial}{\partial \Theta^E_{kl}} C_j = \begin{cases} 0 & D\big(G(y)\big) \geq m \\ -2\Big(Dec\Big(Enc\big(G(\mathbf{y}_i)\big)\Big) - G(\mathbf{y}_i)\Big)_j & \\ \cdot \frac{\partial}{\partial \Theta^E_{kl}} \Big(Dec\Big(Enc\big(G(\mathbf{y}_i)\big)\Big)\Big)_j & otherwise \end{cases}$$

$$= \begin{cases} 0 & D\big(G(y)\big) \geq m \\ -2\Big(Dec\Big(Enc\big(G(\mathbf{y}_i)\big)\Big) - G(\mathbf{y}_i)\Big)_j & \\ \cdot g'\Big(\mathbf{b}^D + \Theta^D\Big(Enc\big(G(\mathbf{y}_i)\big)\Big)\Big)_j \ \frac{\partial}{\partial \Theta^E_{kl}} \Theta^D \Big(Enc\big(G(\mathbf{y}_i)\big)\Big)_j & otherwise \end{cases}$$

$$= \begin{cases} 0 & D\big(G(y)\big) \geq m \\ -2\Big(Dec\Big(Enc\big(G(\mathbf{y}_i)\big)\Big) - G(\mathbf{y}_i)\Big)_j & \\ \cdot g'\Big(\mathbf{b}^D + \Theta^D\Big(Enc\big(G(\mathbf{y}_i)\big)\Big)\Big)_j & \\ \cdot \Theta^D h'\big(\mathbf{b}^E + \Theta^E G(\mathbf{y}_i)\big)_j \ \delta_{jk} \big(G(\mathbf{y}_i\big)_l & otherwise \end{cases}$$

A.2 Update rules for the generator

We continue in this section the derivations for the update rules for the parameters of the generator of GANMF as described in 3.2.2.

$$\frac{\partial A}{\partial \Sigma[\mathbf{y}_i]} = 2\left[Dec\Big(Enc\big(G(\mathbf{y}_i)\big)\Big) - G(\mathbf{y}_i)\right]\frac{\partial}{\partial \Sigma[\mathbf{y}_i]}\left[Dec\Big(Enc\big(G(\mathbf{y}_i)\big)\Big) - G(\mathbf{y}_i)\right]$$

$$= 2\left[Dec\Big(Enc\big(G(\mathbf{y}_i)\big)\Big) - G(\mathbf{y}_i)\right]\left[\frac{\partial}{\partial \Sigma[\mathbf{y}_i]}Dec\Big(Enc\big(G(\mathbf{y}_i)\big)\Big) - \frac{\partial}{\partial \Sigma[\mathbf{y}_i]}G(\mathbf{y}_i)\right]$$

$$\frac{\partial}{\partial \Sigma[\mathbf{y}_i]}Dec\Big(Enc\big(G(\mathbf{y}_i)\big)\Big) = g'\Big(\mathbf{b}^D + \Theta^D\Big(h\big(\mathbf{b}^E + \Theta^E\Sigma[\mathbf{y}_i]V\big)\Big)\Big)$$
$$\Theta^D h'(\mathbf{b}^E + \Theta^E\Sigma[\mathbf{y}_i]V)\Theta^E V$$

$$\frac{\partial}{\partial \Sigma[\mathbf{y}_i]}G(\mathbf{y}_i) = \frac{\partial}{\partial \Sigma[\mathbf{y}_i]}(\Sigma[\mathbf{y}_i]V) = V$$

$$\frac{\partial A_j}{\partial V_{kl}} = 2\left[Dec\Big(Enc\big(G(\mathbf{y}_i)\big)\Big) - G(\mathbf{y}_i)\right]_j\frac{\partial}{\partial V_{kl}}\left[Dec\Big(Enc\big(G(\mathbf{y}_i)\big)\Big) - G(\mathbf{y}_i)\right]_j$$

$$= 2\left[Dec\Big(Enc\big(G(\mathbf{y}_i)\big)\Big) - G(\mathbf{y}_i)\right]_j\left[\frac{\partial}{\partial V_{kl}}Dec\Big(Enc\big(G(\mathbf{y}_i)\big)\Big)_j - \frac{\partial}{\partial V_{kl}}G(\mathbf{y}_i)_j\right]$$

$$\frac{\partial}{\partial V_{kl}}Dec\Big(Enc\big(G(\mathbf{y}_i)\big)\Big)_j = g'\Big(\mathbf{b}^D + \Theta^D\Big(h\big(\mathbf{b}^E + \Theta^E\Sigma[\mathbf{y}_i]V\big)\Big)\Big)_j$$
$$\Theta^D h'(\mathbf{b}^E + \Theta^E\Sigma[\mathbf{y}_i]V)_j\Theta^E\delta_{jk}\Sigma[\mathbf{y}_i]_j$$

$$\frac{\partial}{\partial V_{kl}}(\Sigma[\mathbf{y}_i]V) = \delta_{jk}\Sigma[\mathbf{y}_i]_j$$

We give the partial derivatives of term C for user and item latent factors:

$$\frac{\partial C}{\partial \Sigma[\mathbf{y}_i]} = 2\Big[Enc(\mathbf{x}_i) - Enc\big(G(\mathbf{y}_i)\big)\Big]\frac{\partial}{\partial \Sigma[\mathbf{y}_i]}\Big[Enc(\mathbf{x}_i) - Enc\big(G(\mathbf{y}_i)\big)\Big]$$

$$= 2\Big[Enc(\mathbf{x}_i) - Enc\big(G(\mathbf{y}_i)\big)\Big]\left[\frac{\partial}{\partial \Sigma[\mathbf{y}_i]}Enc(\mathbf{x}_i) - \frac{\partial}{\partial \Sigma[\mathbf{y}_i]}Enc\big(G(\mathbf{y}_i)\big)\right]$$

$$\frac{\partial}{\partial \Sigma[\mathbf{y}_i]}Enc(\mathbf{x}_i) = 0$$

$$\frac{\partial}{\partial \Sigma[\mathbf{y}_i]}Enc\big(G(\mathbf{y}_i)\big) = \frac{\partial}{\partial \Sigma[\mathbf{y}_i]}\big(h(\mathbf{b}^E + \Theta^E\Sigma[\mathbf{y}_i]V)\big) = h'(\mathbf{b}^E + \Theta^E\Sigma[\mathbf{y}_i]V)\Theta^E V$$

$$\frac{\partial C_j}{\partial V_{kl}} = 2\left[Enc(\mathbf{x}_i) - Enc\big(G(\mathbf{y}_i)\big)\right]_j \frac{\partial}{\partial V_{kl}}\left[Enc(\mathbf{x}_i) - Enc\big(G(\mathbf{y}_i)\big)\right]_j$$

$$= 2\left[Enc(\mathbf{x}_i) - Enc\big(G(\mathbf{y}_i)\big)\right]_j \left[\frac{\partial}{\partial \Sigma[\mathbf{y}_i]} Enc(\mathbf{x}_i)_j - \frac{\partial}{\partial \Sigma[\mathbf{y}_i]} Enc\big(G(\mathbf{y}_i)\big)_j\right]$$

$$\frac{\partial}{\partial V_{kl}} Enc(\mathbf{x}_i)_j = 0$$

$$\frac{\partial}{\partial V_{kl}} Enc\big(G(\mathbf{y}_i)\big)_j = \frac{\partial}{\partial V_{kl}}\big(h(\mathbf{b}^E + \Theta^E \Sigma[\mathbf{y}_i]V)\big)_j = h'(\mathbf{b}^E + \Theta^E \Sigma[\mathbf{y}_i]V)_j \; \Theta^E \delta_{jk} \Sigma[\mathbf{y}_i]_j$$