Politecnico di Milano

Facoltà di Ingegneria
Scuola di Ingegneria Industriale e dell'Informazione
Dipartamento di Elettronica, Informazione e Bioingegneria

Master of Science in Computer Science and Engineering

Non-traditional approaches to input corruption in Collaborative Denoising Autoencoders

Supervisor:
Prof. Paolo Cremonesi

Master thesis of:
Laura Cristina Schiatti Sisó,
10594805

Academic Year 2019/2020

To God for his guidance at every moment of every day.
To my mom, sister and aunt Monica for their unconditional love, trust and support.
To my friends for being always there no matter the distance or time.
To all the friends made throughout these years for making Milan feel like home away from home.
To professor Paolo Cremonesi for his support and instruction throughout this project.

Abstract

Recommender systems became part of everyday life. They are present when choosing what movie to watch on *Netflix*, what book to buy on *Amazon*, and even to find friends on *Facebook*. Progressively, people are turning to these systems to help them interact more effectively with overwhelming amounts of content, and find the information that is most valuable to them.

Deep learning-based recommendation systems has increased in the past recent years to overcome the limitations of traditional models, specially when dealing with the enormous volume, complexity and dynamics of data. One successful application is to consider the collaborative filtering from an autoencoder perspective in order to achieve better results.

The collaborative denoising autoencoder (CDAE) is a flexible top-n recommender that uses a corrupted version of the user-item interactions to make recommendations. The type of process used to perform the corruption is an important aspect that substantially affects the final representation learned from the input, and therefore has an impact in the performance.

There are two core purposes for this work. To begin with, it is aimed to replicate the results of the experiments for the collaborative denoising autoencoder (CDAE) model proposed by Wu Y., DuBois C., Zheng A. X. and Ester M. [13], so as to have a clear starting point for further analysis and be able to understand the components of this recommender. Moreover, it proposes novel ways to introduce noise to the input data supplied to the CDAE aiming to overcome the issues inherent in state-of-the-art choices for corruption and improve performance. Usually, in such traditional approaches the noise is stochastically applied to the input, either by some additive mechanism or randomly masking some of the input values, and so, it is hard to have control over it.

Sommario

I sistemi di suggerimento sono diventati parte della vita quotidiana. Sono presenti quando si sceglie quale film guardare su *Netflix*, quale libro acquistare su *Amazon* e persino per trovare amici su *Facebook*. Progressivamente, le persone si rivolgono a questi sistemi per aiutarli a interagire in modo più efficace con una quantità enorme di contenuti e trovare le informazioni più preziose per loro.

I sistemi basati sul deep learning sono aumentati negli ultimi anni per superare i limiti dei modelli tradizionali, specialmente quando si ha a che fare con l'enorme volume, complessità e dinamica dei dati. Un'applicazione di successo consiste nel considerare il filtro collaborativo da una prospettiva di autoencoder al fine di ottenere risultati migliori.

Il collaborative denoising autoencoder (CDAE) è un top-n recommender flessibile che utilizza una versione danneggiata delle interazioni tra l'utente e gli oggetti per formulare raccomandazioni. Il tipo di processo utilizzato per eseguire la corruzione è un aspetto importante che ha impatto importante sulla rappresentazione finale appresa dall'input e quindi ha rilevanza sulla performance.

Ci sono due scopi principali per questo lavoro. Per cominciare, ha la finalità di replicare i risultati degli esperimenti per il collaborative denoising autoencoder (CDAE) proposto da Wu Y., DuBois C., Zheng AX e Ester M. [13], in modo da avere un chiaro punto di partenza per ulteriori analisi e poter comprendere i componenti di questo suggeritore. Inoltre, propone nuovi modi per introdurre il rumore nei dati di input forniti al CDAE con l'obiettivo di superare i problemi inerenti alle scelte all'avanguardia per la corruzione. Di solito, in tali approcci tradizionali il rumore viene applicato in modo stocastico all'input, o mediante un meccanismo additivo o mascherando casualmente alcuni dei valori di input, e quindi è difficile avere il controllo al riguardo.

Contents

1 Introduction **3**

1.1 Context . . . 3

1.2 Problem definition . . . 4

1.2.1 Proposed work . . . 5

1.3 Structure . . . 5

2 State of the art **7**

2.1 Collaborative Filtering . . . 7

2.1.1 Memory-based Approaches . . . 8

2.1.2 Model-based Approaches . . . 11

2.2 Deep Learning-Based Recommender Systems . . . 12

2.2.1 Deep Learning for Recommendation . . . 12

2.2.2 Overfitting Problem in Deep Learning . . . 14

2.3 Autoencoder based Recommendation . . . 15

2.3.1 Fundamentals of autoencoders . . . 16

2.3.2 Collaborative Denoising Autoencoders (CDAE) . . . 17

2.4 Evaluation . . . 23

2.4.1 Dataset splitting . . . 23

2.4.2 Recommender Systems Evaluation . . . 25

3 Input Corruption Formalization **29**

3.1 Traditional input corruption . . . 29

3.1.1 Multiplicative mask-out/drop-out corruption . . . 30

3.2 Alternative approaches to input corruption . . . 31

3.2.1 Input corruption without replacement . . . 32

3.2.2 Alternative approach to without replacement corruption . . . 34

3.2.3 Input corruption with replacement . . . 34

4 Experimental results **37**

4.1 Datasets . . . 37

4.2 Experimental Setup 38
4.3 Model performance comparison 39
4.3.1 Experiments 39
4.3.2 Results 41
4.4 Performance of different input corruption strategies 43
4.4.1 Experiments 43
4.4.2 Results 45

5 Conclusions **53**

A CDAE Implementation Details **57**

Chapter 1

Introduction

1.1 Context

Recommender systems are a sub-field of information filtering systems designed to generate meaningful recommendations to users based on their preferences. They are aimed at suggesting items that might be of interest to users, for instance, books on *Amazon*, songs on *Spotify*, or movies on *Netflix*, all of these are real-world examples of recommender systems.

More formally, the recommendation problem can be seen as a user preferences learning problem in which information about past behaviour of users towards the items in the system, as well as the attributes that describe the items, are leveraged to make predictions about which items are worth recommending to a user.

There are two approaches to formulate this problem, as a *rating prediction* or *ranking prediction* task. The first approach is devoted to estimate the rating value a user will give to an item. However, in practical applications, the goal is not necessarily to look for specific ratings values of user-item combinations, but to learn, for every user, a top-N list of the most relevant items. This is the main task to be considered throughout this thesis.

One of the challenges faced by recommender systems, is to choose the proper data structures that would allow an easy and efficient access to such sources of data. There are two main matrices devoted for that purpose, the User Rating Matrix (URM) and the Item Content Matrix (ICM). The URM is a sparse data structure containing the user-item interactions, such as ratings or buying behavior. On the other hand, the ICM contains descriptive information about the items in the system in the form of item-attribute pairs indicating the relevance of an attribute for an item. The goal of a recommender system is to generate predictions for the unknown elements of the URM, thus, to predict the user's possible ratings.

In general, recommender systems can be distinguished based on the nature of the

data used to make inferences and how the available information is filtered out to make the recommendations. *Collaborative filtering* methods rely on the URM, while the *content-based* uses both, the ICM and URM, to be able to provide personalized recommendations based on past preferences of the users. There most common models in literature can be summarized as

- **Content-Based Filtering:** uses the descriptive attributes of items and the target user's own ratings to provide recommendations. The term "content" refers to these descriptions.
- **Collaborative Filtering:** the collaborative approach does not relay on attributes of items (or users) but on the opinion of a community. It leverages the ratings provided by multiple users to make recommendations.
- **Hybrid Filtering:** combines the capabilities of several approaches so as to improve the quality of the recommendations. It has been proved than an ensemble of models can provide more accurate recommendations than pure approaches.

1.2 Problem definition

From a practical perspective, the use of recommender systems vary from application to application, but in general, they seek to understand users preferences and discover their interests, so as to provide the most appealing recommendations. Currently, a large amount of descriptive information about items and users is available in real-world applications that can be exploited to achieve a better recommender, but at the same time suppose the challenge of dealing with the exponential growth of information available.

Nowadays, many approaches to recommender systems have shown outstanding results but still need to face challenges, thus a lot of work has been devoted to find alternatives, that can be integrated so as to overcome the prominent issues of the recommendation problem.

The collaborative denoising autoencoder (CDAE) [3] for top-n recommendation has been proposed to deal with these challenges gaining better results with respect to conventional recommender systems. It exploits the well-known denoising trick in which the model learns from artificially corrupted data with significant improvements on performance and robustness.

The type of noise introduced to the input has to be chosen, and is an important aspect that affects the final representation learned. The idea is to end up with a robust representation able to retain the most relevant information about the input. Common corruption mechanisms for denoising autoencoders are in essence stochastic that are difficult to control. Indeed CDAE works under the mask-out/drop-out corruption that randomly masks entries of the input by making them zero.

1.2.1 Proposed work

This thesis is aimed to replicate the experiments for the collaborative denoising autoencoder reported in [3] so as to check whether the obtained results are similar, better or worse. Having a clear starting point will allow to make some modifications to this standard implementation that can provide more flexibility to the model in terms of the type of noise adopted.

Moreover, two alternative approaches to input corruption are proposed: *with* and *without* replacement corruption aiming to obtain a performance enhancement with respect to the *mask-out/drop-out* state-of-the-art approach. These mechanisms differ to traditional choices for corruption in terms of the control over the noise introduced to the input.

1.3 Structure

The work is structured as follows: Chapter 2 introduces the state of the art, a summary of the basics on collaborative filtering-based recommenders, deep learning models applied to recommendation, and collaborative denoising autoencoders as well as the evaluation tools needed. Chapter 3 proceeds with a formalization and characterization of the alternative input techniques proposed in this work. Chapter 4 gives a detailed description of the datasets and preprocessing steps applied, also the general setup to be considered for conducting the experiments, and the results obtained. Finally, in Chapter 5 a discussion over the results is presented and conclusions are extracted.

Chapter 2

State of the art

This chapter is organized as follows. In section 2.1 there is an introduction to the basic notions of collaborative-based recommenders. Section 2.2 presents different deep learning models used to overcome the limitations of conventional approaches to collaborative filtering. In section 2.3 autoencoder based recommenders are presented. Section 2.4 provides an introduction of the most used evaluation metrics used in recommender systems.

2.1 Collaborative Filtering

Collaborative filtering (CF) [6], is considered to be the most popular and widely implemented technique in recommender systems. In literature, CF approaches yield to better results with respect to content-based filtering approaches, as they are able to deal with some limitations. For instance, items for which the content is not available or difficult to obtain can still be recommended to users through the feedback of other users. Furthermore, recommendations are based on the quality of items as evaluated by other users, instead of relying on content that may be a bad indicator of quality [12].

The collaborative filtering methods rely on the opinions of a community of users to make recommendations. This approach works under the principles that user preferences stay stable over a period of time, and users that had similar inclinations in the past, are likely to be interested in similar items in the future. Basically, these systems are intended to filter out among the possible items to be recommended to users according to the preference patterns of similar users extracted from the User Rating Matrix (URM).

Among the sources of data needed to produce recommendations, the interactions between users and items (i.e. elements of the URM) are the most relevant to understand user's preferences. In practice, the users offer two types of feedback; implicit and explicit. *Explicit* means the user gives an indicative value to the item typically expressed on a scale corresponding to the level of satisfaction with the item at hand, such as 5-star scale like {1,2,3,4,5}. Diversely, *implicit* ratings are those user-item interactions collected from the user's actions rather than an explicit expression of interest, where by action is meant

the simple act of a user viewing or buying an item, etc.

In general, using collaborative filtering methods leads to significant improvements in recommendation quality. Although, there are some challenges these approaches still need to deal with [10]:

- **Sparsity of the data**. The most challenging aspect of CF techniques is to cope with rating matrices which are typically sparse [1]. Large levels of sparsity can lead to less accuracy and poor quality recommendations. The URM is highly sparse as each user is very likely to interact only with a small subset of all the items in the system, and therefore most entries are unknown.

  An additional problem caused by the data sparsity is the *cold start problem* arising when a new user o item enters the system. For these recently added elements, there is a very limited amount of data for the recommender to work with, and therefore it becomes almost impossible to provide reliable recommendations.

- **Scalability problem.** CF suffers from poor scalability. As the amount of users and items grow, it becomes difficult for the collaborative approach to work since computations are very expensive and the computational resources to satisfy this new requirements are not enough.

There are two kind of algorithms used in collaborative filtering, which are referred to as *memory-based* and *model-based*. The memory-based use past ratings directly in the prediction, so they need the whole dataset to make recommendations. In contrast, the model-based algorithms, as the name suggests, build a predictive model from the ratings to generate recommendations. Once the model is obtained, it does not need the raw data any more in the recommendation process [12].

2.1.1 Memory-based Approaches

The memory-based methods, also referred to as neighborhood-based methods, were the earliest collaborative filtering algorithms used. These algorithms act on the entire URM. They assume that users (and items) can be grouped together by similarity, based on the fact that similar users have similar preferences towards the items, and similar items receive similar ratings.

[1] In a *sparse matrix* most of the elements are zero. The opposite to that is called *dense*

As described in [1], memory-based CF methods address the recommendation problem by means of two different types of algorithms: *user-based* and *item-based*. Both approaches differ in the way in which the neighborhoods are defined, and thus, the ratings are predicted. In the former case, the neighborhoods are defined by similarities among users and the ratings of neighboring users are used for the predictions, whereas in the latter, the similarities are computed among items and the ratings are predicted using the the user's own ratings on closely related (i.e. neighboring) items.

- **User-based CF**. The basic idea is to search for the like-minded users of a target user u, those that have rated items similarly, and therefore tend to have similar taste. With user-based the prediction of a rating is computed from the ratings given by other users on the same item.

  The estimator of a rating $\hat{r}_{ui}$ given by a user u to a new item i, is computed as the average rating given to i by u's k-nearest neighbor users, the k most similar users v, that have rated i [12]. Consider $sim(u, v)$ as the similarity between two users u and v, U represents the $KNN(u)$, and as the r_{vi} ratings provided by neighbors of u to item i.

$$\hat{r}_{ui} = \frac{\sum_{v \in U} r_{vi} \; . \; sim(u, v)}{\sum_{v \in U} \; sim(u, v)} \tag{2.1}$$

- **Item-based CF**. The main principle considered is that a user give similar ratings to similar items [1]. In order to make the rating predictions for a target item by a user, the first thing is to determine the set of items that are the most similar to the target item. This similarity between each pair of items is calculated according to how many users have rated them both. Then, the predicted rating $\hat{r}_{ui}$ is computed from the ratings specified by the user on the most similar items to i.

  The rating estimator is the weighted average of the ratings provided by the same user on other items, where the weights are the similarities between items. The similarity $sim(i, j)$ is defined between two items i and j, I represents the $KNN(i)$, and as the r_{uj} ratings provided by u to similar items j.

$$\hat{r}_{ui} = \frac{\sum_{j \in I} r_{uj} \; . \; sim(i, j)}{\sum_{j \in I} \; sim(i, j)} \tag{2.2}$$

Similarity Measures

Many content-based and collaborative filtering recommender systems require a similarity or distance measure to work. Therefore, it is necessary to define an appropriate similarity measure able to efficiently calculate similarities on large ratings matrices. Based on the approach, the similarity, can be defined either between users or between items.

Several similarity function are used in practice, such as the cosine, Pearson correlation, Euclidean similarity. To make it possible to calculate such similarity, vectors representing user profile (i.e. rating provided by the user) as well as the item content (i.e. attributes describing the items), are defined.

- **Cosine similarity**. The cosine similarity measures, the cosine of the angle between two vectors. The vectors are typically non-zero and are within an inner product space. The angle will be in the range 0 to 180 degrees, regardless of how many dimensions the space has. Given two vectors, A and B be, in an n-dimensional space, the cosine similarity between them is defined as:

$$similarity = cos(A, B) = \frac{A \,.\, B}{||A||\ ||B||} \tag{2.3}$$

  where **.** represents the dot product between the two vectors, and $||A||$ is the magnitude of the vector A. The resulting similarity ranges from -1, exactly opposite, to 1, meaning identical, while 0 means orthogonality.. As the similarity gets closer to 1, then the angle between the two vectors A and B is smaller.

- **Pearson correlation**

  The similarity between items (or users) can also be given by their correlation which measures the linear relationship between objects [5]. Among the possible correlation coefficients, the Pearson correlation coefficient (ρ) is the most commonly used. Given a pair of random variables X and Y, the correlation ρ is defined as:

$$\rho_{X,Y} = \frac{cov(X, Y)}{\sigma_X \,.\, \sigma_Y} \tag{2.4}$$

  where *cov* is the covariance, and σ_X and σ_Y are the standard deviations of X and Y, respectively. It is noteworthy that the Pearson coefficient has a value between -1 and 1. Being 1 a total positive linear correlation, 0 no linear correlation, and 1 total negative linear correlation.

2.1.2 Model-based Approaches

Model-based approaches to collaborative filtering seek to train a machine learning, or data mining model based on sparse data [1]. The model is built by fitting a generic model to a portion of the known dataset, and then it is used to predict user ratings to provide recommendations. Therefore, the training (or model building phase) is clearly separated from the prediction phase.

Generally, since the model-based algorithms have the ability to capture the insights of the data, they tend to provide more accurate results than memory-based algorithms. Indeed, their main advantage is the robustness against the data sparsity, and thus the improvement of the prediction performance. However, depending on the chosen model and on the dataset size, the model building phase can be computationally expensive.

The most common techniques include *Sparse LInear Methods (SLIM)* [11], and *Matrix Factorization (MF) methods* [14], among others.

- **Sparse LInear Methods**. It is the machine learning approach to traditional item-based CF, which tries to estimate the rating $\hat{r}_{ui}$ a user would give to an item, based on the similarity with other items the user rated. The similarities are then used to build an item similarity matrix S. Besides, SLIM produces high quality recommendations than the state-of-the-art methods at a very high speed, so it is better suitable for real-time applications [11].

  The main difference between both approaches is that *KNN item-based CF* is highly dependant on the pre-specified item-item similarity measure to identify the neighbors, while *SLIM* generates S by solving the optimization problem. In SLIM the model learns a sparse similarity matrix S for the items in the system using the user profiles as the ground-truth, by solving a regularized optimization problem.

  The model is mainly a linear regression with combined $l1$ and $l2$ prior regularizers and the unknown parameters are the elements of S. Having both $l1$-norm and $l2$-norm regularizers is also known as *elastic net regularization*.

  $$\hat{R} = R \,.\, S$$

  where $R \in [0,1]^{m \times n}$ is a binary URM, $S \in \mathbb{R}^{n \times n}$ a matrix of aggregation coefficient, n is the number of items and m the number of users. The model learns matrix S using the training URM by minimizing the next regularized optimization problem:

$$\underset{S}{minimize}\ \frac{1}{2}\ ||R - RS||_F^2 + \frac{\beta}{2}\ ||S||_F^2 + \alpha\ ||S||_1$$
$$subject\ to\ S \geq 0$$
$$diag(S) = 0$$

To enforce sparsity and to reduce overfitting while minimizing the error, some regularization terms are added to the loss value. Then it contains the sum of three terms. The first being the real error of our model (i.e. residual sum of squares), measures how well the linear model fits the training data. The last two terms correspond to the elastic net regularizer, which combines a *l1*-norm (or Lasso) and *l2*-norm (or Ridge) of S. The coefficients α and β are the regularization parameters and . The larger the parameters are, the more severe the penalties [11].

The non-negativity constraint is applied on S, such that the learned S represents positive relations between items, if any. An additional constraint is included so as to have all elements of the diagonal of S set to zero. This avoids having all elements in the diagonal equal to 1, with an error of zero, such that an item always recommends itself.

2.2 Deep Learning-Based Recommender Systems

As it was previously discussed, collaborative filtering methods are extensively used in recommender systems since they are able to achieve a good performance in the task of providing personalized recommendations, but there are some limitations they still need to overcome. For instance, dealing with sparse ratings matrices, expensive computations due to the high volumes of data to be processed, and in some cases, not being able to capture the nonlinearities present in users' preferences.

In recent years, a significant research effort has been dedicated to find alternatives to traditional collaborative approaches to recommender systems. These new approaches seek to overcome the obstacles of conventional models, and to improve their modeling capacity by capturing complex relations present in real-world applications, and precisely reflecting the user's preference.

2.2.1 Deep Learning for Recommendation

Deep learning can be considered as a sub-field of machine learning devoted to capturing multiple levels of representations and abstractions from data (i.e. learning deep representation).

It has proved to provide outstanding results in many application domains, such as computer vision, machine translation and speech recognition due to its capability in solving many complex tasks, in both supervised and unsupervised learning tasks.

Several works have emerged to apply such techniques to information retrieval and recommender systems [7]. In essence, to take advantage of the strengths of deep learning methodologies to face recommendation challenges gaining better results with respect to conventional recommender systems [3].

Some of the properties of deep learning that make attractive tools for the development of any recommender system are [15]:

- **Nonlinear modeling**. Conventional recommenders are essentially, which leads to oversimplification and limits the model expressiveness. In contrary, deep models can capture non-linear patterns within data by means of non-linear activation functions. Going beyond linear modelling provide a gran insight of the complex patterns within user-item interactions and precisely reflect user preferences and needs.

- **Representation learning**. Deep models are effective to learn useful representations from the input data. In this case, high-order features (i.e. complex abstractions) can be extracted from the vast amount of descriptive information about items and users which may be useful for recommendation.

- **High flexibility**. Deep learning techniques are highly flexible. Most of the available tools are modular allowing the community to use them as building blocks that can be combined in several ways. For instance, neural structures can be used to formulate models that can boost the recommender performance.

- **Sequence Modelling**. Deep neural networks are useful to mine sequential structures in data. In the field of recommender systems, they provide a great contribution by allowing to have a better understanding of the temporal dynamics of user behaviour and item evolution.

Despite of the wide list of benefits obtained from deep learning methodologies for recommendation, there are potential limitations to be tackled:

- **Interpretability**. Deep learning methods usually behave as black boxes, and due to the complexity of elements involved (e.g. hidden weights, activations, etc.) are non-interpretable, so we can be aware of what is happening without necessarily

knowing why. In addition, this limits explainability, thus it is hard that the system can be explained by human intuition.

- **Data Requirements**. The amount of data required depends both on the complexity of the problem and the chosen algorithm. Typically, deep learning is known to be data-hungry, meaning it needs sufficient data for building predictive models. However, in the context of recommender systems is relatively easy to collect a significant amount of data.
- **Extensive Hyperparameter Tuning**. Hyperparameter tuning is not only a problem for deep learning, but for machine learning in general. It is a latent problem any time it is required to choose a set of optimal hyperparameters for a learning algorithm.

As described in [15], the state-of-the-art models can be categorized according to the deep learning technique employed into two main families. The *recommendation with neural building blocks* in which a single deep model is used, and a second group for *recommendation with deep hybrid models* referring cases in which more than one technique is used by putting together several neural building blocks.

The rationale is to incorporate deep learning models as neural extensions of traditional recommendation methods to achieve reasonable performance gains. *Multilayer perceptron (MLP)* can be used to add nonlinearitities to existing recommender approaches so as to learn hierarchical feature representations, *Autoencoders (AE)* to reconstruct its input data in the output layer, *Convolutional Neural Network (CNN)* to capture the global and local features, *Recurrent Neural Networks (RNN)* suitable for sequential data processing, and so on. Autoencoders based recommendation the collaborative filtering setting will be described in more detail later.

2.2.2 Overfitting Problem in Deep Learning

Overfitting is a serious problem in many machine learning algorithms, in which the prediction is influenced by random fluctuations in the data. It happens when a model learns each detail and noise in the training data, so it will not perform well on new, previously unseen inputs. The ability to achieve a good performance on previously unobserved inputs is called generalization.

Autoencoders, as any neural network, are extremely prone to overfitting because they are very powerful models in terms of model complexity (i.e. number of adjustable

parameters). Several techniques have been developed to prevent overfitting in this type of models, like *early stopping* in which the learning process is interrupted before the model starts to fit the noise; *weights decay* to penalize more complex models by using weights regularization based on prior assumptions; and *dropout* which performs a stochastic regularization by turning off some neurons randomly.

2.3 Autoencoder based Recommendation

There are several variants of traditional autoencoders that can be exploited in recommender systems field. Such types include denoising autoencoders (DAE), variational autoencoders (VAE) and contractive autoencoders (CAE). The architectures to be discussed next are shown in Figure 2.1.

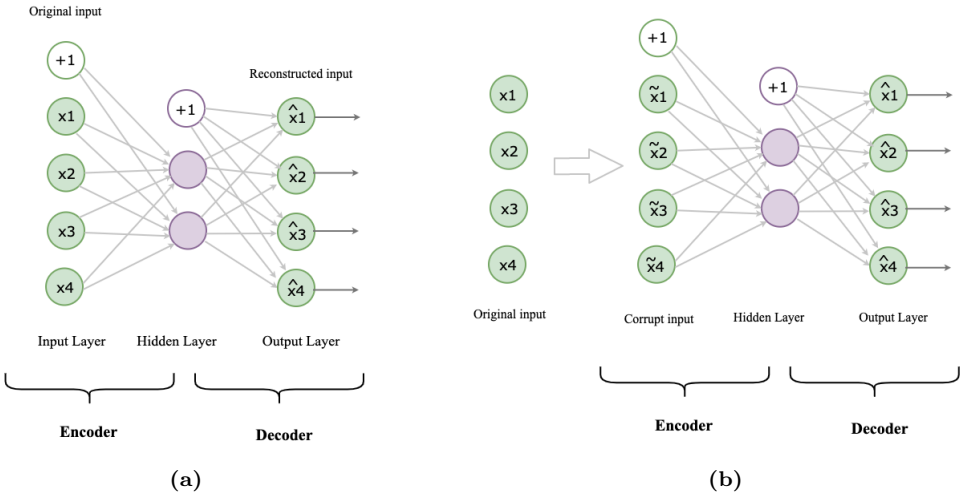

Figure 2.1: Architectures of (a) Classical autoencoder; (b) Denoising autoencoder.

Autoencoders can be applied to recommender systems for two main purposes, to learn lower-dimensional feature representation (at the bottleneck layer), or to fill in the unknown elements in the URM directly in the reconstruction layer. As regards to the latter, there are several state-of-the-art works devoted to both, rating prediction and top-N recommendation tasks [15].

- **AutoRec** is an autoencoder framework for collaborative filtering. The underlying intuition is to take user, or item partial vectors as inputs according to the variant to

be implemented (i.e. item-based AutoRec (I-AutoRec) and user-based AutoRec), and try to reconstruct them in the output layer.

- **Collaborative Denoising Autoencoder (CDAE)** is dedicated to address the general top-N recommendation problem using an extended denoising autoencoder. Unlike the previous model, the input for the CDAE corresponds to a subset of the observed implicit feedback, and the model is trained to recover the whole preference set. These is the major part of our work.

- **Muli-VAE and Multi-DAE** propose an alternative to the conventional VAE for recommendation with implicit data.

2.3.1 Fundamentals of autoencoders

Autoencoders (AE) are common building block of deep learning architectures. In general, are unsupervised models for data reconstruction. The bottleneck layer (i.e. middle-most layer) imposes the network to learn a compressed representation of the original input data. The output is then aimed at reconstructing the input.

Traditional Autoencoders

An autoencoder is typically implemented as a one-hidden-layer feedforward neural network[2] trained to reconstruct its input data in the output layer [13]. It consists of two parts, an *encoder* that takes the original input $x \in [0,1]^D$ and produces a lower-dimensional representation $z \in \mathbb{R}^K$ using a mapping function z.

$$z = h(x) = \sigma(W^T x + b) \tag{2.5}$$

where $W \in \mathbb{R}^{D \times K}$ is a weights matrix connecting neurons in input layer to the hidden neurons, and $b \in \mathbb{R}^K$ is a bias term. Being D the neurons in the input layer, K the nodes in the hidden layer with $N << D$. Then, the *decoder* reconstructs from the compressed representation $\hat{x} \in [0,1]^D$.

$$\hat{x} = g(z) = \sigma(W' z + b') \tag{2.6}$$

The parameters inside the network, weights and biases, W, W', b, b' are learned by minimizing the average reconstruction error, being $l(.)$ the loss function such as square loss, and n the total number of training examples:

[2]information flows in one direction, from the input nodes towards the output nodes.

$$\underset{W,\, \hat{W}',\, b\, ,b'}{argmin}\ \frac{1}{n} \sum_{n} l(x_i, \hat{x_i}) \tag{2.7}$$

Denoising Autoencoders

In general, the more hidden layers in an autoencoder, the more refined is the representation learned from the input data. However, if the number of hidden layers exceeds the number of inputs, the network is at risk of learning data, meaning that the output equals the input. Then, the autoencoder becomes useless, because it does not perform any useful representation learning or dimensionality reduction. This limitation leads to a variant of the basic autoencoder described above.

A denoising autoencoder (DAE) [5] can solve the problem of learning the identity function by corrupting the input data on purpose, adding noise or masking some of the input values. Therefore, a DAE extends the classical autoencoder by adding a stochastic corruption step operating in the input, and then it is trained to "denoise" or recover the original input at the output layer. Undoing the effect of the corruption process can be done only by capturing the statistical dependencies between the inputs.

Essentially, the noisy inputs are provided to the DAE, then the *encoder* part of the network will try to get the compress latent space representation. And the *decoder*, will reconstruct the inputs.

2.3.2 Collaborative Denoising Autoencoders (CDAE)

Matrix Factorization models are considered one of the basic but effective techniques used in collaborative filtering [14]. Basically, are representation learning methods intended to learn the user and item latent factors to predict unknown user preferences. Formally, CDAE is a straightforward generalization of the latent factor model described above. The fundamental difference being that CDAE uses an autoencoder which is a two-layer NN with nonlinear transformation.

CDAE [13] extends a standard DAE with an additional latent vector V_u that is encoded for each user. The underlying intuition is to use the DAE to learn the distributed embedding vectors of users and items by training a corrupted version of the known preference set provided by the user towards the items in the system.

The CDAE works well on implicit feedback, on a binary preference set containing only information about whether a user interacted with an item or not. If a user watches a movie, the system infers that the user is interested, and the corresponding entry value is 1, otherwise 0.

CDAE Structure

The CDAE structure for a generic user u is shown in Figure 2.2. It consists of 3 layers, including the input layer, the hidden layer and the output layer. The nodes in different layers are connected through weights. All weights but V_u are shared across all the users.

- The *input layer* contains $I + 1$ nodes. The first I nodes are known as *item input nodes*, represent user preferences, and each of them corresponds to an item. The last node is the user-specific node called *user input node*, whose associated weights are unique for each user.

  Given O containing the user past preferences of items, the info about whether an item is preferred or not by a user. O is transformed in a training set containing U instances $\{y_1, y_2, \ldots, y_u\}$. The idea is having $y_u = \{y_{u1}, y_{u2}, \ldots, y_{uI}\}$ as the I-dimensional feedback vector of user u in all items in I. Therefore, y_u is a sparse binary vector containing $y_{ui} = 1$ if $I \in O_u$, and $y_{ui} = 0$ otherwise. Where O_u is the set of preferences in the training set for a particular user u.

- In the *hidden layer* there are K nodes and are fully connected to those in the input layer. K is a predefined constant usually much smaller than the size of the input nodes (i.e. $K << I + 1$).

- The *output layer* contains I nodes representing reconstructions of the input vector y_u, and are fully connected to the nodes in the hidden layer.

CDAE Learning Algorithm

Briefly, the first step corresponds to the initialization of model parameters. The weights and biases are initialized randomly. Then, during training, the model is fed with a subset of a user's input set and then the model needs to recover the whole preference set. At prediction time, the model takes u's uncorrupted preference set as input. And then, new items (not in the existing preferences), that have the largest prediction values on the output later are recommended to u.

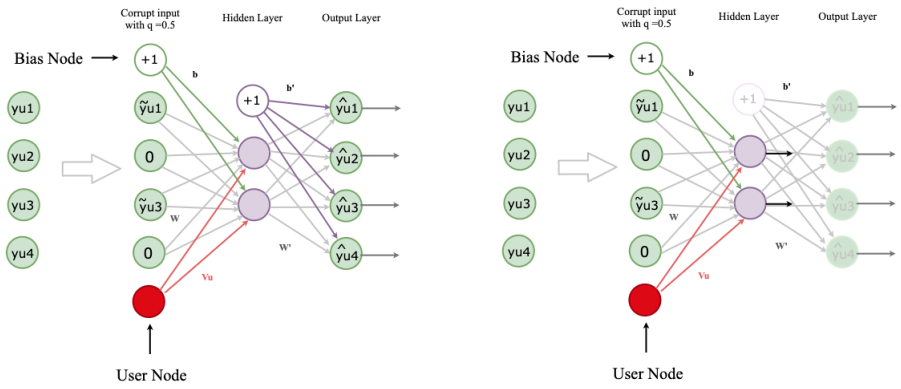

Figure 2.2: Sample CDAE structure for a user u. (a) CDAE during training phase; (b) Extract hidden node representation of input with trained CDAE.

The predictions for user u on the set of items in the system are collected in a vector $\hat{y}_u$, computed as follows:

$$\hat{y}_u = f(W' \, z_u + b') \qquad (2.8) \qquad\qquad z_u = h(W^T \, \tilde{y}_u + V_u + b) \qquad (2.9)$$

where $W \in \mathbb{R}^{I \times K}$, $W' \in \mathbb{R}^{I \times K}$, $b \in \mathbb{R}^K$, $b' \in \mathbb{R}^I$ are decoder and encoder weights and biases, $V_u \in \mathbb{R}^K$ is the embedding vector for user u. $h(.)$ and $f(.)$ are activation functions. Hence, $\tilde{y}_u \in \mathbb{R}^I$ is a noise corrupted version of $y_u \in \mathbb{R}^I$, the original preference vector for user u. Typically, the corruption noise in CDAE is chosen as multiplicative mask-out/drop-out noise.

In details, for the training process, CDAE first corrupts input as $\tilde{y}_u$. Then it maps the corrupted input to a latent representation z_u, by computing the sum of the weighted vectors (equation 2.9). At the output layer, z_u is mapped back to original input space to reconstruct the input vector, and obtain the final predictions (equation 2.8).

The model is trained by minimizing the average reconstruction error, composite of two main terms: the prediction error on training data and the regularization term to control the model complexity.

$$\underset{W,\, W',\, V,\, b,\, b'}{argmin} \; \frac{1}{U} \sum_{u \in U} \mathbb{E}_{p(\tilde{y}_u | y_u)} \left[\, l(\tilde{y}_u, y_u) \, \right] + R(W,\, W',\, V,\, b,\, b') \qquad (2.10)$$

where V is the matrix with each column representing a user embedding vector, $l(.)$ is the loss function, and R is the regularizer to prevent overfitting. CDAE model uses squared $L2-norm$.

$$R(.) = \frac{\lambda}{2}(||W||_2^2 + ||W'||_2^2 + ||V||_2^2 + ||b||_2^2 + ||b'||_2^2) \tag{2.11}$$

CDAE uses *Stochastic Gradient Descent (SGD)* to learn the parameters. The SGD attempts to find the minimum by adjusting the configuration of the network after each training sample, this is, update the network weights. *Algorithm 1* provides the detailed procedure.

Optimization. Since the number of output nodes is equal to the number of items, the overall complexity is unsuitable in real world applications. The authors then proposed a *negative sampling technique*. A subset of negative items S_u is sampled from $\bar{O}_u$ and the gradients are computed on the items in $O_u \cup S_u$, instead of on all the items. Being O_u the set of training preferences for user u, $\bar{O}_u$ the items with which the user has not interacted, this is, the candidate items to be recommended to user u.

The gradients for the parameters are then computed as follows:

$$\frac{\partial l}{\partial W_i'} = \frac{\partial l}{\partial \hat{y}_{ui}} \frac{\partial \hat{y}_{ui}}{\partial W_i'} + \lambda\, W_i' \tag{2.12}$$

$$\frac{\partial l}{\partial b_i'} = \frac{\partial l}{\partial \hat{y}_{ui}} \frac{\partial \hat{y}_{ui}}{\partial b_i'} + \lambda\, b_i' \tag{2.13}$$

$$\frac{\partial l}{\partial z_u} = \sum_{i \in O_u \cup S_u} \frac{\partial l}{\partial \hat{y}_{ui}} \frac{\partial \hat{y}_{ui}}{\partial z_u} + \lambda\, z_u \tag{2.14}$$

$$\frac{\partial l}{\partial W_j} = \frac{\partial l}{\partial z_u} \frac{\partial z_u}{\partial W_j} + \lambda\, W_j \tag{2.15}$$

$$\frac{\partial l}{\partial V_u} = \frac{\partial l}{\partial z_u} \frac{\partial V_u}{\partial V_u} + \lambda\, V_u \tag{2.16}$$

$$\frac{\partial l}{\partial b} = \frac{\partial l}{\partial z_u} \frac{\partial z_u}{\partial b} + \lambda\, b \tag{2.17}$$

The gradient descent optimizer is *AdaGrad* which adapts the learning rate to the parameters during the learning procedure. Rather than using a single fixed learning rate, it uses a separate adaptive learning rate η for every parameter θ at every time step t.

Adagrad basically performs smaller updates (i.e. small η) for parameters associated with frequently occurring features, and larger updates (i.e. high η) for parameters associated with infrequent features. For this reason, it is well-suited for dealing with

sparse data.

Then the per-parameter update formula is,

$$\theta^{t+1} = \theta^t - \frac{\eta\, g_\theta^t}{\sqrt{\beta + \sum_{s=1}^{t} g_\theta^s}} \tag{2.18}$$

where β is a smoothing term that avoids division by zero, θ^t is the value of parameter θ at the t^{th} SGD step, and g_θ^t is its gradient at step t.

Algorithm 1 Learning algorithm for CDAE

```
Initialize parameters with random values.
iter ← 0
while iter < maxIter or error on validation set decreases do
    for all u ∈ 𝒰 do
        Sample ỹ_u ∼ p(ỹ_u|y_u) using mask-out/drop-out corruption.
        Compute z_u using Equation 2.9.
        Sample negative samples S_u ⊂ Ō_u.
        for all i ∈ 𝒮_u ∪ O_u do
            Update W_i and b_i using Equations 2.12, 2.13 and 2.18.
        end for
        Compute ∂l/∂z_u using Equation 2.14.
        for all j ∈ {j|j ∈ I and ỹ_ui > 0} do
            Update W_j using Equations 2.15 and 2.18.
        end for
        Update V_u using Equations 2.16 and 2.18.
        Update b using Equations 2.17 and 2.18.
    end for
    iter ← iter + 1
end while
```

CDAE Components

One of the advantages of CDAE model relies on providing flexibility in choosing its components. Different choices for these components correspond to different variants of the model, and then different representation capabilities. The main components include the corruption level, the type of mapping functions and the loss function.

- **Corruption level** q. Typically, the choice of noise used in CDAE is the multiplicative mask-out/drop-out noise. Under this corruption, one randomly overwrites each of the dimensions of x with 0 with a probability of q. As any probability, the corruption probability varies from 0 to 1; $q = 0$ means obtaining results without *any input corruption* (i.e. the non-denoising variant) and $q = 1$ involves a *complete input corruption*.

- **Activation functions** $h(.)$ **and** $g(.)$**.** Having distinct activation functions makes easier to incorporate nonlinearities into the model. There are four possible mapping functions as shown in Table 2.1.

Name	$f(x)$	**gradient** $\frac{\partial f}{\partial x}$
Identity	x	1
Sigmoid	$\sigma(x) = \frac{1}{1+e^{-}x}$	$\sigma(x)(1-\sigma(x))$
Hyperbolic tangent	$tanh(x) = \frac{1-e^{-}2x}{1+e^{-}2x}$	$1 - tanh^2(x)$
Linear	$U^T x$	x

Table 2.1: CDAE mapping functions

  where U is a $K \times K$ transform matrix

- **Loss functions** $l(.)$**.** There are two main possible loss functions used to compare the output values with the original input in CDAE (see table 2.2). The logistic loss function requires the output values to be a value between 0 and 1, so it must be associated with a sigmoid function on the output layer.

Name	$l(.)$
Square	$l^{SL}(y, \hat{y}) = \frac{1}{2}(y - \hat{y})^2$
Logistic	$l^{LL}(y, \hat{y}) = log\ (1 + exp(-y.\hat{y}))$

Table 2.2: CDAE loss functions

Stochastic Gradient Descent

The *Gradient Descent (GD)* is an optimization algorithm used to minimize some function. To find a local minimum, the function "steps" in the direction of the negative of the gradient. It is widely used in various Machine Learning algorithms, most importantly forms the basis of Neural Networks, to find the optimal parameters of the model.

A *batch* is the size of the samples used to compute the gradient in a single iteration. In its basic version, the batch corresponds to the entire data set, therefore it may be infeasible for a huge training data size. Then, a stochastic version of the algorithm is often used instead. The term "stochastic" indicates that the example in each iteration is chosen at random.

The insight is, at each iteration, rather than computing the gradient on the entire dataset, it uses a single randomly selected training sample (i.e. a batch size of 1) to update the set of parameters that minimize the difference between expected value and model output.

Limitations

In CDAE model, the number of parameters of the grows linearly with both the number of users and the number of items, making it more prone to overfitting. It also requires additional optimization to obtain the latent vector V_u for unseen users to make predictions.

Additionally, the neural network structure proposed in CDAE is shallow, so having more prominent structures such as Convolutional Neural Networks (CNN) and Recurrent Neural Networks (RNN), may generate better results. Among other possible extensions to improve the proposed model.

2.4 Evaluation

The evaluation is one of the most crucial step of any model development process. It helps to have a real notion about the predictive ability of a model, and the effectiveness of the model on future (i.e. unseen) data. It is very important to determine an appropriate evaluation metric, otherwise it may lead to the wrong conclusions, being that underestimating or overestimating the approach at question.

2.4.1 Dataset splitting

The most important thing to properly evaluate a model is to not using the entire dataset during the training process. Models should not be evaluated using the same data used to build the model itself, otherwise the error estimate is not a good indicator of performance on future data, it is possible to find patterns even in random data.

In recommender systems, the dataset splitting usually refers to the division of the entries in the ratings matrix. This process is very important to ensure the algorithm is evaluated in a realistic scenario by using test data it has never seen. One of the most common division approaches is that of splitting the dataset into three parts [1]:

- **Training data.** This portion of the data is used to build the model, to learn the parameters by optimizing a loss function.
- **Validation data**. This part of the data is used for model selection and hyperparameters tuning. The latter process is meant to find the optimal model structure. The values of hyperparameters are not adapted by the learning algorithm itself, this is, cannot be directly learned from the data (e.g. the number hidden layers in a NN, etc.).
- **Test data.** Also known as holdout set. It is used to perform a final evaluation of the model. The test set must not be used to adjust the model in any way during the training phase, including the parameter tuning and model selection to prevent overfitting, and prevent the results to be contaminated with knowledge from the testing data [1].

In practice, real data sets are not pre-partitioned into training, validation, and test datasets. To do so, there are several available partitioning techniques, such us *hold-out method* when there is large amount of data available, or *cross-validation* when there is not enough data.

In the hold-out method, the entries of the URM are randomly divided into two sets: training and validation/test (i.e. a hold-out set). The process involves reserving a fraction of the user-item interactions for testing, and the remaining ones are used to build the model.

Cross-validation

Dividing the dataset into a fixed training set and a fixed test set, can lead to an over-specialization to that particular division. For this reason, the splitting and hence, the training process may be repeated several times.

There are several cross-validation techniques. In *k-fold cross-validation*, the entries in the URM are divided into k equal subsets. One of the folds is used for testing, and the remaining $k-1$ folds are used for training. This process is then repeated k times with each of the k subsamples used exactly once as validation data. Finally, the error estimate is computed as the average performance of the k learned models [12].

2.4.2 Recommender Systems Evaluation

The evaluation of a recommender system can be done principally in two ways:

- **Online evaluation**. It attempts to evaluate recommender systems by means of online experiments where real users interact with the system. The idea is to measure the reaction of the user with regard to the presented recommendations [4]. One of the most used online evaluation technique is *A/B testing*. The core idea is to compare the behavior of users who receive recommendations (set A), and those who do not receive recommendations (set B). The need of active users to perform such evaluation is a major drawback and makes it infeasible for research and development purposes.

- **Offline evaluation.** It uses pre-collected data from user-item interactions. Offline experiments are attractive because they do not need interaction with active users, and thus give the possibility to compare a wide range of candidate algorithms at a low cost [12].

  The general purpose of the offline evaluation is to filter out among candidate algorithms to apply to the problem at hand. For the rest of the work, offline evaluation is considered, and thus the metrics corresponding to it. In this category, there are three main approaches to evaluate recommender systems: *predictive accuracy metrics*, *classification accuracy metrics* and *ranking accuracy metrics*.

Predictive Accuracy Metrics

Prediction accuracy is the most important metric in the offline analysis of recommender systems. These metrics are used to evaluate the predictive power of a recommender in terms of how accurate are the estimates. The error represents how different the predicted ratings are from the actual ratings of the users (i.e. ratings in the test set). The difference is the error e_{ui} made by the algorithm [4].

Being T the test set of user-item pairs (u, i) for which we know the true rating, $\hat{r}_{ui}$ the ratings estimated by the recommender system, r_{ui} the true ratings in T, and $|T|$ the number of non-zero interactions in the test set, we have:

- **Mean Absolute Error.** It computes, for each predicted rating, the absolute value of the difference between the actual rating and the estimated rating. The overall average is the MAE.

$$MAE = \frac{1}{|T|} \sum_{(u,i) \in T} |r_{ui} - \hat{r}_{ui}| \tag{2.19}$$

- **Mean Square Error**. MAE is the simplest metric, but it does not take into account the direction of the error (positive error or negative error). The MSE is the average of the squared difference between each true rating of a user and the estimated rating.

$$MSE = \frac{1}{|T|} \sum_{(u,i) \in T} |(r_{ui} - \hat{r}_{ui})^2 \tag{2.20}$$

- **Root Mean Squared Error**. For normalization, the root of the MSE is considered. This technique is widely used since minimizing the MAE is difficult for the algorithm, whereas minimizing the RMSE is relatively easier. And MSE has a larger penalty on large errors and the squared error does not have an intuitive meaning.

$$RMSE = \sqrt{\frac{1}{|T|} \sum_{(u,i) \in T} |(r_{ui} - \hat{r}_{ui})^2} \tag{2.21}$$

Classification Accuracy Metrics

Recommender systems, from an information retrieval point of view, are mainly used to help users discover new and relevant items. The items that are assumed to be relevant are those belonging to the test set. This can be considered as a binary classification problem in which, for every user, the system classifies the items either as relevant or as not relevant [12]. The possible outcomes of the classification can be represented in *confusion matrix* (see table 2.3). This matrix is used to compare the predictions of the model on a set of test data for which the true values are known.

	Recommended	**Not Recommended**
Relevant	True Positive (**TP**)	False Negative (**FN**)
Non-relevant	False Positive (**FP**)	True Negative (**TN**)

Table 2.3: Confusion matrix for recommender systems

Each of the elements in a confusion matrix have an interpretation:

- *True Positive:* items that were recommended that were indeed relevant for the user.
- *False Positive:* items that were recommended but are not relevant for the user.
- *False Negative:* items that were not recommended but should have been.
- *True Negative:* items that were not recommended and that should not have been recommended.

Error metrics for prediction accuracy, such as RMSE, are not suitable to evaluate the top-k recommendation task. Then, the performance of a top-k recommender performance can be measured using alternative metrics. Measuring directly accuracy can be misleading for imbalanced datasets. Then, other possible metrics are:

- **Precision**. The precision measures how many of the recommended items were relevant for the users.
$$precision = \frac{TP}{FN + TP} \tag{2.22}$$

- **Recall (Sensitivity).** The recall measures how many of the relevant items were recommended by the system.
$$recall = \frac{TP}{FP + TP} \tag{2.23}$$

- **F1 measure.** This metric represents a harmonic average of precision and recall.
$$recall = 2\ \frac{recall\ .\ precision}{recall + precision} \tag{2.24}$$

Ranking Accuracy Metrics

Recommender systems are often evaluated using ranking-based measures. Such metrics allow to evaluate the ability of a recommender system to produce a ranked list of items that matches how the user would have ordered the same items. They consider the order of recommendations, and penalizes incorrect suggestions based on that order. This implies that the recommendation problem was stated as a ranking task, where the system presents a list of items to the user.

Ranking metrics use some of the metrics defined above, as precision or recall, but taking into account the ordering of the items. This is, they are computed by considering

only the subset of recommendations from rank 1 to k, so at a given cutoff k (i.e. precision up to the k^{th} item). The Mean Average Precision is considered the most relevant rank metric.

- **Mean Average Precision**. The Average precision (AP) is a ranked precision metric that gives larger credit to correctly recommended items in top ranks [2]. It is applied to a single user at a time. It takes every recommended list of length k, and compares it to the true ranked set of recommendations for that user, as follows:

$$AP@k = \frac{1}{N} \sum_{i=1}^{k} Precision(i) \; . \; Rel(i) \tag{2.25}$$

  where N is the number of relevant items, $Precision(i)$ is the precision at cutoff i in the top-N list, and $Rel(i)$ is an indicator function equaling 1 if the i^{th} recommended item is relevant, zero otherwise. From this definition, the MAP is defined as the mean of the AP scores for all the U users in the test set, this is:

$$MAP@k = \frac{1}{|U|} \sum_{u \in U} AP@k(u) \tag{2.26}$$

  where U represents the set of all users, and $AP@k(u)$ is the average precision at cutoff k computed for user u. The more precise the recommended items are, the higher the MAP.

Chapter 3

Input Corruption Formalization

There are several ways in which noise can be manually injected into the input samples to be provided to a denoising autoencoder. Therefore, the process used and the level of noise have to be chosen. Both decisions have a significant impact on the final representation learned, so, an appropriate corruption process can lead the network to learn higher level representations (i.e. the most relevant features of the data).

Common choices for corruption in denoising autoencoders are mainly stochastic. The main limitations of those traditional techniques include that the noise can be highly dependent on the specific input data, or that there is no noise for some users. The reason is that there is no control over the noise due to the probabilistic approach of the corruption. Now, to counteract these possible undesired scenarios, new ways of introducing corruption to the input are presented next. The classification of input corruption to be considered is shown in figure 3.1.

This chapter is arranged as follows. In section 3.1, the traditional input corruption approaches are described, in particular the masking corruption adopted for collaborative denoising autoencoders. Section 3.2 provides a detailed characterization of the alternative techniques for corrupting inputs proposed in this work.

3.1 Traditional input corruption

A traditional corruption process works as follows: the *corrupted input* $\tilde{x}$ is obtained by an unknown corruption process from a *clean input* x. And after, the DAE must be able to reconstruct the clean input x from its corrupted version $\tilde{x}$ [9].

During training, denoising autoencoders make many passes over the corrupted training samples. Each corrupted input $\tilde{x}$ is typically drawn from a pre-specified conditional noise distribution $p(\tilde{x}|x)$. Each time a training example x is presented, a different corrupted version $\tilde{x}$ of it is generated according to $p(\tilde{x}|x)$, then the noise is introduced at each training iteration.

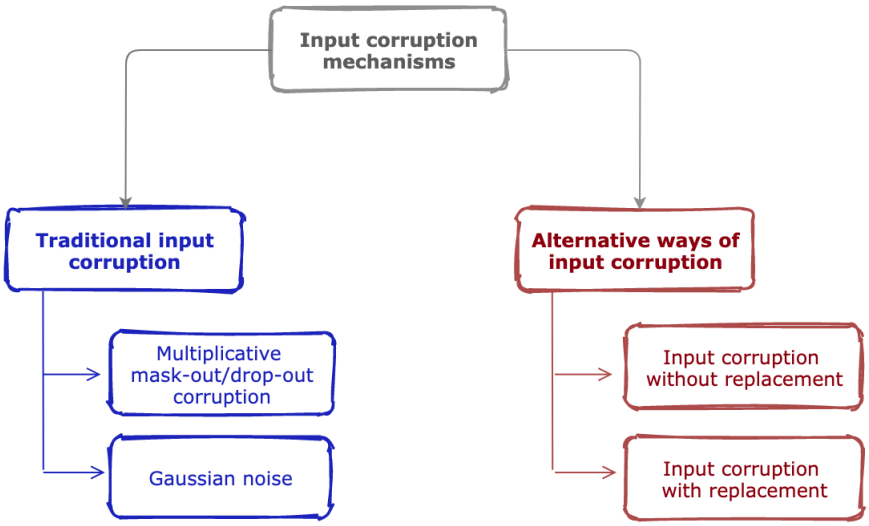

Figure 3.1: Types of input corruption

Common corruption choices are multiplicative mask-out/drop-out noise, which works by masking some of the input values, or additive Gaussian noise [8]. For a binary $x \in [0, 1]$, it is most common to use masking noise, whereas for a continuous training sample x, the Gaussian noise $p(\tilde{x}|x, q) = \mathcal{N}(\tilde{x}|x, q)$ is appropriate.

In either case, the amount of corruption is controlled by a parameter q, and it affects the degree of corruption of the input. If q is high, the inputs are more heavily corrupted during training. On the other hand, masking very few inputs will bias the representation towards local corruptions.

3.1.1 Multiplicative mask-out/drop-out corruption

Under multiplicative mask-out/drop-out corruption, non-zero values are randomly dropped out (i.e. set to zero) independently with a probability of q, while remaining ones are copied to the noised input with probability of $1 - q$. Thus, it applies the stochastic corruption process of randomly masking entries of the input by making them zero.

Typically, this is the type of corruption used in collaborative denoising autoencoders. It works by selecting random interactions from each user profile, and setting them to 0 with some probability q. Remaining ones are kept unchanged. Formally, for each

dimension $i \in 1, 2, ...d$ of the input x, we sample x_i independently as

$$p(\tilde{x_i}|x_i, q) = \begin{cases} 0 & with\ probability\ q, \\ \delta\ x_i & otherwise \end{cases} \tag{3.1}$$

To make the corruption unbiased, set uncorrupted dimensions to δ times their original values, this is:

$$\delta = \frac{1}{1-q} \tag{3.2}$$

In the practice when applying masking corruption to CDAEs, a corrupted copy of the input is created by randomly selecting some of the interactions of the user according to $1-q$. Then the size of the corrupted input will be at most equal to the size of the original input vector times 1 - q as illustrated in figure 3.2. When $q = 0.0$ the input vector suffers no corruption, then the entire input is provided to the network, while setting $q = 1.0$ the empty input (i.e. input size zero) is obtained.

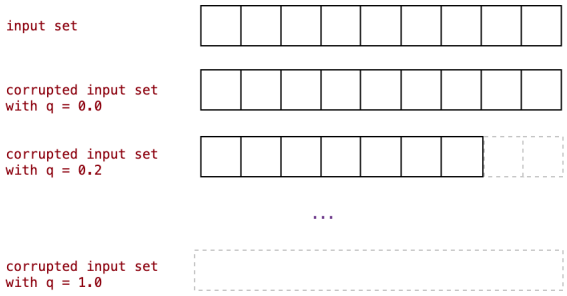

Figure 3.2: Size of the corrupted input set

It is important to notice that the weights connecting the input neurons with the hidden layer W (to be multiplied by the input vector) are not updated during learning and will depend only on its initialization.

3.2 Alternative approaches to input corruption

For the alternative ways of introducing corruption to the input, there is no longer the concept of corruption level. The idea is to remove interactions in a more "controlled"

fashion way, so it is possible to analyze what happens if a randomly selected rating is removed form the original URM, what if two are removed?, and so on.

Overall amount of corruption of the input

In the traditional approaches studied so far as masking or Gaussian noises, the level of noise q is explicitly declared before the corruption process takes place. This corruption level is kept fixed during the training phase of the denoising autoencoder.

In constrast, for the alternative mechanisms to be presented next, the average noise level applied to the input needs to be computed to do proper comparisons with the default corruption used in CDAE. Hence, the degree of corruption of a given input vector $UISC(u)$ can be computed as in equation 3.3. This is, the ratio between the length of an uncorrupted input vector of a given user (i.e. user profile) and the length of the vector obtained after corrupting it.

$$UISC(u) = \frac{|| \textit{ corrupted input set } (u) ||}{|| \textit{ input set } (u) ||} \tag{3.3}$$

where $UISC$ stands for *User Input Set Corruption*. The overall input set corruption is computed as the average percentage of corruption among all the users $u \in U$,

$$\textit{overall input set corruption} = \frac{1}{|U|} \sum_{u \in U} UISC(u) \tag{3.4}$$

The *overall input set corruption* and the input corruption ratio q considered on the traditional approaches are complementary probabilities, so the sum must equal to 1. Therefore, to be able to compare the results obtained with any of the alternative ways of input corruption, it is useful to consider the complement rule for probabilities, as

$$q = 1 - \textit{overall item set corruption} \tag{3.5}$$

3.2.1 Input corruption without replacement

The without replacement corruption works as follows: for each user u, a non-zero interaction r_{ui} given to an item i is randomly selected and removed, and it goes on iteratively as shown in figure 3.3. Afterwards, each corrupted version $\tilde{r}_{ui}$ is used to train the network.

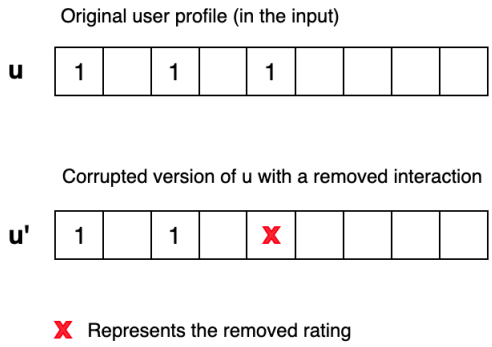

Figure 3.3: Example of a corrupted input without replacement

The main characteristic of this type of corruption is that once a user is selected, and the rating removed, that user is cannot be picked again. A further constraint imposed to avoid introducing any bias on the data by losing the entire profile of a user, is to consider only users with at least two non-zero interactions as candidate for corruption.

Losing the entire profile of a user is possible using mask-out/drop-out approach. It removes all the interactions required to fulfill the level of noise set with q, which means an empty input in the case $q = 1.0$.

The *default input corruption without replacement* removes a single interaction for each user for which the constraints hold (i.e. sufficient non-zero entries). Nonetheless, it is possible to corrupt user profiles by masking more than one interaction, thus to remove n interactions. The n interactions are zeroed-out following an *incremental approach*, one at a time, and at each time the sufficient number of interactions condition is checked. Consequently, it might happen that for a given user the condition is not satisfied and less than n interactions are removed.

Algorithm 2 provides the detailed procedure. In *step 4* there are two available variants for selecting the interaction to be removed for each user: **a)** remove a possible different interaction r_{ui} at each iteration, and **b)** remove the same interaction r_{ui} in all the iterations. These two cases are aimed to take into account two scenarios, presenting a different corrupted version of a input to the network at each training iteration, as done in the traditional approaches, and having an identical corruption iteration-wise.

Algorithm 2 Input corruption without replacement

1: **for all** $u \in \mathcal{U}$ **do**
2: Get R_u containing all interactions r_{ui} for u (i.e. user profile)
3: **if** $|R_u| \geq 2$ **then**
4: Select a random non-zero interaction (a rating $r_{ui} > 0$) from R_u
5: Remove r_{ui} from R_u
6: **end if**
7: **return** R_u (corrupted profile)
8: **end for**

3.2.2 Alternative approach to without replacement corruption

It is possible to use a slightly modified version of the default without replacement corruption, in which instead of removing a fixed number of interactions from each user profile, a portion of the interactions is removed by setting a value for the corruption ratio p from 0.0 to 1.0. The constraint on the minimum available interactions by user still hold, thus the user must have at least two interactions to continue incrementally the removal.

In this case, as in traditional approaches as mask-out/drop-out, the number removed interactions depends on the size of the original user profile (i.e. input size). Thus, if the user profile contains 32 interactions with items and the ratio of corruption is set to 0.1 (i.e. remove 10% of the ratings), the corrupted input vector for that user will have 3 interactions less.

3.2.3 Input corruption with replacement

The last non-traditional approach proposed is the *with replacement corruption*. For each original user profile, several corrupted copies c are generated. A user u is selected at a time, it is replicated c times, each of them having the same interactions of u, but with one interaction less as shown in figure 3.4. Accordingly, every replica is treated as a new user to be independently used for training the network.

It is important to consider that the ratings are always selected from the original user profile, because the corruption is with replacement. More precisely, for every corrupted replica the interaction to be removed is chosen from the complete user profile. Indeed, the interaction removed for each copy may even coincide since it is selected at random.

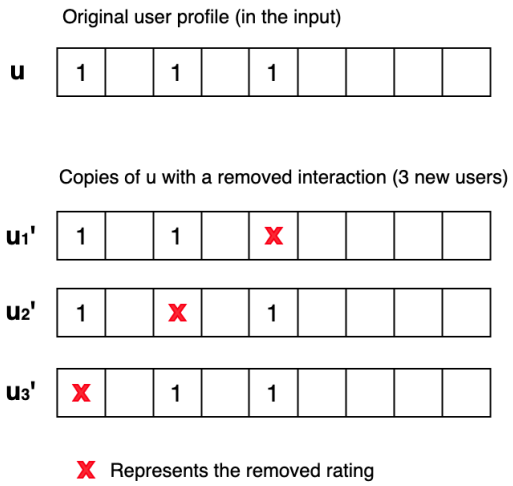

Figure 3.4: Example of a corrupted input with replacement

The purpose is not only to learn from explicitly corrupted inputs, but to enlarge the number of training samples. This is done by generating additional corrupted copies of each user profile whenever the input corruption is performed. The concept to be exploited is data augmentation for improving deep learning.

Data augmentation. A frequent problem in the field of machine learning is the lack of sufficient amount of the training data. One of the ways of dealing with this problem is to adopt data augmentation techniques to artificially add training data so as to make the resulting model more robust, and improve generalization to unseen data, specially when the models are data-hungry.

However, these advantages come at a price in terms of training time and increased computational demands. In DAEs, each data sample (i.e. being a user profile) must be corrupted many times and passed through the learner, so it may present a serious challenge for high-dimensional inputs. With replacement corruption then can effectively increase the training set size in such a way that the number of samples will be larger than the number of users.

In this proposed approach the replicas are generated online, but we are not actually adding the new users to the original URM, they are generated in the way just before

being presented to the network, used for training, and then the new users are not stored in any way which can alleviate the drawback described above.

The detailed procedure is in algorithm 3. Even if is not explicitly stated, the without replacement corruption to be applied at step 4 can be performed with a different value for n, this is, remove more interactions for each replica.

Algorithm 3 Input corruption with replacement

1: **for all** $u \in \mathcal{U}$ **do**
2: Get R_u containing all interactions r_{ui} for u (i.e. user profile)
3: **for** $i = 1$ to c **do**
4: Get a corrupted user profile $R_u{}'$ applying without replacement corruption with $n = 1$ to remove an interaction r_{ui}.
5: Assign R'_u to a new user u' having the same interactions but without r_{ui}
.
6: Train the network on the new user u'.
7: **end for**
8: **end for**

Chapter 4

Experimental results

This chapter is devoted to present the set of experiments that aim to answer the research questions, along with a discussion around the results. It is organized as follows. In section 4.1 some details on the benchmark datasets used for the experiments as well as each step in the data preprocessing are described followed by a description of the general experimental setup to be adopted in section 4.2. The next two sections 4.3 and 4.4 contain the results of the experiments devoted to answer the questions above are reported together with some comments.

4.1 Datasets

To conduct the experiments different small to medium-scale common datasets for evaluating performance of recommender systems were used. The chosen datasets are MovieLens-100K[1], MovieLens-1M[2], MovieLens-10M[3] were considered. Table 4.1 summarizes their properties.

Dataset	# users	# items	# ratings	Sparsity
Movielens-100K	943	1682	100.000	93.70%
Movielens-1M	6040	3706	1'000.209	95.53%
Movielens-10M	69878	10677	10'000.054	98.66%

Table 4.1: Datasets statistics

Preprocessing. The datasets to be used provide explicit feedbacks, where each rating represents the user-item interaction as an integer ranging from 1 to 5. However, CDAE works under implicit user preferences. Then, in all of the datasets, the explicit users' feedback was transformed into implicit by using the following criteria: ratings with more than four stars are kept, and the remaining ones are treated as missing entries. This is, ratings with 1 to 3 stars are transformed into zeros, and the 4 and 5's into ones.

[1] https://grouplens.org/datasets/movielens/100k/
[2] https://grouplens.org/datasets/movielens/1m/
[3] https://grouplens.org/datasets/movielens/10m/

This processing method is widely used on recommendation with implicit feedback and is the binarization principle proposed by the authors.

Dataset splitting. The train-test splitting of the data was performed using a 80-20% split. The division of the interactions is done per user, in the sense that for each user, 80% of the data is randomly selected for the train set used to learn from, and the 20% remaining is used to test out the goodness of the model.

An additional partition of the training set is usually performed to find the best configuration of the model by applying hyperparameter tuning. However, since the authors argued that the parameters used to control the learning process (e.g. learning rate, regularizer penalty, etc.) were chosen by performing 5-fold cross validation, this additional split was skipped.

4.2 Experimental Setup

Detailed experiments were conducted the publicly available source code provided by the authors, making some assumptions on parameter settings not stated in the original paper or in the code considering the poor documentation. Those parameters explicitly stated in the implementation details are kept as suggested, aiming to replicate the same scenarios, run the experiments and be able to make proper comparisons.

- For training, Stochastic Gradient Descent was used for a maximum of 50 iterations. The optimizer adopted is AdaGrad with a shrinkage term $\beta = 1$, and a learning rate $\eta = 0.1$. In the paper, several values were suggested but in practice only 0.1 was used.

- Besides, the negative sampling technique is the same used in the original paper, having $NS = 5$. Then, for each user, the number of negative samples is 5 times the number of observed ratings of this user.

- Also the number of latent dimensions (i.e. neurons in the hidden layer) was set to 50 as in the paper, so $K = 50$.

- For the evaluation, ranking-based metric MAP@N was used, being the most suitable for the top-N task. For the experiments two different cutoff points are considered $N \in \{5, 10\}$. Mainly, for each user, it compares the predicted rank of the held-out items with their true rank.

- Different choices for CDAE components like mapping functions (on both layers) and the loss function lead to different model configurations. The notation for the model variants was adopted from the original paper, this is M1, M2, M3 and M4 as described in Table 4.2. But for our experiments, only two variants out of them were studied in the three datasets, M1 and M3.
- Each variant was trained under varying corruption levels $q \in \{0, 0.2, 0.4, 0.6, 0.8, 1\}$. The main focus is to consider the two different extremes of corruption level. $q = 0$ (no input corruption) and $q = 1$ (complete input corruption).
- Finally, regularization was performed using L2-norm and a generic $\lambda = 0.01$ since there is no explicitly stated value.

	Hidden Layer $h(.)$	**Output Layer** $f(.)$	**Loss function** $l(.)$
M1	Identity	Identity	Square
M2	Identity	Sigmoid	Logistic
M3	Sigmoid	Identity	Square
M4	Sigmoid	Sigmoid	Logistic

Table 4.2: Four variants of the CDAE model

4.3 Model performance comparison

4.3.1 Experiments

For this set of experiments, a lot of assumptions were done on the actual configuration needed to properly run the experiments on the code. This first section, is mainly devoted to understand CDAE performance under various settings proposed by the authors to check whether the results in the paper are correct or not.

It is noteworthy to mention that the MAP scores in the original paper are not precise because are presented using charts without any annotation, and even if there is an additional table, not all corresponding values are presented. So, there are few values which can be used for the comparisons. The *expected results* are those reported in the original paper.

- **Comparison with DAE.** Comparable experiments are conducted including user input node (CDAE) or not (DAE). It measures the repercussion of adding user-specific information compared to standard DAE methods.

- **Effects of using tied weights.** To study the impact of sharing weight matrices between encoder and decoder layers (TW), over learning separate weights (NTW). Since the encoder and decoder in DAE mirror each other in structure, they can be forced to learn one set of weights. When using tied weights, the decode weights are the transpose of the encode weights (i.e. by $W' = W^T$).

  The motivation behind sharing weights is to reduce the number of parameters and speed up the learning process. On the other hand, having untied weights is useful to retain more contextual information into the bottleneck layer. In general, the latter leads to better performance in practice, but still depends on the model at hand.

- **Impact of the denoising technique**. A last set of experiments was used to understand the behavior of the original corruption model implemented in CDAE under varying levels of corruption. Mainly to consider the two extremes of corruption, the non-denoising scenario by having no input corruption (i.e. $q = 0.0$), and the complete input corruption scenario with (i.e. $q = 1.0$).

Due to the way in which the CDAE was implemented, the experiments on *comparison with DAE* and *effects of using tied weights* are somehow related. The default CDAE uses untied weights, then CDAE and NTW configurations are equivalent, and the results of their corresponding experiments must coincide.

The setup for each of the experiments is shown in table 4.3. Where *asymmetric* refers to having an asymmetric DAE, meaning the weights between the encoder and the decoder are untied to retain more contextual information into the bottleneck layer, and *user factor* to include or not the additional user input node.

Experiment	**asymmetric**	**user factor**
Including user input node (CDAE)	true	true
Without user input node (DAE)	true	false
Using untied weights (NTW)	true	true
Using tied weights (TW)	false	true

Table 4.3: Experimental setup for the model performance comparison

Expected results		
q	**MAP@5**	**MAP@10**
0.0	0.1730	0.1402

Experimental results		
q	**MAP@5**	**MAP@10**
0.0	**0.16711**	**0.13842**
0.2	0.1996	0.1612
0.4	0.2142	0.1720
0.6	0.2313	0.18678
0.8	0.2045	0.1650
1.0	0.2582	0.2258

(a)

Expected results		
q	**MAP@5**	**MAP@10**
0.0	0.1619	0.1270

Experimental results		
q	**MAP@5**	**MAP@10**
0.0	0.1481	0.1168
0.2	0.1888	0.1517
0.4	0.2005	0.1619
0.6	0.2141	0.1726
0.8	0.16128	0.1295
1.0	0.1225	0.1046

(b)

Table 4.4: CDAE (a) and DAE (b) performance on MovieLens-10M using model M1. The results that best fit the expected values are indicated in **bold**.

Expected results		
q	**MAP@5**	**MAP@10**
0.0	0.2099	0.1722

Experimental results		
q	**MAP@5**	**MAP@10**
0.0	0.2623	0.2177
0.2	0.2692	0.2231
0.4	0.2743	0.2294
0.6	0.2821	0.2388
0.8	0.2950	0.2486
1.0	0.2963	0.2432

(a)

Expected results		
q	**MAP@5**	**MAP@10**
0.0	0.1953	0.1579

Experimental results		
q	**MAP@5**	**MAP@10**
0.0	0.2432	0.2060
0.2	0.2371	0.1989
0.4	0.2669	0.2284
0.6	0.2611	0.2247
0.8	0.1799	0.1523
1.0	0.1211	0.0963

(b)

Table 4.5: CDAE (a) and DAE (b) performance on MovieLens-10M using model M3

4.3.2 Results

Comparison with DAE

The results obtained using CDAE with variant M1 without any noise (i.e. $q = 0.0$) are very close to the expected values. However, if we consider having larger corruption levels, the results described are smaller than 0.23 for MAP@5 and 0.17 for MAP@10 which does not comply with the observed values. For instance, in the case of $q = 1.0$, provides the

best results for both metrics, and all the possible values in the paper. For DAE using M1, the closest values are obtained with a corruption level of 0.8, which does not correspond to the results of $q = 0.0$.

Regarding model M3, if we compare the empirical results on table 4.5 against the reported there is no correspondence. In the chart presented on [13], MAP@5 and MAP@10 have a downwards trend as values for q are increasing. Whereas in the experiments the values are not just increasing with q, but also are much better.

Besides, the experiments help to understand somehow the effects of introducing nonlinearities as in model M3. Even if the results are different from the reported ones, it still outperforms M1 which has identity for both activation functions.

Effects of using tied weights.

Expected results		
q	**MAP@5**	**MAP@10**
0.0	0.0983	0.0763

Experimental results		
q	**MAP@5**	**MAP@10**
0.0	0.0367	0.0286
0.2	0.0389	0.0307
0.4	0.0283	0.0230
0.6	0.0271	0.0233
0.8	0.1007	0.0850
1.0	0.2585	0.2258

(a)

Expected results		
q	**MAP@5**	**MAP@10**
0.0	0.2044	0.1669

Experimental results		
q	**MAP@5**	**MAP@10**
0.0	0.2533	0.2173
0.2	0.2560	0.2191
0.4	0.2587	0.2216
0.6	0.2630	0.2253
0.8	0.1923	0.1627
1.0	0.2561	0.2171

(b)

Table 4.6: CDAE performance using TW on MovieLens-10M using: (a) model M1; (b) model M3

As it was stated before, the results obtained using untied weights (NTW) are identical to those obtained with the standard CDAE, therefore, we can compare results obtained using tied weights (TW) with those in tables 4.4 and 4.5 (a).

Independently of the model variant results of NTW are much better than TW for all the tested corruption levels. For M1 except for the complete input corruption scenario in which setting q = 0.0 can improve the results in a notoriously, TW are significantly

worse than NTW.

Now, despite of having better results with variant M3, the best MAP scores are from models with NTW. The outstanding performance of variant M3 even under TW conditions may be due to the intrinsic nonlinearities in the model, which appears beneficial.

Impact of the denoising technique

In the experiments described above, the corruption level determines the amount of noise introduced in the input, thus, can be used to measure the impact of the denoising technique in the performance of a standard CDAE. It reflects the denoising capability in reconstruction process. The results obtained using NTW are preferred for this analysis, which is treated as the default CDAE setting with respect to weights from now on.

Comparing the results obtained using the two extreme corruption levels: $q = 0$ meaning there is no input corruption, and $q = 1$ the maximum corruption (i.e. empty input scenario). For both variants, the higher the corruption level, the better the MAP scores. Model M1, using $q = 0$ performs worse that when $q = 1$ as expected from the original paper. For model M3, not only increasing the level of noise improves the results, but those results are larger than for M1 for all q which may be due to the use of nonlinear mapping functions.

The results of CDAE, under any of the analyzed settings, are always better for the maximum corruption level $q = 1.0$. This indicates that simply summing up all the input vectors ($q = 0.0$) is insufficient for learning good representations, and is in fact worse than dropping out all of them ($q = 1.0$).

4.4 Performance of different input corruption strategies

4.4.1 Experiments

Between models M1 and M3, M1 seems to present the most consistent results. Yet cannot obtain exactly the same performance in the paper but the difference is smaller than with M3. We can assume that this means the coding CDAE setting was done properly, and it can be used to test the new corruption mechanisms with a smaller level of uncertainty.

Even if variant M1 is not the best model presented by the authors, is the one we were able to replicate closely.

In each of the experiments, the corruption level q is shown to do proper comparisons with the *multiplicative mask-out/drop-out noise* as baseline. For alternative approaches q was computed. The experiments are meant to evaluate the following types of input corruption:

(a) **Multiplicative mask-out/drop-out corruption**. The model will be trained under varying corruption levels $q \in \{0.0, 0.2, 0.4, 0.6, 0.8, 1.0\}$. The corruption level represents the percentage of the interactions that will be set to zero, thus removed.

(b) **Default without replacement corruption**. The experiments will measure the impact of varying the number of removed interactions by user when applying the corruption procedure. The number of removed interactions to be examined from $n \in \{1, 2, 5, 10, 20\}$ for both variants, by removing a) a possibly different interaction at each iteration or b) the same interaction in all the iterations.

(c) **Alternative without replacement corruption.** In this second approach, instead of removing a fixed number of interactions, a percentage is dismiss. The corruption ratio is evaluated from $p \in \{0.1, 0.2, 0.5, 0.9\}$

(d) **With replacement corruption**. The idea is to evaluate the impact of training the network using more than one corrupted version of each user profile. The number of corrupted replicas generated for each user is $c \in \{2, 10, 20\}$. For this particular set of experiments, also n needs to be selected as with replacement corruption basically repeats c times the default without approach. Only the without replacement corruption *variant a)* is considered under $n \in \{1, 2, 5, 10\}$ as it achieves better results.

(e) **With replacement corruption using alternative without replacement**. These last experiments use the ratio-oriented corruption in the without replacement scheme. As previously, the corruption ratios to be evaluated are $p \in \{0.1, 0.2, 0.5, 0.9\}$ and the number of corrupted copies $c \in \{2, 10, 20\}$.

More datasets. Further experiments were conducted on other benchmark datasets, to check how the model behaves in the face of new datasets like MovieLens-1M and MovieLens-10M. The idea would be to compare the original corruption (a) with the non-traditional corruption that provides the best results (d) in these datasets.

Training speed. Besides the final MAP scores, the different types of input corruption can be compared by means of the training speed. Thus, compare how long does it take to train variant M1 using the default model settings for a maximum of 50 iterations using both mechanisms (a) and (d).

4.4.2 Results

Experiment 1		
q	**MAP@5**	**MAP@10**
0.0	0.2214	0.1883
0.2	0.2584	0.2153
0.4	**0.2589**	**0.2147**
0.6	0.2490	0.2048
0.8	0.2342	0.1880
1.0	0.2548	0.2138

Experiment 2		
q	**MAP@5**	**MAP@10**
0.0	0.2214	0.1883
0.2	0.2551	0.2145
0.4	**0.2557**	**0.2154**
0.6	0.2425	0.2032
0.8	0.2367	0.1922
1.0	0.2548	0.2138

(a) Mask-out/drop-out corruption

n	**avg % corrupt.**	**q**	**MAP@5**	**MAP@10**	**MAP@5**	**MAP@10**
1	0.9759	0.0240	**0.2271**	**0.1857**	**0.2209**	**0.1844**
2	0.9519	0.0480	0.2196	0.1813	0.2026	0.1668
5	0.8797	0.1202	0.2119	0.1736	0.1866	0.1557
10	0.7595	0.2404	0.2047	0.1690	0.1586	0.1312
20	0.5441	0.4558	0.2089	0.1761	0.1239	0.1004
			variant a)		*variant b)*	

(b) Default without replacement corruption

p	**avg % corrupt.**	**q**	**MAP@5**	**MAP@10**
0.1	0.911	0.089	0.2387	0.2001
0.2	0.8009	0.1907	**0.2594**	**0.2188**
0.5	0.5005	0.4950	0.2449	0.2084
0.9	0.1095	0.8904	0.2009	0.1714

(c) Alternative without replacement corruption removing different interactions

n	q	c	MAP@5	MAP@10
1	0.0240	2	0.2271	0.1857
1	0.0240	10	**0.2816**	**0.2336**
1	0.0240	20	0.2782	0.2250
2	0.0480	2	0.2196	0.1813
2	0.0480	10	**0.2839**	**0.2355**
2	0.0480	20	0.2713	0.2266

n	q	c	MAP@5	MAP@10
5	0.1202	2	0.2119	0.1736
5	0.1202	10	**0.2905**	**0.2425**
5	0.1202	20	0.2731	0.2218
10	0.2404	2	0.2047	0.1689
10	0.2404	10	**0.2836**	**0.2382**
10	0.2404	20	0.2674	0.2213

(d) With replacement corruption using default without replacement variant a)

p	q	c	MAP@5	MAP@10
0.1	0.0890	2	0.2387	0.2001
0.1	0.0890	10	**0.2626**	**0.2204**
0.1	0.0890	20	0.2417	0.1980
0.2	0.1907	2	0.2525	0.2129
0.2	0.1907	10	**0.2577**	**0.2161**
0.2	0.1907	20	0.2267	0.2185

p	q	c	MAP@5	MAP@10
0.5	0.4950	2	**0.2448**	**0.2084**
0.5	0.4950	10	0.2397	0.1976
0.5	0.4950	20	0.2292	0.1886
0.9	0.8904	2	**0.2009**	**0.1714**
0.9	0.8904	10	0.1565	0.1336
0.9	0.8904	20	0.1304	0.1233

(e) With replacement corruption using alternative without replacement

Table 4.7: Different input corruption techniques in MovieLens-100K using model M1. Best results in each scenario are indicated in **bold**.

Multiplicative mask-out/drop-out corruption

For *mask-out/drop-out corruption*, two different experiments were conducted as shown in table 4.7 (a). They are just a couple runs of M1 which do not differ in terms of model setup. The purpose was to analyze the impact in performance when levels of noise q are set between 0.2 and 0.8. The percentage difference between the corresponding values in both experiments are due to the intrinsic randomness of the corruption, and are very small. For instance, MAP@5 using $q = 0.2$ change was of $0,0530\%$.

On the other hand, the results obtained for the extreme levels of corruption 0.0 and 1.0 are identical, which makes perfectly sense considering that there is no randomness involved, there is no corruption or complete corruption. The best scenario for this type of corruption is $q = 0.4$, for both experiments. But the results obtained with the other corruption levels are not so different.

Default without replacement corruption

The users are supposed to have at least two interactions in the training set under the 80-20% data splitting. Then in the case of $n = 1$ (i.e. removing one interaction by user), the performance of *without replacement corruption* can be directly compared with the results obtained applying the baseline corruption with $q = 0.0$. Contrary, for values $n > 1$ this not necessarily holds, meaning that even if the overall percentage of corruption is close to any of the possible values of q, it may happen that users have few interactions, so the removed interactions for those users might be less than n.

However, for purposes of this analysis, the comparisons using the corruption level q computed for the alternative approaches will be directly compared to the q selected for the experiments using *mask-out/drop-out corruption*.

The results obtained for both variants of *without replacement corruption* are reported in 4.7 (b). The performance achieved removing a different interaction at each iteration (variant a) always produces better results. Thus, from now on the results presented will only contemplate variant a) for without corruption.

Having one or two interactions less by user, can be compared to having $q = 0$. For $n = 1$, results are at least as good as the baseline. But this performance tends to drop as n increases, for instance when $n = 10$, which corresponds to an overall corruption level of $q = 0.2$, the values obtained are almost 25% lower.

Alternative without replacement corruption

The results presented for *alternative without replacement corruption* only consider the scenario in which different interactions were removed throughout the iterations (equivalent to variant a) in the default mechanism). Those corresponding to the other variant (same iteration is zeroed-out at each iteration) were omitted since did not provide any performance gain.

For low corruption ratios like $p = 0.1$ or $p = 0.2$, this mechanism provides better outcomes not only with respect to default without replacement, but it even goes beyond

mask-out/drop-out baseline. This has not been the case for larger corruption ratios like 0.9 in which the results are of inferior quality. Generally, the smaller p is, the better the improvement of this corruption mechanisms against the other approaches analyzed so far.

With replacement corruption

For *with replacement corruption*, two parameters need to be specified, the number of corrupted replicas to be generated for each user c as well as the amount of interactions removed in each of them n.

Another thing to keep in mind is that since every replica is generated by applying without replacement corruption several times, the variant to be adopted is a) so as to generate mostly different replicas by user. Otherwise, if we consider removing always the same interaction, all the corrupted replicas of each given user would be identical, losing the opportunity of learning from distinct user profiles. Even though, experiments considering the variant b) were conducted but the results were very low in comparison to variant a), then they are omitted in the results table.

There are three scenarios to be evaluated, taking into account the number of replicas by user to be generated for training the model, this is $c = 2$, $c = 10$ and $c = 20$. For two corrupted replicas, the larger the number of interactions removed n, the lower the performance. Indeed, the best performance for $c = 2$ is attained for the minimum amount of masked interactions $n = 1$; for the largest $n = 10$, the results are more or less 10% reduced.

The contrary occurs for $c = 10$. In which there is a continuous enhancement up to five removed ratings, $n = 5$. In fact, this setting produces the best results up to this point, a MAP@5 and MAP@10 of 0.2905 and 0.2325, respectively. Moreover, it is better-performing than the other two approaches, mask-out/drop-out and default without replacement, in their best possible scenarios, $q = 0.4$ for the former and $n = 1$ for the latter. For values $n > 5$, the performance starts to decrease at a slow speed as in the case of $n = 10$. In general, as the number of removed interactions n grows, the results get lower.

Finally, having twenty corrupted replicas (i.e. $c = 20$), drops a bit the performance

with respect to $c = 10$, but still better than the other approaches, specially when one interaction is discarded at each generated replica.

With replacement corruption using alternative without replacement

Here the without replacement corruption to be applied receives a portion of the interactions to be removed from each user profile, instead of a specific number of them.

For small levels of corruption like $p = 0.1$ and $p = 0.2$, this method works similarly to the previous approach (d). This is, for a number of replicas $c = 2$, it produces better outcomes than the baseline for its comparable scenario $q = 0.0$. Then, the best scores are obtained when $c = 10$. Yet do not improve regarding those previously obtained. Lastly, the performance reduces when the corrupted copies are higher as for $c = 20$. Instead, for larger corruption ratios like $p = 0.5$ and $p = 0.9$, the more the corrupted replicas, the larger the results.

Additional datasets

Complementary experiments were conducted on two more datasets to validate the discussion up to this point. Sometimes the results are tied to the specific dataset into consideration.

q	MAP@5	MAP@10
0.0	0.2529	0.2052
0.2	0.2575	0.2039
0.4	0.2501	0.1966
0.6	0.2405	0.1876
0.8	0.2589	0.1704
1.0	**0.2806**	**0.2289**

(a) Mask-out/drop-out corruption

n	q	c	MAP@5	MAP@10
1	0.0180	2	0.2599	0.2091
1	0.0180	10	0.2778	0.2229
1	0.0180	20	**0.2904**	**0.2376**
2	0.0370	2	0.2596	0.2093
2	0.0370	10	0.2779	0.2243
2	0.0370	20	**0.29063**	**0.23833**

n	q	c	MAP@5	MAP@10
5	0.0940	2	0.2597	0.2082
5	0.0940	10	0.2832	0.2303
5	0.0940	20	**0.2934**	**0.2401**
10	0.1880	2	0.2542	0.1862
10	0.1880	10	0.2803	0.2282
10	0.1880	20	**0.2895**	**0.2394**

(d) With replacement corruption using default without replacement variant a)

Table 4.8: Different input corruption techniques in MovieLens-1M using model M1

q	MAP@5	MAP@10
0.0	0.1671	0.1384
0.2	0.19956	0.1612
0.4	0.21417	0.1720
0.6	0.23127	0.1868
0.8	0.20454	0.1649
1.0	**0.25816**	**0.2258**

(a) Mask-out/drop-out corruption

n	q	c	MAP@5	MAP@10
1	0.0226	2	0.1760	0.1444
1	0.0226	10	0.2535	0.2195
1	0.0226	20	**0.2549**	**0.2232**
2	0.0450	2	0.1641	0.1354
2	0.0450	10	0.2504	0.2163
2	0.0450	20	**0.2577**	**0.2208**

n	q	c	MAP@5	MAP@10
5	0.1131	2	0.1611	0.1359
5	0.1131	10	0.2449	0.2126
5	0.1131	20	**0.2494**	**0.2199**
10	0.2263	2	0.1619	0.1345
10	0.2263	10	0.2499	0.2099
10	0.2263	20	**0.2487**	**0.2101**

(d) With replacement corruption using default without replacement variant a)

Table 4.9: Different input corruption techniques in MovieLens-10M using model M1

For Movielens 1M, which is larger than the one analyzed before, *mask-out/drop-out corruption* behaves different in terms of the best-performing corruption level, being $q = 1.0$; but in general, the results varying q are not so distant.

Using *with replacement corruption removing a fixed number of interactions (d)* behaves similarly for this dataset. It outperforms the mask-out/drop-out corruption in all the experiments. When the number of removed interactions n increase up to 5, the results are comparable to a corruption level of $q = 0.0$, and when $n = 10$ to a level of noise $q = 0.2$.

Being the best model configuration for this dataset having a number of replicas $c = 20$ despite of the number of removed interactions on each of them. A small drop in the performance occur when the number of removed ratings is larger (i.e. $n = 10$), but still the results are very close.

Similarly, for Movielens-10M dataset, the alternative approach outperforms the baseline, proving to be more effective when there are twenty corrupted copies used for training the network. Besides, the differences in the MAP scores are quite small.

Training speed

- **MovieLens-100K**

q	Time(sec)
0.0	35
0.2	25
0.4	25
0.6	24
0.8	24
1.0	22

(a) Mask-out/drop-out corruption

n	q	c	Time(sec)
1	0.0226	2	25
1	0.0226	10	172
1	0.0226	20	359
2	0.0450	2	27
2	0.0450	10	186
2	0.0450	20	385

n	q	c	Time(sec)
5	0.1131	2	32
5	0.1131	10	226
5	0.1131	20	484
10	0.2263	2	38
10	0.2263	10	293
10	0.2263	20	641

(d) With replacement corruption using default without replacement variant a)

Table 4.10: Execution of the different input corruption techniques in MovieLens-100K

- **MovieLens-1M**

q	Time(sec)
0.0	322
0.2	330
0.4	317
0.6	286
0.8	282
1.0	252

(a) Mask-out/drop-out corruption

n	q	c	Time(sec)
1	0.0226	2	347
1	0.0226	10	2186
1	0.0226	20	3817
2	0.0450	2	306
2	0.0450	10	2052
2	0.0450	20	4362

n	q	c	Time(sec)
5	0.1131	2	352
5	0.1131	10	2932
5	0.1131	20	5797
10	0.2263	2	498
10	0.2263	10	3792
10	0.2263	20	6833

(d) With replacement corruption using default without replacement variant a)

Table 4.11: Execution of the different input corruption techniques in MovieLens-1M

Lastly, the training times (in seconds) for the *mask-out/drop-out* baseline corruption mechanism, and the best performing alternative *with replacement* removing a fixed number of ratings on each user replica are reported in tables 4.10 and 4.11. The corresponding times for movielens 10M dataset, are not reported but the analysis next also holds.

Mostly, for mask-out/drop-out corruption, the larger the level of corruption q, the lower the time it takes for the model to train. On the other hand, for the alternative, two parameters are considered during the corruption process, the number of replicas and the number of removed interactions on each of them, and either have an impact in learning time.

Having few replicas by user, with few removed interactions each (i.e. $c = 2$ for $n = 1$) takes less than the baseline in its comparable scenario of a corruption level $q = 0.0$. However, as the number of replicas increases and also the removed ratings, the time enlarges.

For both datasets, the best scenarios correspond to having twenty replicas by user, which indeed corresponds to the largest time, almost 10x the baseline, but the performance enhancement is significant. Then, even if the time for training the model using the alternative approaches reduces the training speed, it is worth in terms of performance.

Chapter 5

Conclusions

In this thesis, a comprehensive set of experiments were conducted aiming to answer two key reseach questions: whether we were able to replicate the experiments described in the paper [13] concerning collaborative denoising (CDAE) autoencoders, while achieving comparable results, and if it is possible to find alternative ways in which noise can be introduced in the input provided to a CDAE so as to obtain better results with state-of-the-art noise strategies.

The experiments in the paper were reproduced without achieving the same results described in the paper. In fact, the empirical results are much better than those reported, being the complete input corruption the best-performing scenario. The differences may be due to some assumptions on the values for parameters in the code, bugs in the implementation or even intrinsic randomness in the implemented model.

Further reasoning can be done on the observed results. Including the user-specific vectors consistently outperforms the denoising autoencoder as concluded by the authors. This indicates that adding a user node into the encoding process has a positive impact in the final performance. Besides, the authors recommended to avoid using tied weights for CDAE which is consistent with the results. And adding noise on the input consistently improves the performance, which means that the denoising technique can help with learning more robust representations.

After all the corrupting mechanisms were assessed, the alternative ways to introduce corruption into the input did perform at least as well as the traditional mask-out/drop-out approach used by the standard collaborative denoising autoencoder. The improvement in performance may vary according to the dataset, but there is a consistent enhancement despite of the size and other statistics of the datasets.

In a default without replacement corruption scenario, the results are near enough to those obtained using the masking noise. However, as the number of removed interactions increase, the performance reduces considerably. The reason behind this loss in terms of performance may be that for each user, the number of removed interactions can be greater than or smaller than the average, meaning that it may happen to have users with almost all interactions removed restricting the item correlation patterns originally present in the data.

On the other hand, the with replacement corruption, where a fixed number of ratings removed for each user replica, consistently outperforms across datasets and metrics, the other corruption mechanisms. For this scenario, independently from the amount of removed interactions, the number of replicas to be considered can vary up to a certain value before the performance gets compromised. There should be a trade-off between those two parameters: too many corrupted replicas by user may counteract the gain obtained by artificially adding instances to the training data, this is, corrupted copies of each user profile.

It can be concluded that, although the results obtained from the experiments are not exactly those reported by the authors, for this work the source CDAE implementation was extended so as to be flexible in terms of the corruption mechanism to be used for manually injecting noise into the input, and produce comparable results, being the with replacement technique the one providing the most significant improvements.

Bibliography

[1] Charu C. Aggarwal. *Recommender systems. Vol.1.* Reading, Massachusetts: Cham: Springer International Publishing, 2016. ISBN: 978-3-319-29659-3.

[2] Manas Bajaj et al. "SLIM: collaborative model-based systems engineering workspace for next-generation complex systems". In: *Aerospace Conference. IEEE* (2011), pp. 1–15. DOI: 10.1109/AERO.2011.5747539.

[3] Zeynep Batmaz et al. "A review on deep learning for recommender systems: challenges and remedies". In: *Artificial Intelligence Review* 52.1 (2019), pp. 1–37. DOI: 10.1007/s10462-018-9654-y.

[4] Mingang Chen and Pan Liu. "Performance Evaluation of Recommender Systems". In: *International Journal of Performability Engineering* 13.8 (2017), pp. 1246–1256. DOI: 10.23940/ijpe.17.08.p7.12461256.

[5] Minmin Chen et al. "Marginalized denoising auto-encoders for nonlinear representations". In: *International Conference on Machine Learning* 32 (2014), pp. 1476–1484.

[6] Paolo Cremonesi, Yehuda Koren, and Roberto Turrin. "Performance of Recommender Algorithms on Top-n Recommendation Tasks". In: *Proceedings of the Fourth ACM Conference on Recommender Systems* (2010), pp. 39–46. DOI: 10.1145/1864708.1864721.

[7] Diana Ferreira et al. "Recommendation system using autoencoders". In: *Applied Sciences* 10.16 (2020), p. 5510.

[8] Krzysztof J. Geras and Charles Sutton. "Scheduled denoising autoencoders". In: *Conference on Learning Representations* (2015).

[9] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep learning.* MIT Press, 2016.

[10] Miha Grčar et al. "Data sparsity issues in the collaborative filtering framework". In: *International workshop on knowledge discovery on the web* (2005), pp. 58–76.

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

[12] Francesco Ricci, Lior Rokach, and Bracha Shapira. “Introduction to recommender systems handbook”. In: (2011), pp. 1–35.

[13] Yao Wu et al. “Collaborative denoising auto-encoders for top-n recommender systems”. In: *Ninth ACM International Conference on Web Search and Data Mining* (2016), pp. 153–162.

[14] Feng Yuan, Lina Yao, and Boualem Benatallah. “Adversarial collaborative auto-encoder for top-n recommendation”. In: *International Joint Conference on Neural Networks (IJCNN)* (2019), pp. 1–8.

[15] Shuai Zhang et al. “Deep learning based recommender system: A survey and new perspectives”. In: *ACM Computing Surveys (CSUR)* 52.1 (2019), pp. 1–38.

Appendix A

CDAE Implementation Details

The CDAE implementation presented in this work can be downloaded from `https://github.com/lauricdd/Collaborative-Denoising-Auto-Encoder`. Additional, the description of model can be seen in Chapter 2.

The publicly available source code provided by the authors [13] to be able to reproduce the experimental scenarios. Several libraries were used in the legacy code implementation such as `Glog` for logging, Google's test framework called `Gtest`, `Boost` for multi-threading and `Eigen` for linear algebra. The experiments for CDAE were run on a personal computer with `8GB` of RAM and `4` cores.

Legacy Code

In order to be able to replicate the results described in the paper, and to include some new features to the already existing code, it was necessary to gain a better understanding of the logic and behavior of legacy modules of the `c++ CDAE model`.

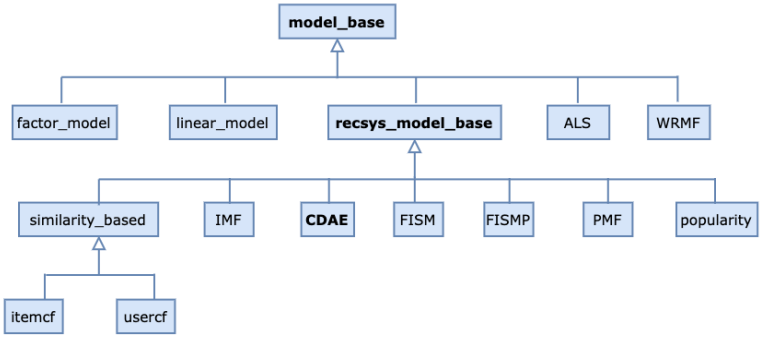

Figure A.1: CDAE legacy code hierarchy of classes

The classes, attributes, and relationships used are shown in figure A.2.

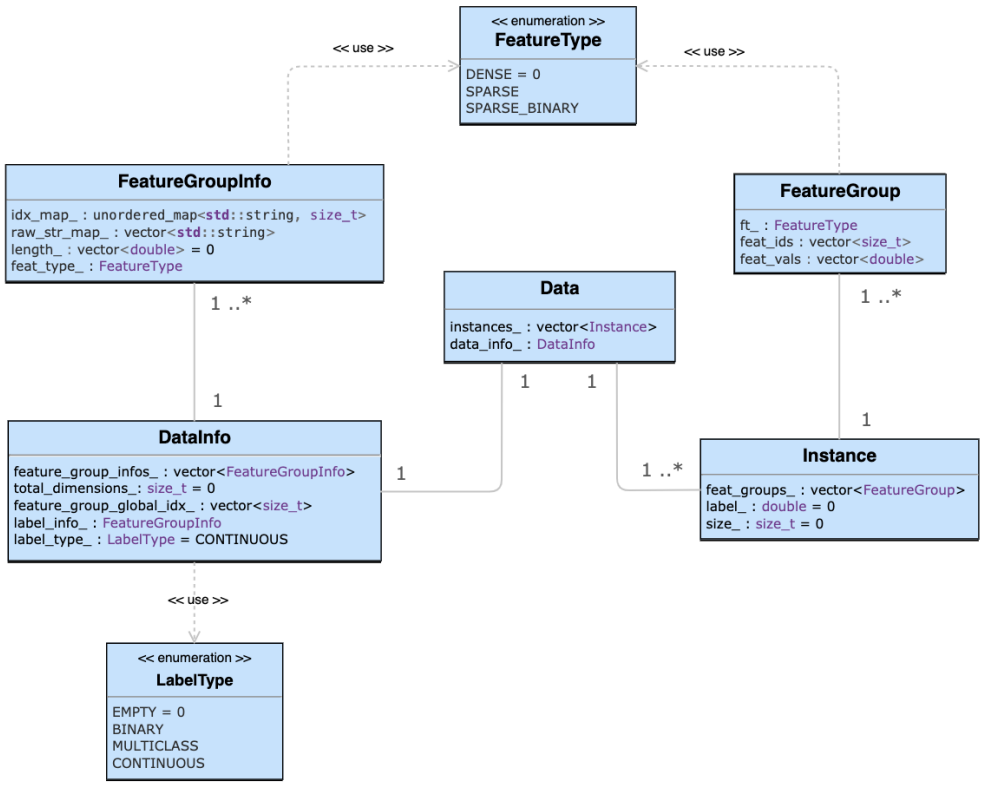

Figure A.2: CDAE Class Diagram

- `Data:` it contains the main, both the data instances itself and summary statistics on the data.
- `DataInfo:` summary on the dataset, num of instances, of feature groups, etc.
- `Instance:` corresponds to each line of the dataset, each interaction. It mainly contains a label, corresponding to the rating (in implicit datasets, 0 or 1), and two feature groups (for the user and the item IDs).
- `FeatureGroup:` values of each of the features, users and items.
- `FeatureGroupInfo:` summary on the feature groups.

CDAE model components

A basic configuration for the CDAE model is as follows:

```
CDAE config: {
    model_variant = M1    // model to train: "M1", "M2", "M3", "M4"
    num_dim = 50          // K : num of latent dims (hidden neurons)
    num_neg = 5           // NS: num of negative samples

    // training using SGD (and AdaGrad)
    max_iteration = 50    // max num of iterations
    learn_rate = 0.1      // learning Rate
    adagrad = true        // use AdaGrad
    beta = 1.0            // beta for AdaGrad
    holdout_perc = 0.2    // holdout data
    lambda = 0.01         // regularization rate
    penalty = L2          // regularization penalty

    // model settings
    // h(.): activation function on the hidden layer
    bool linear;            // true=identity, false=check tanh/sigmoid
    bool tanh;              // true=tanh, false=sigmoid
    bool linear_function;   // true=linear_mapping
    // f(.) activation function on the output layer
    bool sigmoid_output;    // true=sigmoid , false=identity
    string loss_type;       // loss function type l(.):
                            // "SQUARE", "LOG", "HINGE", "LOGISTIC", "CE"

    // additional parameters for experimental settings
    user_factor = true      // true=include user input node (CDAE), false=DAE
    asymmetric = true       // true=asymmetric DAE (tied weights)
                            // false=untied weights

    // input corruption
    corruption_type = "mask_out"    // "mask_out", "without_replacement",
                                    // "with_replacement"
    num_corruptions = 1             // number of corruption by input

    // params for mask_out corruption
    corruption_ratio = 0.0.          // level of corruption
```

```
    scaled = true                    // scaled input to prevent corruption bias

    // params for without_replacement replacement corruption
    num_removed_interactions = 2     // number of ratings to remove
    remove_same_interaction = false  // false=different interaction at each iteration
                                     // true=same interaction

    // params for with_replacement replacement
    num_corrupted_versions = 10      // num of user's profile corrupted replicas
}
```

Model parameters correspondance

In the implementation, there are separated variables for each of the model parameters, for the case in which they are updated using Adagrad ($_ag$ used) and the case they are not.

Paper	Code	Description
W	W and W_ag	Weights vector between the item input nodes and the nodes in the hidden layer.
W'	V and V_ag	Weights between nodes in the hidden layer and the output layer.
Vu	Wu and Wu_ag	Weight vector for the user input node.
b	b and b_ag	Weight vector for the bias node in the hidden layer.
b'	b_prime and b_prime_ag	Offset vector for the output layer.

Table A.1: CDAE model parameters correspondance

Model variants

The implementation of the 4 variants of CDAE model to be analyzed for the experiments (on chapter 4), according to the model parameters described above, are as follows.

```
// Model M1: h(.) = identity, f(.) = identity, l(.) = SQUARE
model = {.linear = true, .tanh = false,
         .linear_function = false, .sigmoid_output = false,
         .loss_type = "SQUARE"};
```

```
// Model M2: h(.) = identity, f(.) = sigmoid, l(.) = LOGISTIC
model = {.linear = true, .tanh = false,
         .linear_function = false, .sigmoid_output = true,
         .loss_type = "LOGISTIC"};

// Model M3: h(.) = sigmoid, f(.) = identity, l(.) = SQUARE
model = {.linear = false, .tanh = false,
         .linear_function = false, .sigmoid_output = false,
         .loss_type = "SQUARE"};

// Model M4: h(.) = sigmoid, f(.) = sigmoid, l(.) = LOGISTIC
model = {.linear = false, .tanh = false,
        .linear_function = false, .sigmoid_output = TRUE,
        .loss_type = "LOGISTIC"};
```