UNIVERSITAT DE BARCELONA

Facultat de Matemàtiques
i Informàtica

GRAU DE MATEMÀTIQUES
GRAU D'ENGINYERIA INFORMÀTICA

Treball final de grau

Context-Aware Recommender Systems with Graph Convolutional Embeddings (CARS-GCE)

Author: Lorenzo Andrés Vigo del Rosso

Director:	**Dr. Jordi Vitrià Marca**
Co-director:	**Paula Gómez Duran**
Conducted in:	**Departament de Matemátiques i Informática**

Barcelona, **24 de gener de 2021**

Contents

Abstract

The amount of online service providers is increasing every year, including multimedia streaming services and online shops. These services show a great interest in accurately recommending more products and more content to their costumers, as this strategy clearly encourages their clients to purchase or consume more items provided by them.

Recommender Systems are a useful tool that automatizes the task of predicting the preferences of the users of a service in order to recommend them items that will match their taste. Research on this area generally seeks for ways to improve the performance of the mathematical models these systems are based on in order to obtain better recommendations as result.

In this work, our main goal is to understand some traditional models used for recommendation and extend them so that they can detect complex patterns in the ratings given by users to items, capturing high-order interactions between features. Also, we aim to adapt them as Context-Aware Recommender Systems, which also take into account information about the context in which a user consumes a given item while computing their predictions.

First, the recommendation problem and Recommender Systems will be clearly defined and then, two traditional models will be introduced: Matrix Factorization and Factorization Machines. These both models are related to the concept of Embedding, which will also be detailed. It will be explained that these models present limitations that prevent them from capturing high-order interactions between features.

We aim to give the models the ability to capture these high-order interactions by using Graph Convolutional Networks (GCN) instead of the Embedding Layer. GCNs allow us to approach the recommendation problem as a graph link prediction problem, called Graph Convolutional Matrix Completion. GCNs encode the information of each feature in the graph and aggregates to it the correlated knowledge from neighboring features in the graph.

Then, the graph structure will be adapted so that context information can be included in it. Also, the models will be fed with item metadata, formatted as side-information.

Finally, we will detail the data used to train the model, how this data is treated and how the model is configured. In order to fairly compare the results obtained by each model, each one of their optimal settings will be calculated through

Bayesian Optimization. Afterwards, we will expose and analyze the results.

To conclude, it should be remarked that the inclusion of Graph Convolutional Networks with context information in the model implementation has a great positive impact on the results. Also, working with context in a traditional embedding structure may be benefitial only for specific models. The addition of item metadata shows different behavior depending on the metadata added and the model that is being evaluated.

In future work, we plan to check whether adding item metadata into the graph structure may have better results than including it as side-information. Also, we would aspire to extend the Bayesian Optimization to more model parameters and compare the model performances with different data representations and loss functions, among others.

Resumen

Cada año, la cantidad de proveedores de servicio en línea se ve incrementada, entre los cuales podemos encontrar servicios de *streaming* de contenido multimedia y tiendas en línea. Estos servicios muestran un gran interés en recomendarle a sus clientes más productos y más contenido de forma precisa, ya que esta estrategia de negocio les garantiza más consumo por parte de sus clientes.

Los sistemas de recomendación son una herramienta que permite automatizar la tarea de predicción de las preferencias de los usuarios de un servicio con el objetivo de recomendarles ítems que concuerdan con sus gustos. La investigación y el desarrollo en este área buscan mejorar el rendimiento de los modelos matemáticos en los que se basan estos sistemas con el objetivo de obtener mejores recomendaciones como resultado.

En nuestro trabajo, nuestro objetivo es la comprensión de ciertos modelos tradicionales utilizados en los sistemas de recomendación y extenderlos de manera que puedan detectar patrones complejos en las valoraciones dadas por los usuarios a los ítems, capturando interacciones de alto grado entre sus características. También, buscaremos adaptarlos como Sistemas de Recomendación con Contexto, los cuales tienen en consideración información sobre el contexto que rodea a un usuario cuando consume un ítem para calcular sus predicciones.

Para empezar, se definirán con detalle el problema de la recomendación y los Sistemas de Recomendación. Más adelante, se introducirán dos modelos tradicionales: la Factorización de Matrices y las llamadas *Factorization Machines*. Ambos

modelos se relacionan con el concepto de *Embedding*, el cual también se detallará. Se verá que estos modelos presentan limitaciones a la hora de capturar interacciones de alto grado entre características.

Tendremos como meta otorgarle a los modelos la habilidad de capturar dichas interacciones utilizando GCNs (*Graph Convolutional Networks* o Redes Convolucionales de Grafos), sustituyendo sus capas de *Embedding*. Las GCNs nos permiten interpretar el problema de la recomendación como un problema de predicción de enlaces en el grafo, el cual recibe el nombre de *Graph Convolutional Matrix Completion*. Las GCNs codifican la información de cada característica en el grafo y le agregan conocimiento correlacionado de características colindantes o vecinales.

Luego, se adaptará la estructura del grafo para que sea capaz de incluir información sobre el contexto. También, se alimentará a los modelos con metadatos sobre los ítems, los cuales se presentarán como *side-information*.

Finalmente, se detallarán los datos utilizados para entrenar los modelos, el tratamiento de dichos datos y como se configuran los modelos. Para poder asegurar comparaciones justas entre modelos, estos se configurarán según los valores obtenidos mediante Optimización Bayesiana. Más adelante, se expondrán y analizarán los resultados.

Para concluir, la inclusión de GCN con información sobre el contexto en la implementación de los modelos tiene un impacto positivo en los resultados. También podemos detectar que trabajar con contexto en una estructura basada en *Embeddings* tradicionales puede ser beneficioso solo en modelos concretos. La adición de metadatos sobre los ítems muestra diferentes comportamientos dependiendo de los metadatos que se añaden y del modelo que se evalúa.

Como propuesta de trabajo futuro, podríamos plantear la posibilidad de añadir los metadatos en la estructura del grafo en vez de presentarlos a los modelos como *side-information*. También, buscaríamos incluir configuraciones más avanzadas en la Optimización Bayesiana, como la representación de los datos o la función de pérdida utilizadas, entre otras.

2010 Mathematics Subject Classification. 68T05

Chapter 1

Introduction

As the amount of services and content available on the internet increases, the interest in automatic processes which are able to recommend users what they should do, consume, buy, read, listen to or watch next grows bigger. On one side, service providers desire more user engagement and, on the other, users want to know which products or items fit their interest best so that they can consume them. This generates a situation known as **the recommendation problem**, where companies attempt to predict users' preferences in order to offer them products that suit best their interests.

For example, an Amazon user that is searching for a product -or is about to buy it- will always find the information on what similar clients acquire or which products are often bought together useful. Displaying those details accurately will help to increase the number of purchases, in benefit of the company.

An analogous example would be Netflix showing a user which movies or series they should watch next: the watchers want more content that matches their taste and the platform looks for more watch-time.

Recommender Systems are a useful tool to deal with recommendation problems. They use available information of the past, in the shape of interactions between users and items that result in a rating or score from the user, to predict the score each user would assign to each still yet to rate items. The predictions with the highest score will be given to the user as recommendations. [1]

In order to compute these predictions, Recommender Systems use Machine Learning algorithms which are based on a set of data, known as training set, that allows to optimize a loss function that represents the solution we want to obtain.

The data used in Recommender Systems is generally represented as an inter-

action matrix. Each row of an interaction matrix represents the ratings given by a user and each column shows the ratings given to an item. This way, every cell contains the score given by a specific user to a specific item.

Recommender Systems suffer from the known as **cold start problem** [2]: they need known ratings in order to work, which generally are not available in the early stages of a service. Moreover, the more available items, the bigger the amount of known ratings that is required in order to compute accurate generalizations.

However, in order to avoid this issue, more information can be added to the models and base the initial predictions on it. This new data can be composed of user profiles (explicitly asking the user for their interests), item metadata (keywords, item classification), external data, etc. [1]

Nonetheless, even with this new input to the model, Recommender Systems may fail to identify patterns in users' behavior. Imagine a generic Netflix user enjoys watching Christmas-themed movies exclusively during the month of December. Traditional Recommender Systems would not take into account this kind of information and may recommend this kind of movie to the user also during the rest of the year, something that could be annoying to the user.

Consequently, we can conclude it is important to give the models information about the context surrounding the interactions between users and items. There are many variables that could be considered context such as date and time of the interaction, the location the interaction took place in or the device used to interact.

Due to this necessity, the concept of **Context-Aware Recommender Systems** [3] is introduced as a way to extend Recommender Systems and make them able to consider the context-related variables when computing its predictions.

In this work, we will start by carefully defining both Recommender Systems and Context-Aware Recommender Systems. Afterwards, we will explore two established models used for traditional Recommender Systems: **Matrix Factorization** and **Factorization Machines**. Our first goal is to explain how they work and then, we will introduce **Graph Convolutional Networks**, which we will use to extend these models so they can capture high-order interactions between features and so that they can be used as models for Context-Aware Recommender Systems.

Next, we will detail how to implement our context-aware models and also, how we can feed them with online data, seeking to include more advanced information about the content of items.

Finally, we will conclude by training and evaluating our extended models,

using online available datasets and metadata, and compare their performance in their optimal settings (which will be found using **Bayesian Optimization** [4]) and their results to those achieved by the traditional implementation of those models.

Chapter 2

State of the Art

As stated in the introduction, we will define what a Recommender System is and then deeply analyze two traditional models used for recommendation: Matrix Factorization and Factorization Machines. To end up the chapter, we will explain the concept of Graph Convolutional Networks and how they can be used to extend these models.

2.1 Recommender Systems

First of all, we can define a **Recommender System** in the following manner:

Definition 2.1. Given a set of users U, a set of items I and a rating space R, let $\mathbb{I} = U \times I$ be the set of all possible interactions and $r : \mathbb{I} \mapsto R$ be a function that associates each interaction with its rating. Given a subset $\mathbb{I}' \subsetneq \mathbb{I}$ of interactions which ratings are known, a *Recommender System* (RS) is a mathematical model that using $\mathbb{I}'$ is able to estimate the value of $r(i)\ \forall\ i \in \mathbb{I}$ by generalization.

According to [5], understanding **generalization** is arguably one of the most important questions in deep learning. For our intended purposes, we could define generalization this way:

Definition 2.2. A model presents the ability to *generalize* when it can infer properties of unobserved data or entities from the properties of observed or given data or entities.

It is important to evaluate the performance of the models by checking the accuracy of its generalization or by using other kinds of metrics. In our case, we count with a set of observed interactions with a rating, and we want a model to deduce the ratings that yet to occur interactions would have.

In simpler words, and as we explained in the introduction, a Recommender System uses past interactions among users and items in order to predict future interactions and estimate the users' preferences [1].

There are three types of traditional Recommender Systems.

- **Collaborative Filtering**: given a user u_k and an item i_m that u_k has not interacted with, this method assumes that the best way to calculate $r(u_k, i_m)$ is finding other users that have rated common items with u_k in a similar way and use their ratings to compute $r(u_k, i_m)$.

  For example, let $u_1, u_2 \in U$ be two users considered similar by the system and let $i \in I$ be an item only one of the users has interacted with. If it is imposed that u_1 has rated it, the system will deduce that u_2 will rate i approximately like u_1 did.

  This type of Recommender Systems may also find items that are rated similarly by the users. Let $i_1, i_2 \in I$ be two items that have been rated similarly by a subset of users $U' \subset U$. Let $u \notin U', u \in U$ be a user that has rated i_1 but not i_2. The system will deduce that u will rate i_2 as he rated i_1 in the past.

- **Content-based Filtering**: this method makes the assumption that the best way to calculate $r(u_k, i_m)$, where u_k is a user and i_m an item this user has not interacted with, is using the ratings given by u_k to items which are similar to i_m to compute $r(u_k, i_m)$.

  Given labels or any type of classification of the available items, let $i_1, ...i_k$ be a subset of items that have the same label. This label generally is based on the content of the items. Let u be a user that has rated of some of the items, for example, $i_1, ..., i_{k'}$ with $k' < k$, the system will deduce the user u will rate the items $i_{k'+1}, ..., i_k$ in the same manner as $i_1, ..., i_{k'}$.

- **Hybrid Systems**: consist in a weighted combination of the both previous types of Recommender Systems.

The previous definition of a Recommender System is easily extendable to **Context-Aware Recommender Systems**. Considering a context space C that represents the value of each one of the context-related variables we want our system to take into consideration, we are now able to define the concept of **Context-Aware Recommender Systems**.

Definition 2.3. Given a set of users U, a set of items I, a context space C and a rating space R, let:

- $\mathbb{I} = U \times C \times I$ be the set of all possible interactions,
- $\mathbb{I}(c) \subsetneq \mathbb{I}$ be the subset of all possible interactions in a fixed context c,
- $r : \mathbb{I} \mapsto R$ be a function that associates each interaction with its rating
- and $r\Big|_c : \mathbb{I}(c) \mapsto R$ be the restriction of r to our previously fixed context c.

Given a subset $\mathbb{I}' \subsetneq \mathbb{I}$ of interactions which ratings are known, a *Context-Aware Recommender System* (CARS) is a mathematical model that using $\mathbb{I}'$ is able to estimate the value of $r\Big|_c \ \forall\, i \in \mathbb{I}(c)$ by generalization.

It should be noted that it is generally impossible to compute the value of $r(i)$ $\forall\, i$ due to the infinite nature of some context variables, such as date or time. If the number of possible context values is manageable, the restrictions to context are not needed.

The rating space should include all the possible values a user can use to punctuate any item. Some examples could be $R = \{0, 1\}$, $R = [0, 5]$ or $R = \{-, +\}$. It is recommendable to use either binary, labeled or continuous ratings as the estimations will never be exact.

Having reached this point, how the generalization in said systems is carried out should be explained. Both RS and CARS may use different mathematical models in order to get this task done. Recommender Systems should be implemented in a way that assures an easy integration of new models. We will now introduce two of them, which we will use during the rest of our work.

2.2 Matrix Factorization

Matrix Factorization is a model used for Collaborative Filtering Recommender Systems based on matrix decomposition [6]. It requires a representation of the possible interactions and ratings in the shape of a matrix that is commonly named **adjacency matrix** (or rating matrix).

Definition 2.4. Given a set of n users $U = \{u_j\}_{j \in [0,n)}$ and a set of m items $I = \{i_k\}_{k \in [0,m)}$, the *Adjacency Matrix* associated to our Recommender System is a matrix $A = \{a_{jk}\}_{j,k} \in \mathbb{M}_{n \times m}$ where $a_{jk} = r((u_j, i_k))$, being r the rating function of our RS.

It should be reminded that a Recommender System presents a set of possible interactions $\mathbb{I}$ represented by tuples of a user and an item, a rating space R and a function $r : \mathbb{I} \mapsto R$ that relates each interaction with its rating.

For instance, if we suppose that we have a total of 3 users and 4 items, that the rating space is $R = [0,5]$ and that all the interactions had a rating associated to them, the adjacency matrix could look like this: [1]

$$A := \begin{array}{c} \\ u_0 \\ u_1 \\ u_2 \end{array} \begin{array}{cccc} i_0 & i_1 & i_2 & i_3 \\ 2 & 1 & 5 & 3 \\ 3 & 4 & 2 & 1 \\ 5 & 3 & 3 & 0 \end{array}$$

Matrix Factorization works by decomposing the adjacency matrix.

Definition 2.5. Let k be an arbitrary value and given an adjacency or rating matrix $A \in \mathbb{M}_{n\times m}$, the *Matrix Factorization* model consists in finding two matrices $P \in \mathbb{M}_{n\times k}$, $Q \in \mathbb{M}_{k\times m}$ such that $A \approx PQ$.

There are several matrix decomposition methods that back up the existence of said matrices, such as Singular Value Decomposition (SVD) or Probabilistic Matrix Factorization (PMF). The one method that suits our purpose best in this work is **Non-negative Matrix Factorization** (**NMF** or NNMF). [6] [7]

Definition 2.6. Given a matrix V such that all of its elements are non-negative, the *Non-Negative Matrix Factorization* is an algorithm that will result in the decomposition of V in two matrices W, H with no negative elements such that $V \approx WH$.

It should be noted that the elements of our adjacency matrix are included always in a rating space defined by ourselves. So, we can impose a non-negative rating system or adapt the rating system to this algorithm. For instance, if users were to dislike or like items, we would define our rating space as $R = \{0,1\}$ instead of $R = \{-1,+1\}$. Also, there are other decomposition methods that may fit different scenarios better. Some of them can be found in [6].

Following the NMF algorithm with a dimension $k = 2$, the previously defined adjacency matrix would be decomposed like this:

$$\begin{pmatrix} 2 & 1 & 5 & 3 \\ 3 & 4 & 2 & 1 \\ 5 & 3 & 3 & 0 \end{pmatrix} \approx \begin{pmatrix} 1.7885 & 0.0000 \\ 0.2396 & 1.6664 \\ 0.2942 & 2.0242 \end{pmatrix} \begin{pmatrix} 1.1188 & 0.5584 & 2.7959 & 1.6768 \\ 2.0376 & 1.7722 & 0.9636 & 0.0000 \end{pmatrix}$$

[1] The labels u_j and i_k are only added this time for clarification. Each row of the matrix represents the ratings given by a user to each one of the items and each column represents the ratings received by an item.

Each one of the rows of matrix P receives the name of user embedding vector. Something similar happens with matrix Q, which columns are called item embedding vectors. They are also named latent vectors. The **embedding vectors** live in a vector space named **Embedding Space**.

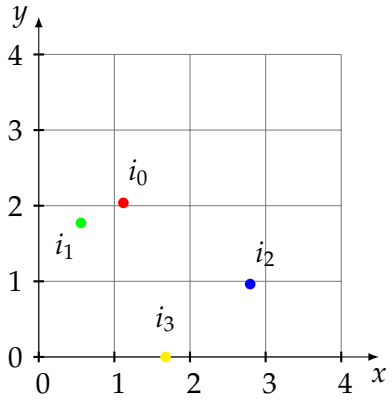
Definition 2.7. *Embedding space* is the space in which the data is embedded after dimensionality reduction. Its dimensionality is typically lower than that of the ambient space. [8]

In other words, the embedding space defines a space where the different entities (in our case, users and items, and context if included) will live. The dimensionality of this space is determined by the value k, named latent, embedding or **hidden dimension**. In practice, each dimension will represent an entity feature and similar instances of said entities will be close neighbours in the dimensions where they present resemblance.

The idea is to assume that users rate the items, consciously or not, influenced by the features characterized by the embedding or latent vectors.

Based on the values obtained from our example matrix A, we may represent the items in their item embedding space like this:

Item Embedding Space



In our case, as the NFM method generates two matrices W and H with no negative elements, our embedding space is logically restricted to the first quadrant. The space is bi-dimensional because we previously imposed that $k = 2$. We can see that i_0 and i_1 would be the most similar pair of items in our set. Also, it is worth noting that, even though i_1 and i_2 are very different in the feature characterized by the x axis, they are relatively similar in terms of the other feature.

So far, we have been supposing that the adjacency matrix would be full. This

is not a realistic scenario as these matrices generally are very big and users can not rate all the items. In fact, the adjacency matrix is very sparse. Let's assign the question mark symbol (?) in the matrix to every interaction user-item that has not occurred yet:

$$A := \begin{pmatrix} 2 & ? & ? & 3 \\ 3 & 4 & 2 & ? \\ ? & 3 & 3 & ? \end{pmatrix}$$

The conventional approach of Non-Negative Matrix Factorization method works through error minimization. It seeks to find W and H by minimizing the difference between V and WH [7], imposing that $W_{ia} \geq 0, H_{bj} \geq 0 \; \forall a, b, i, j$:

$$\min_{W,H} f(W,H) = \frac{1}{2} \sum_{i=1}^{n} \sum_{j=1}^{m} (V_{ij} - (WH)_{ij})^2 \tag{2.1}$$

We can adapt the process so it minimizes the error only in the observed values. This way, WH would be a matrix such that $V_{ij} \approx WH_{ij}$ if the interaction between user i and item j is an observed value. The generated matrix will include new values in the cells that correspond to the interactions that have not happened yet. These new values can be used as rating predictions.

Finally, in order to fetch recommendations for each user, we should retrieve the highest values in every row of the matrix that do not correspond to ratings actually done by the user.

2.3 Factorization Machines

Definition 2.8. *Factorization Machines* are general predictors able to estimate reliable parameters under very high sparsity. [9]

This model works with any data input shaped as **multi-label One Hot feature vectors**. For this reason, this model can easily add side-information to their predictions. Factorization Machines can model different degree relationships between variables using factorized parameters [9].

Definition 2.9. The *One Hot Encoding* transforms d distinct values to d binary variables. [10]

These variables can be recollected in a vector and, in our case, each one of them represents a feature. The value of these variables in a feature vector linked

to a specific entity indicates if said feature describes or not the entity. The term multi-label implies that more than one variable can be set to 1 in a same One Hot Feature Vector.

First, it should be noted that we will work with the same Recommender System structure as seen in 2.1. In order to consider side-information in this structure, we will define a vector space $\mathbb{V}$ where the feature vectors will live. r is now considered as a composition of functions $r = r' \circ f$, where $f : U \times I \mapsto \mathbb{V}$ and $r' = \mathbb{V} \mapsto R$. f is the function that constructs the feature vector that corresponds to each interaction, and r' is the function that connects said feature vector with the rating its interaction received.

As a first task, the feature vectors without side information should be built. Given $\#\mathbb{I}' = n$ observed interactions, $\#U = j$ users and $\#I = k$ items, then n $\mathbb{R}^{j+k}$-vectors must be constructed. When side-information is added, the vectors will have a larger dimension. Every feature vector will have a target associated to it, which represents the rating given in the interaction.

For example, imposing that we have 3 users (Juan (J), Marta (M) and Pep) and 4 movies (Batman (B), Inception (I), Finding Nemo (F) and Superman (S)). We will consider the following observed interactions:

- Juan rated Inception ($x^1 = (J, I)$) with a 3 ,
- Juan also rated Batman ($x^2 = (J, B)$) with a 5,
- Marta rated Superman ($x^3 = (M, S)$) with a 1,
- Marta also rated Finding Nemo ($x^4 = (M, F)$) with a 5 and
- Pep rated both Batman and Superman ($x^5 = (P, B)$, $x^6 = (P, S)$) with a 4.

The associated feature vectors would look like this: [2]

	J	M	P	B	I	F	S	Rating
x^1	1	0	0	0	1	0	0	3
x^2	1	0	0	1	0	0	0	5
x^3	0	1	0	0	0	0	1	1
x^4	0	1	0	0	0	1	0	5
x^5	0	0	1	1	0	0	0	4
x^6	0	0	1	0	0	0	1	4

[2]Please note that the 'Rating' column is not part of the feature vectors. We added it to the table as a convenient way to portray all the available information at once.

The second task is adding side-information to these vectors. For instance, we could choose to add the movie genre. This information is very rich since two movies classified in the same genre are more prone to be alike and to generate similar ratings by each user. In this case, 3 genres will be considered (Animation (A), Science Fiction (SF) and Superheroes (SH)) which columns will be added to the end of each vector:

	J	M	P	B	I	F	S	A	SF	SH	Rating
x^1	1	0	0	0	1	0	0	0	1	0	3
x^2	1	0	0	1	0	0	0	0	0	1	5
x^3	0	1	0	0	0	0	1	0	0	1	1
x^4	0	1	0	0	0	1	0	1	0	0	5
x^5	0	0	1	1	0	0	0	0	0	1	4
x^6	0	0	1	0	0	0	1	0	0	1	4

It is desirable to generate a model that can conclude that, since Pep liked both Batman and Superman, users will consider those two movies similar. In that case, as Juan enjoyed Batman, the model should deduce that he will be interested in Superman too. Our model ought to identify Marta and Pep as two users with opposite tastes for their ratings on Superman. For this reason, Marta and Juan should also have different recommendations due to the similarity found before between Juan and Pep. Finally, taking into account these conclusions and Marta's ratings, the model should not recommend Finding Nemo to the other users. A Factorization Machine would get this job done.

Factorization Machines work as predictors by estimating a function $y : \mathbb{R}^n \mapsto T$ that takes any real vector and returns a value included in a defined target space T [9], which can be imposed to be the rating space R. This function aims to be an extension to $\mathbb{R}^n$ of the previously defined function r'.

This function can be based on different models. We will start explaining the simplest one and then extend it progressively:

$$y(x) = w_0 + \sum_{i=1}^{n} w_i x_i \tag{2.2}$$

Where x_i is the i-th component of the input feature vector x. Several parameters can be found in this model definition. In this formula:

- w_0 is the model's general bias.

- w_i for $i = \{1, ..., n\}$ represents the strength of the i-th variable in the rating prediction.

In the previously defined example, it can be deduced that $w_3 = w_P$ will likely have a higher value because Pep has rated all the movies he has watched positively. Something similar happens with $w_4 = w_B$, since all ratings given to Batman have been good.

It could be said that this model has degree 1, since it only takes into account each variable on its own. Once a feature vector is input, the function will calculate its prediction using only the information on the strength of the data included in that input. This way, the model will hardly identify more complex patterns in rating. To solve this, we will now analyze the model that corresponds to degree 2:

$$y(x) = w_0 + \sum_{i=1}^{n} w_i x_i + \sum_{i=1}^{n} \sum_{j=i+1}^{n} \hat{w}_{i,j} x_i x_j \tag{2.3}$$

- The new parameters $\hat{w}_{ij}$ model the interaction between the i-th and the j-th variable. This value will only be relevant when both variables are equal to 1 in the input feature vector, and, in that case, it quantifies the strength of that pair of active variables.

This model will generally be used in very sparse problems with a huge number of variables to include in the feature vectors. This would cause an even bigger number of parameters $\hat{w}_{ij}$ that need to be calculated. In order to avoid this, these parameters will be factorized by defining a latent or embedding vector v_i of size k for every variable, where k is both the number of factors and the dimensionality of the factorization. Intuitively, every vector v_i represents every variable with k factors.

Now, if $\langle \cdot, \cdot \rangle$ represents the dot product, we can define the following notation:

$$\hat{w}_{i,j} = \langle v_i, v_j \rangle \tag{2.4}$$

Using this, we can define our model with a reasonable number of parameters:

$$y(x) = w_0 + \sum_{i=1}^{n} w_i x_i + \sum_{i=1}^{n} \sum_{j=i+1}^{n} \langle v_i, v_j \rangle x_i x_j \tag{2.5}$$

The dot product between two embedding vectors works as an **embedding layer**. As stated in [11]:

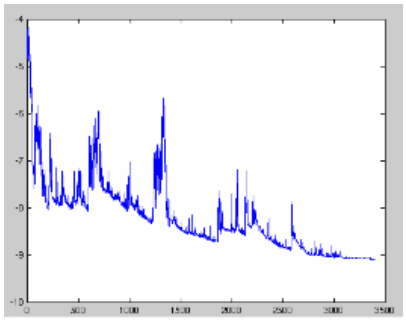


Figure 2.1: Figure taken from [12]. SGD fluctation.

Definition 2.10. An *embedding layer* is a representation of an entity (node, word, letter, etc.) with k latent factors. It is more computationally efficient than other encoding ways such as one-hot encoding when using very big datasets. The embeddings get updated during the training process of a deep neural network, and allows to identify entities that are similar to each other in a multi-dimensional space.

Let $W \in \mathbb{M}_{n\times n}$ be the interaction matrix (that is, $W = (\hat{w}_{i,j})$), we know there exists a matrix $V \in \mathbb{M}_{n\times k}$ where k is large enough such that $W = VV^t$. Each row of V would be one of the vectors v_i we defined before. Therefore, the existence of the factorization could be proven.

In our example of observed ratings, there are a couple deductions we can take directly from the given data. For instance, we can be mostly certain that $\hat{w}_{2,8} = \hat{w}_{M,A} > \hat{w}_{M,SH} = \hat{w}_{2,10}$ because Marta rates animation movies better than films about superheroes. Following the same logic, $\hat{w}_{2,10} = \hat{w}_{M,SH} < \hat{w}_{P,SH} = \hat{w}_{3,10}$ given that Pep enjoys superheroes more than Marta.

Finally, we need to know how to compute the value of all of the parameters in the formula in order to be able to use the model for predictions. This will be achieved by training the model with a set of observed interactions and by using the **Gradient Descent method**. As stated in [12]:

Definition 2.11. *Gradient descent* is a way to minimize an objective function $J(\theta)$ parameterized by a model's parameters $\theta \in \mathbb{R}^d$ by updating the parameters in the opposite direction of the gradient of the objective function $\nabla_\theta J(\theta)$ with respect to the parameters. The learning rate η determines the size of the steps we take to reach a (local) minimum. In other words, we follow the direction of the slope of the surface created by the objective function downhill until we reach a valley.

In other words, the method works in the following manner: first of all, parameters are set to arbitrary or random numbers (generally, all of them are set to 0). Then, the next steps are repeated until the iterations do not minimize the objective function anymore (or at least not enough in order to consider it profitable to keep the method going) or after a definite number of iterations.

First, predictions of the model with the currently assigned parameter values are computed for a set of interactions with known rating. Afterwards, the error between predictions and targets (the actual ratings) is calculated using the chosen objective Loss Function. The objective function should always be a loss function, since minimizing it will result in minimizing the error in the predictions computed by the model.

In our case, the objective function will be the BPR Loss Function as seen in [13]:

$$BPR - Opt(\theta) = \sum_{(u,i,j) \in D_S} ln\ \sigma(\hat{x}_{uij}) - \lambda_\theta \|\theta\|^2 \tag{2.6}$$

Where D_S is the training set, σ is the sigmoid activation function, $\hat{x}_{uij}$ is a function that represents the relationship between a user u and two items i and j, θ is the parameter vector of a definite model class (e.g. Matrix Factorization or Factorization Machine) and λ_θ are model specific regularization parameters. Further in our work, in chapter 4, our chosen data representation will be explained.

Then, the gradient of the loss function must be calculated, which indicates in which direction the steps should be oriented to in a space formed by all parameters in order to decrease the error in our predictions. If we analyze such 'direction' axis by axis, that is, parameter by parameter, the gradient computation is indicating whether the value of each parameter should be decreased or increased. Therefore, the parameter values should be changed according to this direction and the step size defined by η.

After enough iterations are executed, we can retrieve the parameter values learnt by the Gradient Descent method. This optimizing method may get stuck in local minimums of our objective function. However, this is only often in low-dimensional scenarios and not in problems with a greater amount of parameters, which is our case.

To end this section, we will state that the Factorization Machine model can in fact be defined for higher degrees, and it may even be generalized for any degree d. In this work, we will not follow this path of extension which can be consulted in [9]. However, we will aim to use Graph Convolutional Networks in order to implement a more powerful model.

2.4 Graph Convolutional Networks

Factorization Machines make adding some side-information to the interaction possible through extending the feature vectors, but such data structure may provoke a very high number of parameters. The number of needed parameters depends on the size of feature vectors, which grows if we try adding features that may have many different values. For example, we could be interested in adding the location each interaction took place in: we would need to add a new column to our vectors for every unique location. In the case of Matrix Factorization, we still do not know any way to add any information to the adjacency matrix.

In this section, we will give an insight on how to use **Knowledge Graphs** to extend our Factorization Models.

Definition 2.12. A *Knowledge Graph* is a multi-relational graph composed of entities and relations which are regarded as nodes and different types of edges, respectively. [14]

A graph $\mathcal{G}$ is generally defined by $\mathcal{G} = (\mathcal{V}; \mathcal{E})$ where:

- $\mathcal{V}$ is a set of nodes representing several entities. In our case we must have nodes representing users and items, and optionally other entities.
- $\mathcal{E}$ is a set of edges or connections between nodes. They represent relationships between instances of the entities included in the graph.

It is convenient to represent the available data in a Knowledge Graph since it is an easier way to consider more entities involved in the interactions. Also, it is useful to prevent the Factorization Machine from having to handle too many parameters. The management of variables with many possible values is rather carried out by more advanced structures applied on the graph than by the Factorization Machine model.

In order to identify rating patterns on the graph, the objective is to learn a function of signals or features on it, by using a **Graph Convolutional Network** (GCN) [15]. GCNs perform convolutions on the graph in the same way Convolutional Neural Networks do on images. They work by assuming that connected nodes tend to be similar and share labels and they enable us to identify high-order interactions between features.

GCNs take as input:

- A matrix $X \in \mathbb{M}_{n \times d}$ where n is the number of nodes and d is the number of features per node. This matrix is called **Feature Matrix** and every row of it represents a description in d features of one of the nodes.
- A matrix $A \in \mathbb{M}_{n \times n}$ called **Adjacency Matrix** that describes the connectivity in the graph, by specifying the edges in the graph.

Multi-layer Graph Convolutional Networks follow this propagation rule:

$$H^{(l+1)} = \sigma(\hat{D}^{-\frac{1}{2}}\hat{A}\hat{D}^{-\frac{1}{2}}H^{(l)}W(l)) \tag{2.7}$$

Where:

- D is the diagonal matrix that normalizes A by taking the average of neighboring node features. $D := (d_{ij})_{ij}$ where $d_{ii} = \sum_j A_{ij}$.
- $\hat{A} = A + I$, where I is the Identity Matrix ($I \in \mathbb{M}_{n \times n}$). This way, we add self-connections to our adjacency matrix and we will reflect the node itself in the graph interactions.
- σ is an activation function (for example, ReLU).
- $W^{(l)}$ is a matrix of layer-specific trainable weights.

We will impose that $H^{(0)} = X$ and the Graph Convolutional Matrix will convert these input features through convolution into an output feature matrix $H^{(l)} = Z \in \mathbb{M}_{n \times k}$, where k is the number of output features.

GCNs are needed for Graph Convolutional Matrix Completion (GC-MC), where the matrix completion problem is reinterpreted as a graph link prediction problem [15]. From now on, we will consider that an edge between a user i and an item j represents a rating from user i to item j.

In [15], a graph auto-encoder framework is proposed, which works by producing latent features of all nodes, which later are used to reconstruct the graph. They state that this type of local graph convolution can be seen as a form of message passing, where vector-valued messages are being passed and transformed across edges of the graph. In their case, they can assign a specific transformation for each rating level, resulting in edge-type specific messages $\mu_{j \to i,r}$ for items j to users i of the following form:

$$\mu_{j \to i,r} = \frac{1}{c_{ij}} W_r x_j \tag{2.8}$$

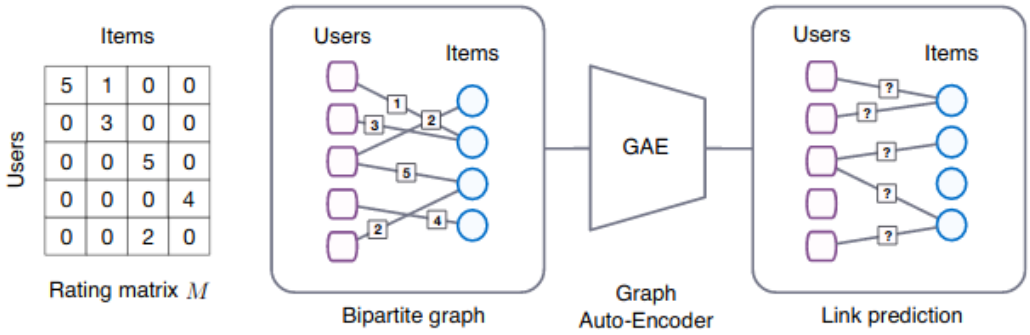


Figure 2.2: Figure taken from [15]. *Left*: Rating matrix M with entries that correspond to user-item interactions (ratings between 1-5) or missing observations (0). *Right*: User-item interaction graph with bipartite structure. Edges correspond to interaction events, numbers on edges denote the rating a user has given to a particular item. The matrix completion task (i.e. predictions for unobserved interactions) can be cast as a link prediction problem and modeled using an end-to-end trainable graph auto-encoder.

Here, c_{ij} is a normalization constant, which we choose to either be $|\mathcal{N}_i|$ (left normalization) or $\sqrt{|\mathcal{N}_i||\mathcal{N}_j|}$ (symmetric normalization), with $\mathcal{N}_i$ denoting the set of neighbors of node i; W_r is an edge-type specific parameter matrix, and x_j is the (initial) feature vector of node j.

In our work, we will not consider the punctuation given in ratings. As stated in [13], most feedback by users in real-world scenarios is implicit and not explicit. **Implicit Feedback** is generally much easier to collect as the users do not have to give feedback explicitly and it is already available in most of information systems in the shape of user activity logs. Some examples of implicit feedback are the items previously purchased, watch-time recollection and click monitorization. In our case, we will consider item consumption. A user rating an item implies a previous consumption of this item by said user and we will consider that consuming an item implies choosing at one point said item over the rest. Therefore, all of the ratings will be considered as positive interactions, while not having rated an item will be considered as a negative sample or interaction. This means that our adjacency matrix will turn out to be a binary matrix. For this reason, in our case, edge-type specific messages will not be necessary, as there only will be one kind of edge since a user may only rate or not an item.

The idea behind the link prediction can be simply explained. Given our graph

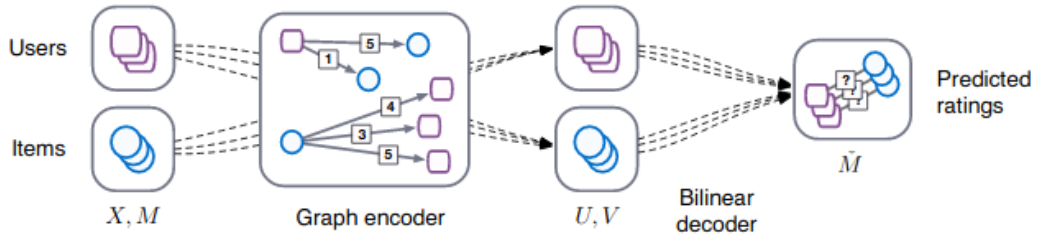


Figure 2.3: Figure taken from [15]. Schematic of a forward-pass through the GC-MC model, which is comprised of a graph convolutional encoder $[U, V] = f(X, M_1, ..., M_R)$ that passes and transforms messages from user to item nodes, and vice versa, followed by a bilinear decoder model that predicts entries of the (reconstructed) rating matrix $\hat{M} = g(U, V)$, based on pairs of user and item embeddings.

with edges and user and item nodes, the cited framework will provide us with a graph encoder that translates each kind of node to a latent or embedding space through convolution. In some sort of way, as a function has its inverse function, this encoder counts on a bilinear decoder in order to reconstruct the graph. This bilinear decoder generates new unseen values when elements that were not included in the original graph are input. These new values may be considered as link predictions for yet to be seen interactions between users and items.

In order to integrate the GCN into a Factorization Machine, we can easily adapt the model this way:

$$y(x) = w_0 + \sum_{i=i}^{n} w_i x_i + \sum_{i=1}^{n} \sum_{j=i+1}^{n} \langle g(x_i), g(x_j) \rangle x_i x_j \tag{2.9}$$

where g is the GCN embedding function, which will provide our Factorization Machine model with the ability to capture high order interactions and signals. Basically, the embedding layer is being substituted by a more advanced embedding, provided by the GCN module.

Finally, in this work, we will seek to implement all the cited models above and aim to compare them with a new possible extension. Not only do we want to include user and nodes in our graph structure, but we also want to add the context surrounding every interaction as another type of entity or node. Our goal is to compare all the results obtained through the different models in a fairly chosen parameter setting for all of them and check whether adding context leads to better results or not.

To fulfill our purpose of obtaining fair comparisons, we intend to use a Bayesian Optimization framework to evaluate each model in their optimal hyperparameter settings. [4]

Lastly, we will also analyze the impact of side-information inclusion in the shape of feature vectors by extending the used datasets with metadata found online.

Chapter 3

Implementation

In this chapter, we will analyze how to extend Graph Convolutional Networks to context, in order to integrate them into the models previously explained in our work so that we can create Context-Aware Recommender Systems based on them.

Given a dataset that includes information about the context around every interaction, the first step is to include the context in the data structures that have been defined so far. Previously, ratings were represented in the graph as an edge between a user node u and an item node i, accompanied by a label that represented the score given in such rating. [1]

Now our graph will include 3 kinds of entities: users, contexts and items. Given a context space C with n variables, interactions (which used to be an edge *user-item*) will now be represented by:

- A set of edges from a user node u to n context nodes $c_1, ..., c_n$,
- a set of edges from the same n context nodes to an item node i and
- an edge connecting u and i.

This set of paths can be represented as u-c_1-c_2-...-c_{n-1}-c_n-i. However, it must be pointed out that context nodes are not connected with each other.

We will impose that all edges are bidirectional to represent ratings as an interaction where every entity involved influences the rest with their features. This is important in the construction of the graph adjacency matrix.

For each context variable, a new node for every value the variable takes in the observed interactions will be added. If we suppose that we can identify 3 users, 3

[1] It should be reminded that, in our case, as we are working with implicit feedback, such labels are not needed.

contexts with up to 2 different possible values each and 2 movies, the next figure would be a possible graphic representation of the graph.

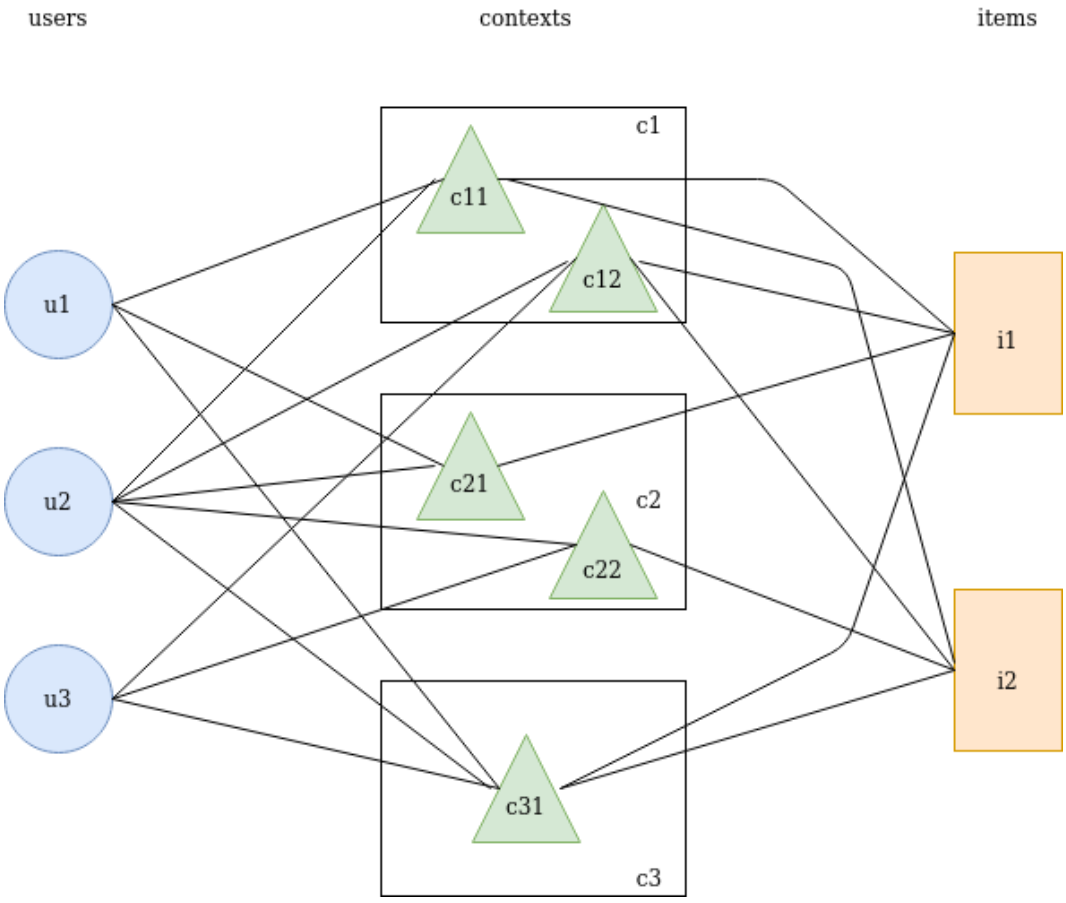


Figure 3.1: In this example, context is defined by 3 variables: c_1, c_2, c_3. The first two variables have 2 possible values while the last one only one. The nodes are grouped accordingly in the graph representation. Notice how reconstructing the interactions from the graph gets trickier than before. In fact, it may even be impossible to do so. In our case, though, we can retrieve the following interactions: $u_1 - c_{11} - c_{21} - c_{31} - i_1$, $u_2 - c_{12} - c_{21} - c_{31} - i_1$, $u_2 - c_{11} - c_{22} - c_{31} - i_2$ and $u_3 - c_{12} - c_{22} - c_{31} - i_1$

This new graph structure implies a different adjacency matrix. It should be noted that the dimensions of this matrix also change, as the number of nodes has been incremented. As always, this matrix will represent the edges that form the graph. In this case, we should note that, since context nodes are never connected

with each other, most of this matrix will be full of zeros. Also, as we are considering that all links are bidirectional, the matrix will be symmetrical. We will represent the matrix structure by defining it block by block:[2]

$A =$

	users	*items*	$context_1$	$context_2$	...	$context_n$
users	0	$u - i$	$u - c_1$	$u - c_2$	...	$u - c_n$
item	$i - u$	0	$i - c_1$	$i - c_2$	...	$i - c_n$
$context_1$	$c_1 - u$	$c_1 - i$	0	0	...	0
$context_2$	$c_2 - u$	$c_2 - i$	0	0	...	0
...	...	...	...	...	...	...
$context_n$	$c_n - u$	$c_n - i$	0	0	...	0

In this representation, $u - i$ and $i - u$ stand for the edges between items and users and vice versa. Also, $u - c_i$, $i - c_i$, $c_i - u$ and $c_i - i$ represent the edges between items or users and context and vice versa. As we wanted to show, having divided our adjacency matrix A in $(n+2)^2$ blocks where n is the number of context variables, there will always be at least $n^2 + 2$ empty blocks and at most $4n + 2$ non-empty blocks.

It should not be forgotten that Graph Convolutional Networks take two matrices as input: the matrix adjacency A and a matrix X that is formed by a feature vector per node in the graph. The rows of X work as a descriptor of each node

[2] As the nodes are not self-connected, the diagonal of A will be full of zeros. It should be reminded though that the GCN module will work with $\hat{A} = A + I$, as seen in section 2.4.

in the graph. It is recommended to define X as a column-wise extension of the n-dimensional identity matrix so that, even if no node features are added, the nodes will describe themselves.

To end this chapter, the way messages are passed in GCNs needs to be adapted in order to fully extend Factorization Machines with Graph Convolutional Networks (FM-GCN) as a context-aware model. As stated in [11], the local convolution in [15] is seen as a sort of message passing from items to users and vice versa, as seen in 2.8. Now we need the message to pass through all the contexts involved in the interaction. This can be achieved with the following adaptation of the equation, given n context variables $(c_1, ..., c_n)$:

$$\mu_{j,c_1,...,c_n \to i} = \frac{1}{\hat{c}_{ij}}((Wx_j)x_{c_1}...)x_{c_n} \tag{3.1}$$

Where W is a tensor with as many dimensions as the amount of nodes in the graph. In this case, we are treating message passing through several nodes as a composition of several two-node message passing: the result of passing the message from one node to another is passed to the next node. The result is similar to the one that would be obtained by function composition, but in this case, maintaining only one constant $\hat{c}_{ij}$ and only one parameter matrix W. In order to avoid working with multi-dimensional arrays, this other variant can be used:

$$\mu_{j,c_1,...,c_n \to i} = \frac{1}{\hat{c}_{ij}}(W_j x_j + W_{c_1} x_{c_1} + W_{c_n} x_{c_n}) \tag{3.2}$$

Let N be the number of nodes in the graph, it can be stated that $W \in \mathbb{M}_{n \times d}$, where each row is an embedding of a given entity. In 3.2, each node feature embedding x_i is accompanied by the corresponding node's entity embedding included in W, noted as W_i.

Having all the needed pieces defined, we just need to put them together in our code. Given a baseline Factorization Machine, we will adapt the model as shown in 2.9. Through this adaptation, we are substituting the embedding layer of the Factorization Machine with a Graph Convolutional Network that performs said embeddings. We expect the Graph Convolutional Network Embeddings $g(x_i)$, also named **Graph Convolutional Embeddings** (GCE), to encode the information of x_i and from other correlated features.

GCEs also work as a layer, which means that they can be integrated into any traditional model that uses an embedding layer. In fact, we should note that we also extended Matrix Factorization to context in a similar way.

The provided code is our implementation of all that has been explained during our work. We chose to develop a *Python-based project*, as Python is a Programming Language that provides a large amount of libraries that are very useful for Machine Learning related purposes. In fact, as stated in [16]: Python is the most preferred language for scientific computing, data science, and machine learning, boosting both performance and productivity by enabling the use of low-level libraries and clean high-level APIs.

We use libraries that are useful for data management, such as *Numpy*, *Pandas* and *PyTorch*. An extension of PyTorch called *PyTorch Geometric* counts with a series of various methods and classes for deep learning on graphs and other irregular structures [17], including a module that implements Graph Convolutional Networks. This library is the keystone of our project.

To end up this chapter, we will describe our project structure. It counts with a number of scripts that handle the downloading and the processing of the data the models will be fed with. Also, it provides with a script that looks for the optimal parameters for each model using Bayesian Optimization, as explained in section 4.2. A folder with model implementations can be found, all of which may work with or without context and with traditional embeddings or Graph Convolutional Embeddings.

The main script is responsible for adding context to the dataset and performing the preprocessing, the splitting and the negative sampling that will be explained in the next chapter. Finally, the main script also executes the training iterations for each model, through Gradient Descent and using the $BPR - Opt$ Loss Function. It also validates and evaluates the trained models calculating the metrics described in section 4.2 and generates logs of the results obtained throughout the training iterations, which can be plotted as it will be shown in section 4.3.

Chapter 4

Experiments

Now it is our turn to evaluate our models. We will describe which datasets we are using, what data they are comprised of, how we will treat and process our data, the model settings and the results we obtain.

4.1 Datasets with context

We needed a dataset that included a set of users, a set of items and a set of interactions from users to items with information enough for us to be able to add context and item metadata.

We decided to use a dataset provided by GroupLens: MovieLens 100k. It is composed by 100000 real ratings from users to movies. It includes approximately 1000 users and 1700 movies.

This dataset satisfies some very relevant properties: all movies have received ratings and all users have rated at least one movie. Entities with few interactions may be counter-productive while training our model, so we will filter them out in advance. This will be detailed in the next section.

We will extend both datasets by using the API provided by MovieDB, an online database with information on movies and series. We are using this tool to add the actors that take part in each movie seen in the interactions. We will also filter out the actors that do not participate in less than 10 movies within the dataset because they are not relevant enough for them to be considered. We will rank the remaining actors and add 3 flags to every movie that will represent if the highest ranked actor in the movie is either a top 10 actor, a top 25 actor or not. This information will be searched in the API by using the movie title and the movie release year that are included in the datasets. In the original datasets we can also

find which genres are assigned to every movie. Both actors and genres will be added to the model as side-information and will be included in the X feature matrix of our graph.

We also need to define a context that can be determined with the available information. Since MovieLens includes a timestamp along with each interaction, we can identify the chronological order in which a user has interacted with the movies they have rated. Therefore, the immediate previous movie that the user has rated is a property that depends solely on the context of the interaction. It definitely is not a property of the user or the item involved in said interaction. In consequence, we can add the "last clicked item" to the graph as context. This specific context presents the advantage of having limited possible values: the movies included in the dataset.

Even though the context values are entities that are already included in our graph, we should make clear that we are differentiating between the movie interacted with and the movie that is used as context. Therefore, as all movies have been interacted with, we should have two nodes per movie in our graph: one that represents the movie as item and one that is used as context node. We must not reuse the item nodes for context as we would not be checking if our extension of message passing through several nodes is working correctly.

4.2 Methodology

Before showing the results of experiments, we should specify how we treat the data during the whole process and in which configuration we are running our models. The default settings explained in this section are generally customizable through program parameters.

First of all, it is a good and a common practice to clean our data before using it. This process receives the name of **preprocessing**. We could choose to work with the whole dataset, but it is certainly better to reduce the number of entities that do not provide enough information: this why we choose to filter out users and movies with less than 10 interactions and also actors with less than 10 appearances.

Next, it is important to point out that we need to **binarize** the ratings observed in the dataset due to our decision of working with implicit feedback: a user has either watched or not an item. In recommendation problems it is generally needed to define a threshold that divides the rating space in 'bad' and 'good' ratings. We decided to consider all ratings positive due to the following reasoning: a user needs to watch a movie to rate it, and if such user ever, at some point, decided

to watch a specific movie rather than other ones, it means they were interested enough to invest time watching it. This also implies that the user preferred a specific movie over the rest, even if they regret watching it afterwards. We are trying to predict which movie the user will be inclined to watch next time, not taking into account whether they will like it or not. This interpretation of the data is also linked to our choice of the used Loss Function.

After binarizing our data we might **split** it. When models are trained models with observed data, it is essential not to use the whole dataset to train it. Some of the interactions must be kept apart with the intent to validate the training and then to test it and check its performance. The interactions used for validation and testing must not be used during the training stages. There are several kinds of splits available. The *leave one out* splitting strategy (loo) consists in keeping one interaction apart for testing per user. In our case, we will leave one interaction for testing, another one for validation and the rest for training. For this reason, we need each user to have at least 3 interactions. Analyzing our recommendation problem more deeply, we are trying to figure out what we should recommend the user to watch next. Knowing that our chosen dataset provides a timestamp next to every interaction, we could choose to position ourselves 2 ratings behind for each user. This way, we will get to know if our model would have recommended the users the next 2 movies they actually watched. This method receives the name of **time-aware leave one out** (tloo) and it consists in leaving each user's last interaction for testing, and the one right before for validation.

So far, our dataset is only composed by positive interactions due to our of implicit feedback and our interpretation of the available data. We need to generate negative interactions through **negative sampling** in order to train the models correctly. For each movie a user has seen, we will add to the training set a definite number of pairs user-item with movies the user has not watched yet (with target 0). If we are working with context, the new interactions will have the same context as the original interaction that actually took place. This represents that the user chose the movie they interacted with over the rest in that definite context.

During training, as explained in section 2.3, the descent gradient method needs an objective function to minimize, called **Loss Function**, that represents the error between the predictions on the training set and this set's targets. As stated before, we chose to use the **Bayesian Probabilistic Ranking** Loss Function (known as $BPR - Opt$) defined in [13] and found in 2.6. This Loss Function is especially adequate for recommendation problems. The observed data in this loss function is input in pairs: $\hat{x}_{uij}$ represents that the user u preferred the item i over j (that is, u has watched i but not j). As stated in [13], this approach helps to prevent

overfitting and allows the model to learn rankings instead of assigning zeros and ones to all the items. This loss function is also used during validation and tuning while evaluating our models.

After the training is done, we should evaluate the model performances using the test set. In order to compute some kind of score to represent how well our model works after training, we must use **metrics**. It is convenient to point out that each user will have a test set with one and only positive interaction (called ground truth item) and several (by default, 99) negative samples. We chose to calculate the following metrics:

- **HR@k:** checks whether the positive interaction is among the top k scored interactions from the test set by the trained model. Then we calculate the ratio of test sets in which the ground truth item was part of the top rated samples.
- **NDCG@k:** gives a vision of which position the ground truth item is at in the top k ranked interactions from the test set, giving higher scores to higher positions. It follows the next equation:

$$1/\log_2(2 + index) \tag{4.1}$$

  Where *index* is the position of the ground truth item.

Before exposing the results we obtained, we are going to explain how to configure some of the models' settings. [18] questions the techniques that have been generally used to evaluate and compare models and recommender system algorithms. Since we aim to compare the performance of our extended models to other traditional models, we must make sure those comparisons are fair.

We will take for granted that Recommender Systems are used looking for the best performance and results, so we will evaluate each model in their own optimal settings. To do so, we first need to find said configuration.

In order to achieve this, we will use a **Bayesian Optimizer** on parameters such as the batch size used, the number of factors in embedding vectors, the number of epochs the model will be trained for and the model's learning rate.

The **Bayesian Optimization method** [4] is based on Sequential Model-Based Optimization (SMBO). It minimizes an objective function by building a surrogate function, a probability model based on past evaluation results of the objective function. In other words, it decides which values it will try next depending on the result of past choices using a probabilistic model. In order to carry out a Bayesian Optimization we need the following 4 elements:

- The objective function we are trying to minimize. It could be an error, a loss function, or a metric. Of course, in the case of metrics, the intention is to maximize them as it means we are getting better results. To achieve this: the objective function should be the opposite value of said metric, in order to minimize it towards minus infinity (which, of course, should never be reached).
- The domain space we are looking for the best values in. This is the domain space we will evaluate for every model:

setting	possible values
batch size	256, 512
number of factors	16, 32, 64, 128
number of epochs	10, 20, 40, 50, 70, 100, 120, 150, 180, 200
learning rate	0.05, 0.01, 0.001

- The optimization method previously explained, SMBO.
- The results obtained and a log of evaluations called trials.

Once we obtain these results, our models will always be run under the settings suggested by the optimizer.

4.3 Results

In this section, we will expose the results obtained from running our different models with the intention to confirm that supporting Recommender Systems with context and side-information leads to a better performance. We will measure the results given by:

- A baseline implementation of both Matrix Factorization and Factorization Machine.
- Matrix Factorization and Factorization Machine extended as models for Context-Aware Recommender Systems.
- Both Context-Aware Recommender Systems implemented with GCE.
- Context-Aware Matrix Factorization and Factorization Machine with Graph Convolutional Embeddings with genres added as side-information.

- Context-Aware Matrix Factorization and Factorization Machine with Graph Convolutional Embeddings with genres and actors added as side-information.

All the models will be evaluated by the metrics HR@10, HR@20, NDGC@10 and NDGC@20.

First of all, we will expose the results obtained with Matrix Factorization:

Matrix Factorization: MovieLens 100k Results

Context	GCE	Genres	Actors	HR@10	HR@20	NDGC@10	NDGC@20
No	No	No	No	0.6034	0.7667	0.3400	0.3814
Yes	No	No	No	0.5557	0.7147	0.3190	0.3586
Yes	Yes	No	No	0.7264	0.8664	0.4430	0.4758
Yes	Yes	Yes	No	0.7275	0.8643	0.4431	0.4710
Yes	Yes	Yes	Yes	0.7126	0.8579	0.4371	0.4750

This table shows, for each Matrix Factorization implementation, the best value for each metric value obtained throughout all the epochs. All models are configured with their optimal settings determined by the Bayesian Optimization.

Matrix Factorization seems not to be integrating context properly in its predictions if Graph Convolutional Embeddings are not included. However, the inclusion of GCE shows a great improvement in all metrics, adding at least a 10% more accuracy in both Hit Ratio metrics and around 10% more quality in ranking (NDGC).

Nonetheless, adding genres as side-information seems to have little to no effect on the results obtained by Context-Aware Matrix Factorization with GCE. The addition of actors to the model did not have the impact we expected: it causes worse Hit Ratio values, but seems to mantain the ranking quality.

Regarding the plotting of loss functions and metrics, we will always show the evolution for 100 epochs even though the Bayesian Optimization may have set another number of epochs to run. This way, we can analyze the behavior of their values after reaching their best values.

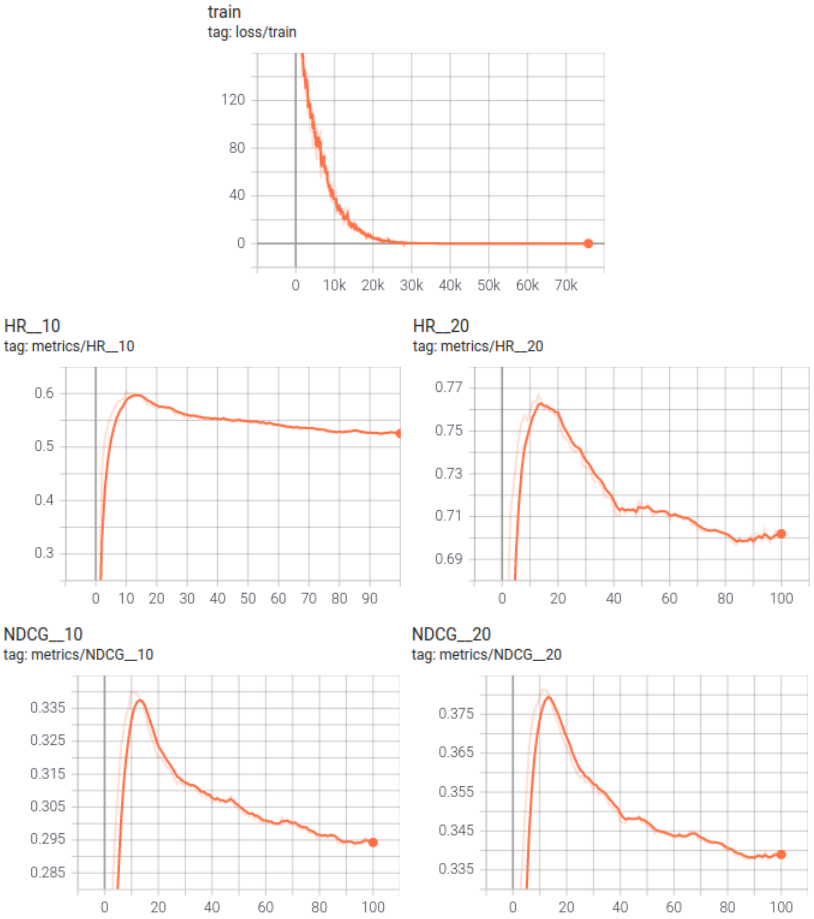


Figure 4.1: Loss Function, HR and NDCG values through epochs for Matrix Factorization with no context, GCE or side-information.

The Loss Function values are calculated once per observed data input to the model in every epoch, while the metrics are only calculated at the end of each epoch.

We can see that the Loss Function is early minimized and the metrics reach their highest values in the early epochs of the training. We can see how there generally is a drop in metrics after reaching the peak, except for HR@10, which shows a much more stable behavior.

This drop may be caused by several reasons:

- **Model settings**: our settings were chosen by the Bayesian Optimizer in order to obtain the best result in a specific number of epochs. This does not

guarantee any kind of stability after the peak is reached if we increment the number of epochs. An incorrect learning value rate may provoke this kind of drop. We can always choose to sacrifice some accuracy or ranking in exchange for a greater stability, but this was not the objective of our work.

- **Overfitting**: As explained in [19], a model may still be improving the Loss Function values while iterating over the training set while the values obtained through metrics using the validation set suddenly start getting worse. This phenomenon is known as *overfitting* and it represents that the model has stopped learning to generalize from the training data and, instead, it started to memorizing it. This would be prevented by the epoch number defined by the Bayesian Optimization or by an early stopping system.

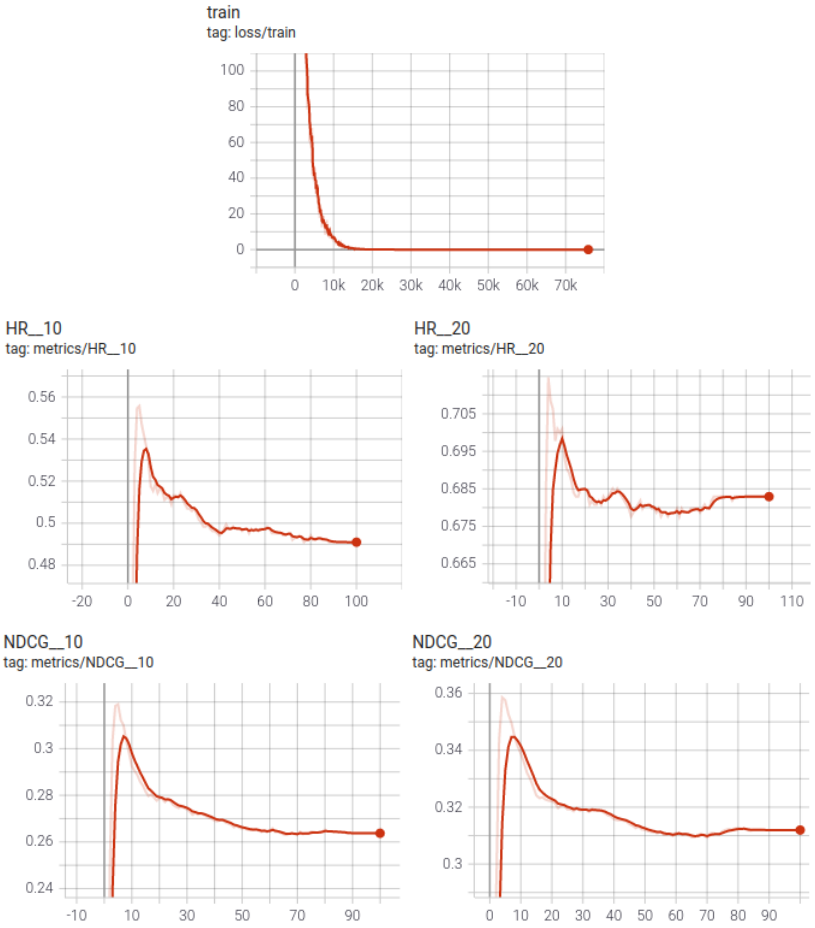


Figure 4.2: Loss Function, HR and NDCG values through epochs for Matrix Factorization with context, but no GCE or side-information.

By adding context to our model, the Loss Function is minimized faster, although the results obtained are worse. However, drops are less drastic except in HR@10 where the stability is lost.

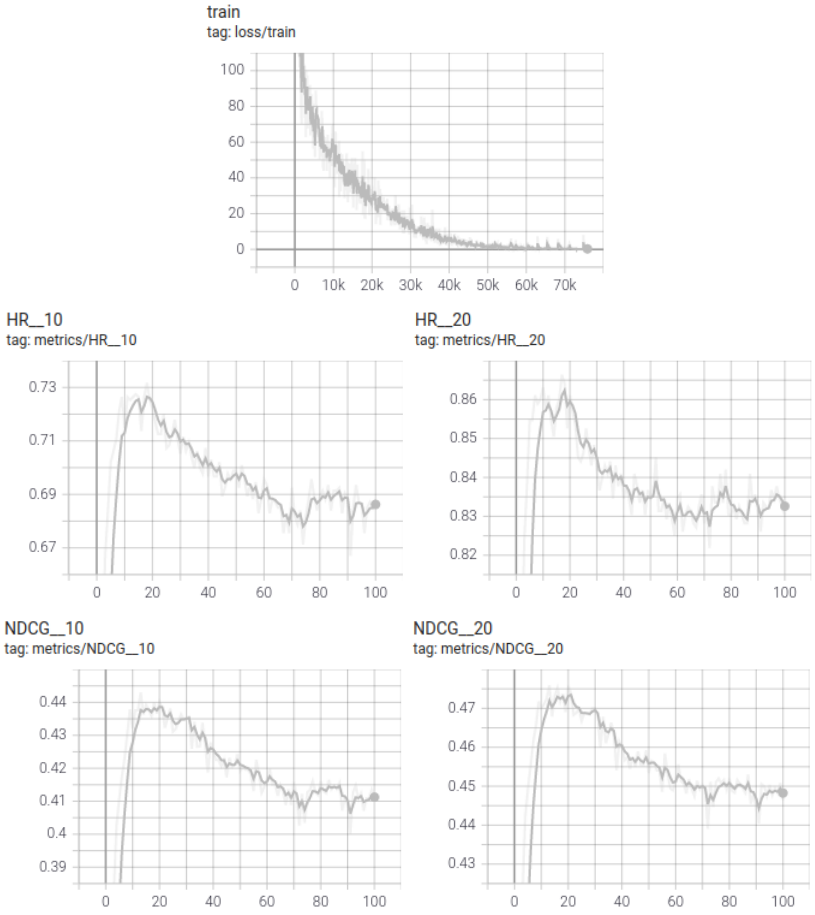


Figure 4.3: Loss Function, HR and NDCG values through epochs for Matrix Factorization with context and GCE but no side-information.

Changing the regular embeddings by GCE has several effects on the model's performance. The Loss Function takes many more iterations to be minimized. The best results are obtained way before the Loss Function takes values close to 0. It should be reminded that the results are much better than in the previous two implementations. Finally, the drop does not show a significant change in size respect to the last experiment, but the general behavior definitely shows less stability than before.

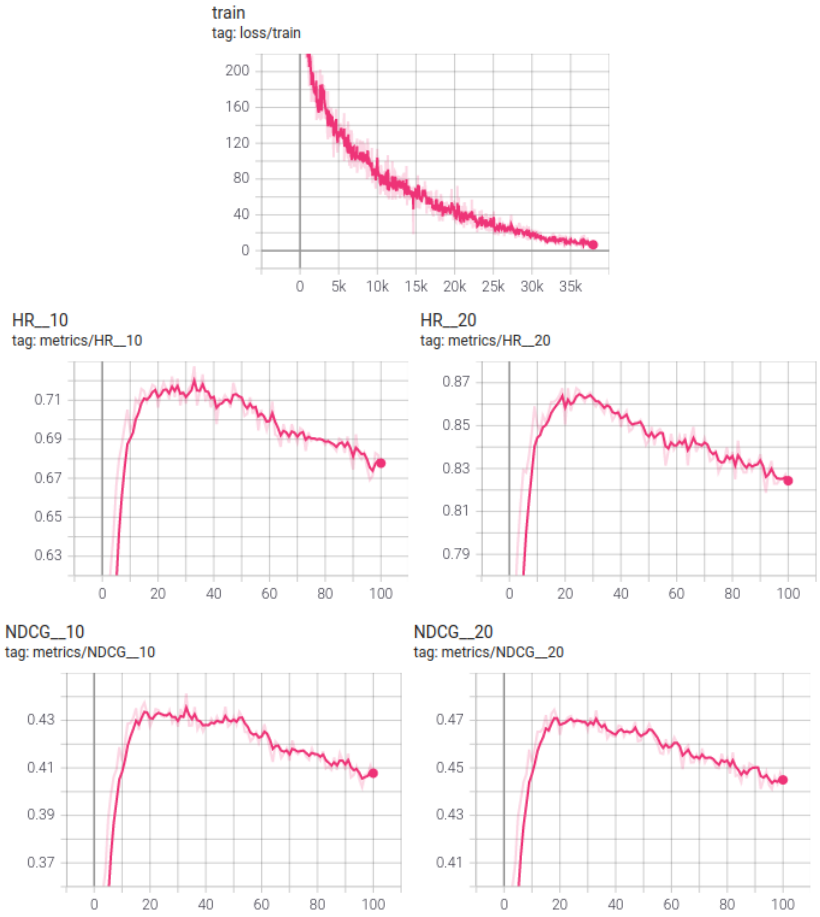


Figure 4.4: Loss function, HR and NDCG values through epochs for Matrix Factorization with context, GCE and side-information (genre).

After adding side-information to our model, the Loss Function barely approaches zero values after 100 epochs. Even though the results may not improve much with this addition, we can see how instead of a sudden drop after reaching the best values there is a gentle descent in the metric values. This is a clear symptom of overfitting rather than one of incorrect model settings for 100 epochs.

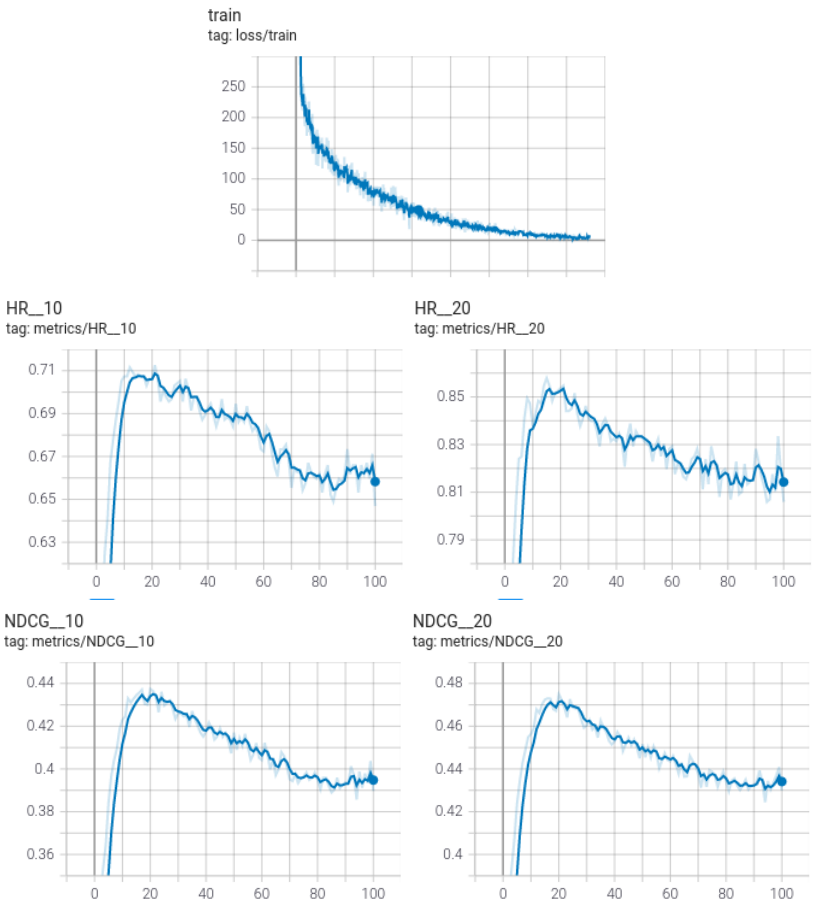


Figure 4.5: Loss function, HR and NDCG values through epochs for Matrix Factorization with context, GCE and side-information (genre and actors).

The addition of actors as side-information did not improve the results as we would have expected. However, the minimization of the Loss Function is achieved earlier. Also, the drop in metric values after the peak is reached is more similar to the ones detected before adding genre information.

It is clear that this model is not integrating this kind of information well into its predictions. Therefore, we should consider getting rid of it or trying to input it in a different data structure (for example, inputting it in the graph instead of as side-information).

And now, we will perform the experiments same with Factorization Machine. This is the summarized result table for this model:

Factorization Machine: MovieLens 100k Results

Context	GCE	Genres	Actors	HR@10	HR@20	NDGC@10	NDGC@20
No	No	No	No	0.5907	0.7805	0.3399	0.3867
Yes	No	No	No	0.6935	0.8275	0.4095	0.4429
Yes	Yes	No	No	0.7253	0.8632	0.4398	0.4730
Yes	Yes	Yes	No	0.7349	0.8600	0.4553	0.4868
Yes	Yes	Yes	Yes	0.7328	0.8696	0.4499	0.4855

In this case, it can be concluded that each addition improves the results noticeably, except for the actor addition. Hit Ratio adds either 14% or 8% to its score from the first experiment to the last one, while NDCG presents at least 10% more quality.

In this model, adding actors as side-information does not improve the results of the experiment with genres as the only kind of side-information. However, actors do not affect the result enough for them to be worse than those obtained without any kind of side-information, which happened with Matrix Factorization.

Also, we should remark that the results obtained with Factorization Machine are close to those obtained by Matrix Factorization (and generally a little better), except for the versions with context and regular embeddings: in this case, Factorization Machines clearly outperform Matrix Factorizations.

To finish, we will analyze the behavior of Loss Function and metric values as we did with Matrix Factorization.

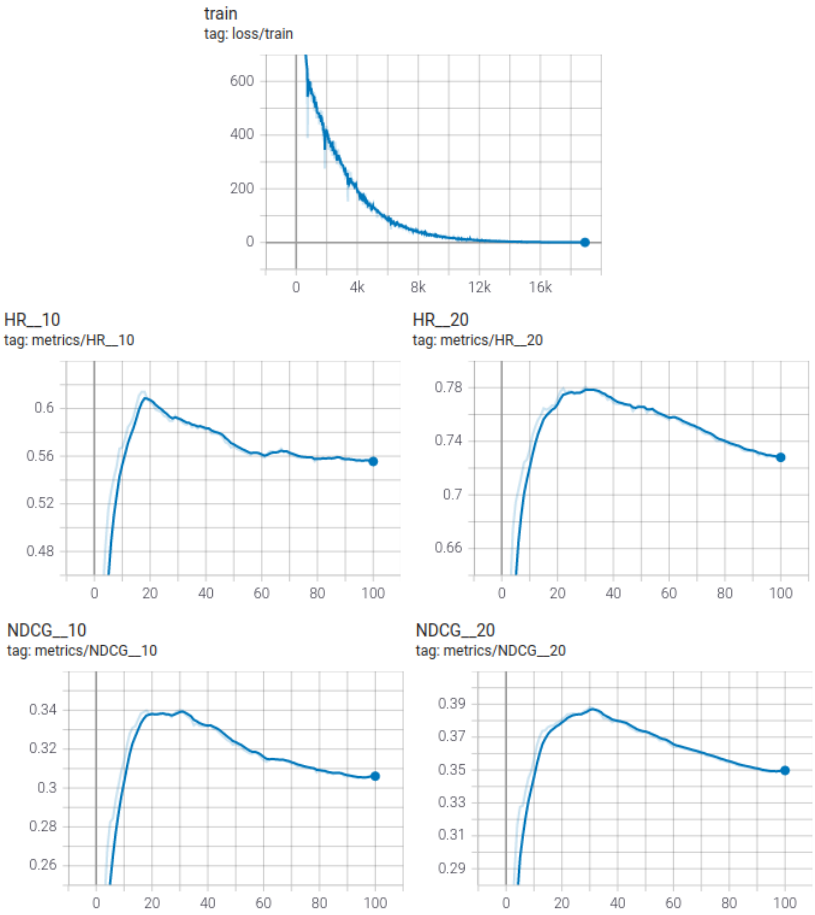


Figure 4.6: Loss Function, HR and NDCG values through epochs for Factorization Machine with no context, GCE or side-information.

Beginning with the Factorization Machine baseline implementation, it can be observed that the Loss Function is minimized in the early stages of the training process. Compared to the same version of Matrix Factorization, the metrics have less steep drops now, showing clearer signs of overfitting.

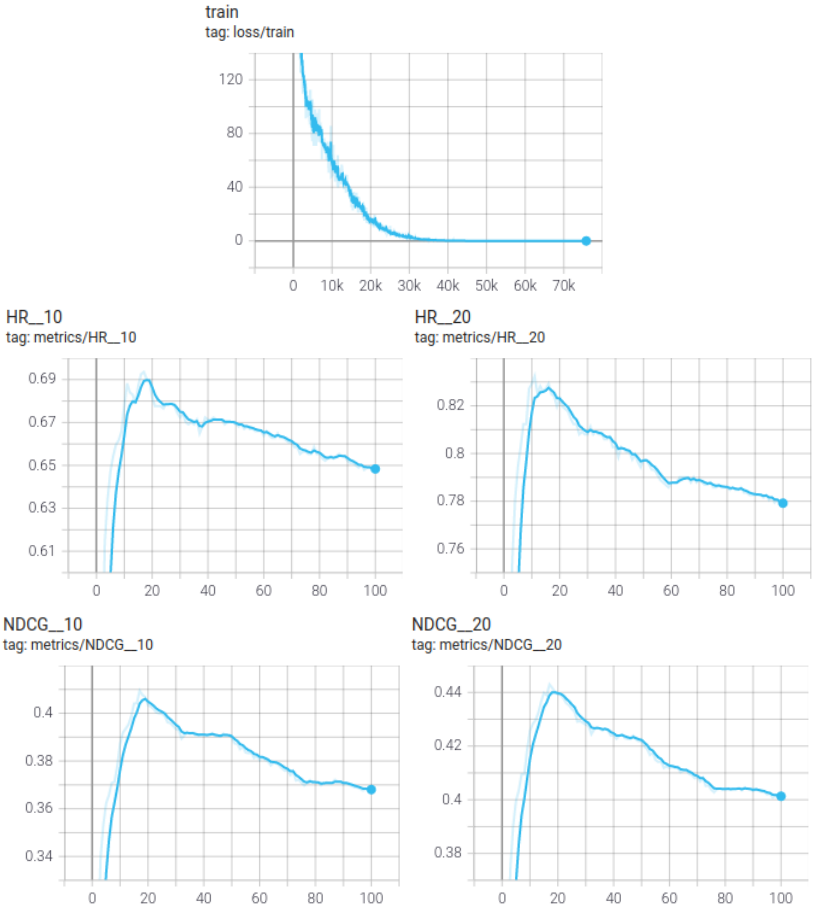


Figure 4.7: Loss Function, HR and NDCG values through epochs for Factorization Machine with context, but no GCE or side-information.

Adding context to Factorization Machines shows better results than doing the same to Matrix Factorization. However, in this model, the Loss Function takes more effort to be minimized. This minimization process is also longer compared to the previous implementation of this model.

Metrics show a similar behavior as the one seen in Factorization Machines without context. It may be said that the descent is steeper. Compared to Matrix Factorization, the peak is much higher in value and the drop is gentler.

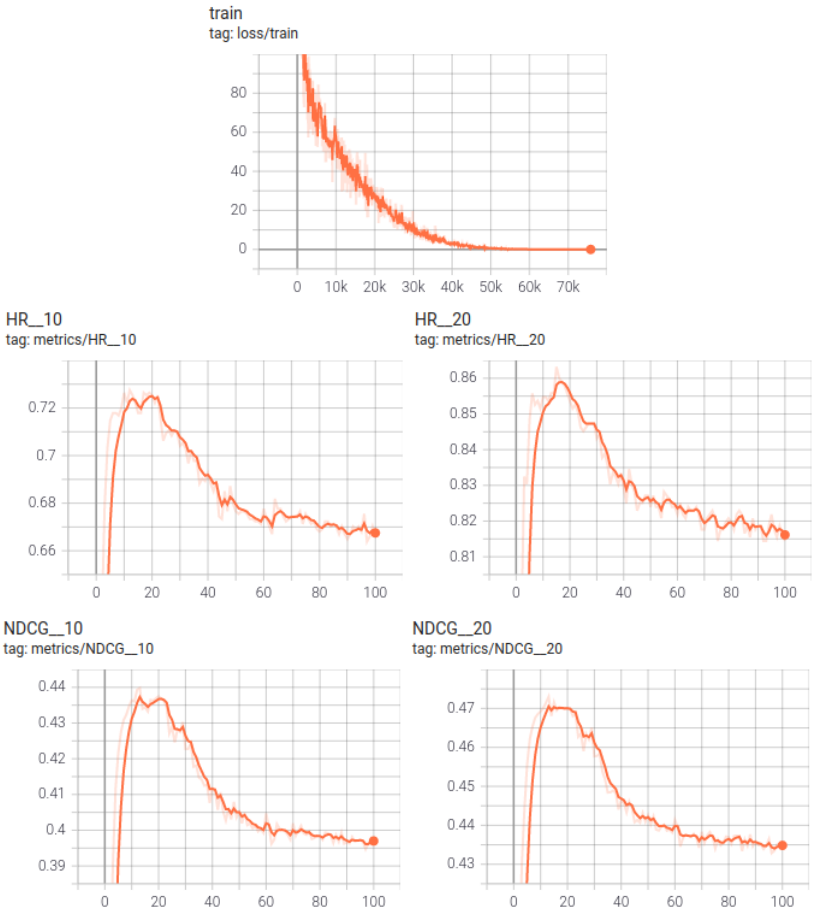


Figure 4.8: Loss Function, HR and NDCG values through epochs for Factorization Machine with context and GCE but no side-information.

Adding Graph Convolution Embeddings to Factorization Machines has the same effect on Loss Function's behaviour as adding them to Matrix Factorization: the minimization process takes longer, around the half of total iterations. In this case, the best results are also obtained before the Loss Function gets close to 0.

The drop in metrics after reaching the best results is more significant than in the last experiment. This difference is more noticeable than the one found between the same experiments with Matrix Factorization, principally because of the poor performance obtained from Matrix Factorization with context and without Graph Convolutional Embeddings.

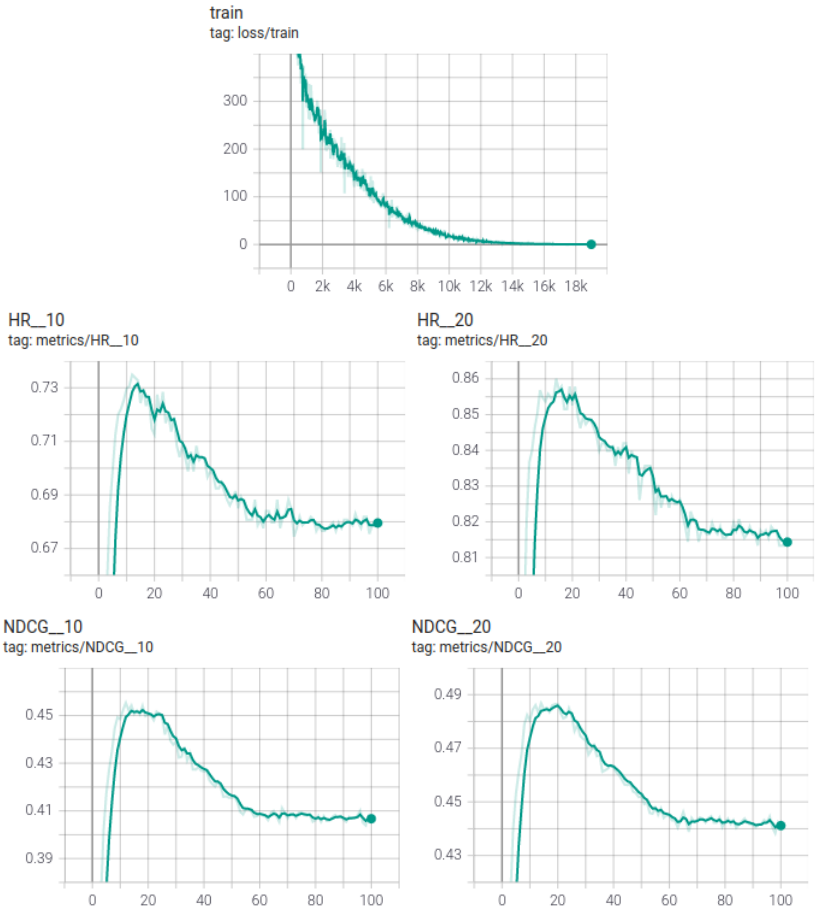


Figure 4.9: Loss Function, HR and NDCG values through epochs for Factorization Machine with context, GCE and side-information (genres).

Finally, adding side-information to Factorization Machines showed a slight improvement in performances that could not be shown in Matrix Factorzation. There is not a great change in the Loss Function comparing to the previous experiment. However, we should remind that the Loss Function in the same experiment with Matrix Factorization did not reach values close to 0 until the final iterations.

In terms of metric behavior, we do not detect the additional stability that adding side-information to Matrix Factorization showed. In fact, this behavior is almost identical to the one seen in the previous experiment.

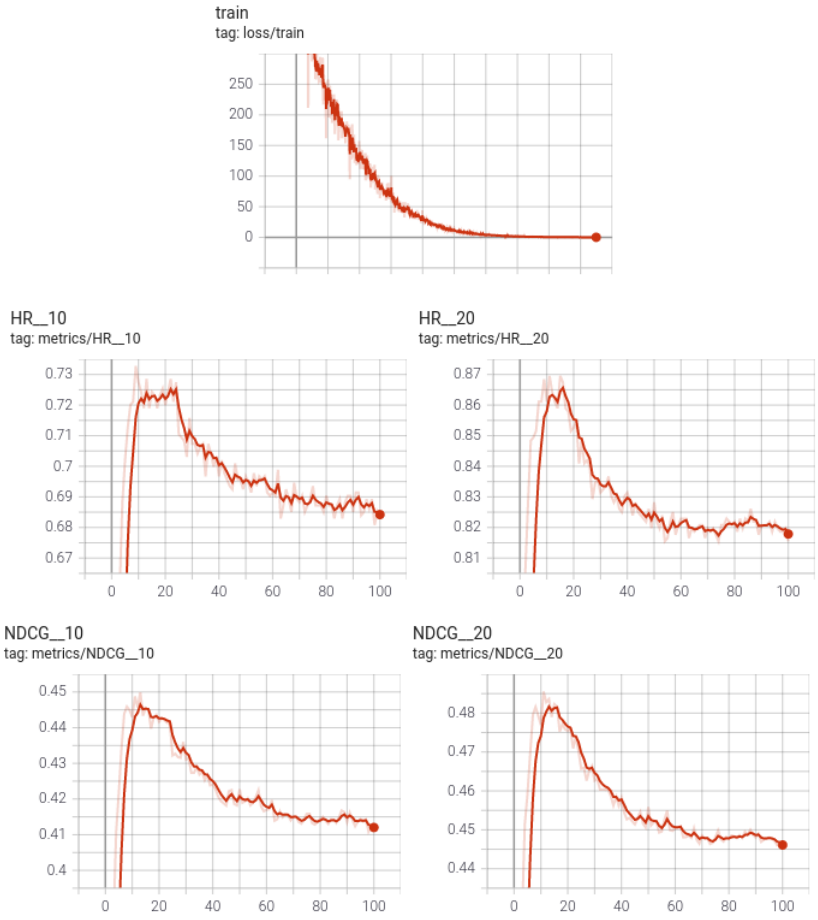


Figure 4.10: Loss Function, HR and NDCG values through epochs for Factorization Machine with context, GCE and side-information (genres and actors).

Adding actors as side-information to the latest experiment does not change the Loss Function's and metric's behavior noticeably enough. It should be remarked though that this addition did not improve the results as we expected.

To sum up, the introduction of Graph Convolutional Embeddings into these two models show a remarkable improvement in the results obtained. Factorization Machines and Matrix Factorization showed similar performance in terms of results, which were only slightly better using Factorization Machines, except for when context information is provided and regular embeddings are used: in this case, Factorization Machines outperform Matrix Factorization by a greater gap.

Including movie genres did not influence the results obtained by Matrix Factorization, but it showed a change on the metrics behavior along the epochs. In

the case of Factorization Machines, this inclusion provoked a slight improvement in results.

Finally, adding the actors that take part in every movie as side-information did not improve the results in any of the models. In fact, the results were worse in Matrix Factorization. Models did not incorporate this information in a good manner. We should research more in order to decide whether this information should be included in our model with a different data structure or this information is not relevant enough for users in order to watch a movie and therefore, we should skip adding it to the models. We could try adding certain item metadata in the graph structure we already defined: the actors would be a new entity in the graph and their nodes would be connected to the item nodes that represent the movies they take part in.

Chapter 5

Conclusion

The goals of our work included learning about some traditional Recommender System models and understand their behavior, detailing how they could be extended by introducing Graph Convolutional Networks in order to include information about the context interactions between users and items take place in and compare the results obtained with all the models defined during the work using fair settings for all of them, making sure each model is working at its full potential.

We fulfilled our first goal by starting our work detailing the concepts of Matrix Factorization and Factorization Machine including what they consist in, the data structures they work with, how they are trained and the possibilities they offer. Afterwards, we explained what Graph Convolutional Networks are, how they are defined, what information they need, how they generally work and the purpose behind the usage of them in Recommender Systems.

After quickly mentioning how to integrate them into Factorization Machines, we explained in Chapter 3 how to implement this integration and specified the GCN's behaviour when working with interactions that include context.

Finally, we described how we treated our data and several other factors such as preprocessing, data splitting, sampling, data representation, loss function and metrics, and the reasoning behind this decisions. Also, we explained how we looked for the optimal settings for each model in order to compare the best results each one of them could compute, which was one of our initial intentions described in the first paragraph.

By comparing the results we obtained from all the models, we can conclude that including context does in fact improve the model's performance. We also used this opportunity to check whether adding side-information related to the entities involved in the recommendation problem is also productive in the sense of better

results. In this case, we will have to conclude that it that depends on the model that is being used and the kind of data that is being included.

In future work, we plan to integrate the GCN layer into other models and evaluate the results obtained from these new model implementations. Also, we should define a standardized way of inputting datasets with context to our models, instead of defining methods that are too dependent on the way datasets are presented. We would also aspire to add more datasets to the result table and set more parameters using the Bayesian Optimizer. Finally, we would evaluate other ways of including item metadata into our models, such as adding actors as a new entity in the graph structure. This would imply that every actor would be represented by a new node in the graph which would be connected to the nodes of every movie that actor plays in.

Bibliography

[1] Ansari, Asim & Essegaier, Skander & Kohli, Rajeev. (2000). Internet Recommendation Systems. Journal of Marketing Research - J MARKET RES-CHICAGO. 37. 363-375. 10.1509/jmkr.37.3.363.18779.

[2] Zhu, Y., J. Lin, Shibi He, Beidou Wang, Ziyu Guan, H. Liu and Deng Cai. "Addressing the Item Cold-Start Problem by Attribute-Driven Active Learning." *IEEE Transactions on Knowledge and Data Engineering* 32 (2020): 631-644.

[3] Gediminas Adomavicius and Alexander Tuzhilin. Context-aware recommender systems. In *Recommender systems handbook*, pages 217-253. Springer, 2011.

[4] Hyperparameter Optimization for Machine Learning Models Based on Bayesian Optimization, Journal of Electronic Science and Technology, Volume 17, Issue 1, 2019, Pages 26-40, ISSN 1674-862X, https://doi.org/10.11989/JEST.1674-862X.80904120.

[5] Yiding Jiang et al. *NeurIPS 2020 Competition: Predicting Generalization in Deep Learning*. 2020. arXiv: 2012.07976 [cs.LG].

[6] Dheeraj Bokde, Sheetal Girase, Debajyoti Mukhopadhyay, Matrix Factorization Model in Collaborative Filtering Algorithms: A Survey, Procedia Computer Science, Volume 49, 2015, Pages 136-146, ISSN 1877-0509, https://doi.org/10.1016/j.procs.2015.04.237.

[7] Lin, Chih-Jen. (2007). Projected Gradient Methods for Non-Negative Matrix Factorization. Neural computation. 19. 2756-79. 10.1162/neco.2007.19.10.2756.

[8] (2009) *Embedding Space.* In: Li S.Z., Jain A. (eds) Encyclopedia of Biometrics. Springer, Boston, MA. https://doi.org/10.1007/978-0-387-73003-5_573

[9] S. Rendle, "Factorization Machines," 2010 *IEEE International Conference on Data Mining, Sydney, NSW*, 2010, pp. 995-1000, doi: 10.1109/ICDM.2010.127.

[10] Potdar, Kedar & Pardawala, Taher & Pai, Chinmay. (2017). A Comparative Study of Categorical Variable Encoding Techniques for Neural Network Classifiers. International Journal of Computer Applications. 175. 7-9. 10.5120/ijca2017915495.

[11] Gómez Duran, Paula (2020). Graph Convolutional Networks for context-aware recommender systems.

[12] Ruder, Sebastian. (2016). An overview of gradient descent optimization algorithms.

[13] Rendle, Steffen & Freudenthaler, Christoph & Gantner, Zeno & Schmidt-Thieme, Lars. (2012). BPR: Bayesian Personalized Ranking from Implicit Feedback. Proceedings of the 25th Conference on Uncertainty in Artificial Intelligence, UAI 2009.

[14] Q. Wang, Z. Mao, B. Wang, and L. Guo, "Knowledge graph embedding: A survey of approaches and applications," *IEEE TKDE*, vol. 29, no. 12, pp. 2724-2743, 2017

[15] Rianne van den Berg, Thomas N Kipf, and Max Welling. Graph convolutional matrix completion. *arXiv preprint arXiv:1706.02263, 2017*.

[16] Raschka, Sebastian & Patterson, Joshua & Nolet, Corey. (2020). Machine Learning in Python: Main developments and technology trends in data science, machine learning, and artificial intelligence.

[17] PyTorch Geometric Documentation https://pytorch-geometric.readthedocs.io/en/latest/

[18] Zhu Sun, Di Yu, Hui Fang, Jie Yang, Xinghua Qu, Jie Zhang, and Cong Geng. 2020. Are We Evaluating Rigorously? Benchmarking Recommendation for Reproducible Evaluation and Fair Comparison. In Fourteenth ACM Conference on Recommender Systems (RecSys '20). Association for Computing Machinery, New York, NY, USA, 23-32. DOI:https://doi.org/10.1145/3383313.3412489

[19] Ying, Xue. (2019). An Overview of Overfitting and its Solutions. Journal of Physics: Conference Series. 1168. 022022. 10.1088/1742-6596/1168/2/022022.