UNIVERSIDADE DE LISBOA

FACULDADE DE CIÊNCIAS

DEPARTAMENTO DE INFORMÁTICA

Recommender Systems Based on Deep Learning Techniques

Fábio Iúri Gaspar Colaço

Mestrado em Ciência de Dados

Dissertação orientada por:

Professor Doutor Francisco José Moreira Couto

2020

Acknowledgements

I would like to thank my teacher and supervisor, Professor Francisco Couto, for allowing me to work on such an interesting topic, and for all the help and support throughout the year. I would also like to extend my gratitude to Márcia Barros, for providing useful insights and support. To my father, Fernando Colaço, for teaching me that remaining calm is the best way to go through hard challenges; to my mother, Carla Gaspar, for always believing in my capacities to achieve more; and to my partner, Ana Gonçalves, for providing me with the emotional support, unconditional love, and helpful suggestions on everything that I do.

Resumo

O atual aumento do número de opções disponíveis aquando a tomada de uma decisão, faz com que vários indivíduos se sintam sobrecarregados, o que origina experiências de utilização frustrantes e demoradas. Sistemas de Recomendação são ferramentas fundamentais para a mitigação deste acontecimento, ao remover certas alternativas que provavelmente serão irrelevantes para cada indivíduo. Desenvolver estes sistemas apresenta vários desafios, tornando-se assim uma tarefa de difícil realização. Para tal, vários sistemas (frameworks) para facilitar estes desenvolvimentos foram propostos, ajudando assim a reduzir os custos de desenvolvimento, através da oferta de ferramentas reutilizáveis, tal como implementações de estratégias comuns e modelos populares. Contudo, ainda é difícil encontrar um sistema (framework) que também ofereça uma abstração completa na conversão de conjuntos de dados, suporte para abordagens baseadas em aprendizagem profunda, modelos extensíveis, e avaliações reproduzíveis.

Este trabalho introduz o DRecPy, um novo sistema (framework) que oferece vários módulos para evitar trabalho de desenvolvimento repetitivo, mas também para auxiliar os praticantes nos desafios mencionados anteriormente. O DRecPy contém módulos para lidar com: tarefas de carregar e converter conjuntos de dados; divisão de conjuntos de dados para treino, validação e teste de modelos; amostragem de pontos de dados através de estratégias distintas; criação de sistemas de recomendação complexos e extensíveis, ao seguir uma estrutura de modelo definida mas flexível; juntamente com vários processos de avaliação que originam resultados determinísticos por padrão.

Para avaliar este novo sistema (framework), a sua consistência é analisada através da comparação dos resultados produzidos, com os resultados publicados na literatura. Para mostrar que o DRecPy pode ser uma ferramenta valiosa para a comunidade de sistemas de recomendação, várias características são também avaliadas e comparadas com ferramentas existentes, tais como extensibilidade, reutilização e reprodutibilidade.

Palavras Chave: Ferramentas de Software, Sistemas de Recomendação, Aprendizagem Profunda, Avaliação, Implementação, Extensibilidade, Reprodutibilidade.

Abstract

The current increase in available options makes individuals feel overwhelmed whenever facing a decision, resulting in a frustrating and time-consuming user experience. Recommender systems are a fundamental tool to solve this issue, filtering out the options that are most likely to be irrelevant for each person. Developing these systems presents us with a vast number of challenges, making it a difficult task to accomplish. To this end, various frameworks to aid their development have been proposed, helping reducing development costs by offering reusable tools, as well as implementations of common strategies and popular models. However, it is still hard to find a framework that also provides full abstraction over data set conversion, support for deep learning-based approaches, extensible models, and reproducible evaluations.

This work introduces DRecPy, a novel framework that not only provides several modules to avoid repetitive development work, but also to assist practitioners with the above challenges. DRecPy contains modules to deal with: data set import and conversion tasks; splitting data sets for model training, validation, and testing; sampling data points using distinct strategies; creating extensible and complex recommenders, by following a defined but flexible model structure; together with many evaluation procedures that provide deterministic results by default.

To evaluate this new framework, its consistency is analyzed by comparing the results generated by DRecPy against the results published by others using the same algorithms. Also, to show that DRecPy can be a valuable tool for the recommender systems' community, several framework characteristics are evaluated and compared against existing tools, such as extensibility, reusability, and reproducibility.

Keywords: Software Frameworks, Recommender Systems, Deep Learning, Evaluation, Implementation, Extensibility, Reproducibility.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Motivation

With the ever-increasing number of available options when taking a certain decision, these processes are getting more complex and time-consuming than ever. If one decides to buy an item online and has no exact product model/variant in mind, it might be very frustrating for the user to browse across all the displayed options to try to find exactly what matches his/her taste. This is not only bad for the consumer, who gets tired and wastes unnecessary time, but also for the vendor, as the *interaction to checkout* ratio decreases when increasing the number of available products. On physical stores, where stock is limited and all the products are displayed in-place to the customer, this issue is reduced by decreasing the offer and maintaining only the most popular (i.e. most bought) products and combinations of products. Although, since there are no physical restrictions on online stores, we can choose to show different sets of items to distinct users. Essentially, a recommender system (RS) is a tool that suggests sets of items to each user, serving as a filter for uninteresting products, preferably tailored to each customer.

Various techniques can be used to develop a RS: based on popularity, using similarities between the user's previous purchases (and its relations with other users' preferences), by using some characteristics of the interacted items, etc. Their applicability is also very broad, since these systems can be applied, for example, to suggest products on e-commerce platforms Schafer et al. [1999], to enhance learning habits by recommending the correct learning resources for each learner Manouselis et al. [2011], to display custom sets of videos to each user Davidson et al. [2010], and to suggest new movies or shows Gomez-Uribe and Hunt [2016].

Moreover, a good RS should also leverage long-tail recommendations, by suggesting not only the most popular products, but also the niche products as targeted as possible, since they can grow to become a large percentage of the total sales (Figure 1.1). As a popular example, about 30-40% of Amazon[1] book sales come from niche books that can't be found easily on any physical book store. The value created

[1] https://www.amazon.com

from promoting and making these books accessible exceeds one billion dollars annually Brynjolfsson et al. [2006].

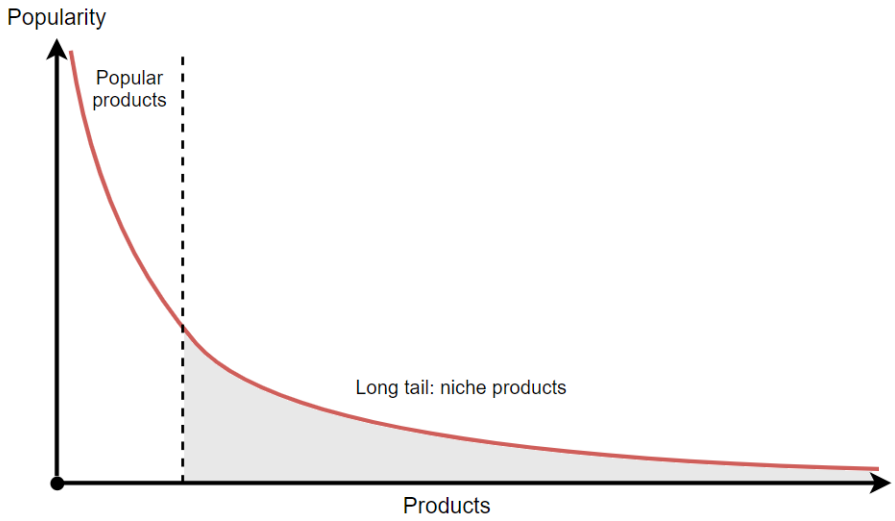


Figure 1.1: The Long-Tail Phenomena: Online stores may offer all the products to the customers, since they are not limited by the amount of items they can hold. These low-demand niche products sold in high quantities might account as being a very significant percentage of the total sales for a given business.

With the increase in the available computational power, training data, and its resulting trend on deep learning (DL) based techniques across all fields, it is without surprise that the number of publications of RSs based on these methods is almost doubling each year Batmaz et al. [2019], attracting the attention of both members of the academia and members of the industry. Although superior results are usually shown in multiple articles presenting these new models, there are various obstacles before applying them in practice:

1. A huge portion of papers does not provide any source code, making it hard to test the model (specially if a small but important concept was not well specified).
2. Distinct train-test splitting techniques are applied, using different frameworks, therefore achieving non-identical results.
3. Implementations using distinct languages (Python, Java, C++, JS, ...) or DL frameworks (PyTorch, Tensorflow, Keras, Chainer, ...) will most likely produce different outcomes.
4. Most of the times, in order to compare a new technique against others, it is required for the experimenter to implement the baseline algorithms, as well as the new proposed system.

When developing a RS, one also needs to think about various common problems: which structure to follow? How to deal with data sets containing invalid identifiers? What if the provided data set does not

fit in memory? How should I split my data set for training and testing purposes? How to share and deploy a trained model? How to quickly test my new model against other models in the same environment? It is clear that at the moment, any tool that simplifies these processes to avoid these issues as much as possible is welcome to the RSs community. For the industry, to facilitate the deployment of these systems; as for the academia, to allow researchers to speed up their creative process and to experiment new strategies faster and with more confidence, since ”developing a common evaluation framework” Beel et al. [2016] is very important for the future of this field.

1.2 Objectives

Existing frameworks to build RSs already speed up this process and tackle some of the issues mentioned in the previous section, but there are still difficulties that could be mitigated. To this end, the main objective of this work is to **create a modular and open-source Python framework to assist with the creation and evaluation of RSs** (with more emphasis on deep learning-based RSs).

All the developed work should aim to be extensible (other researchers should be able to contribute easily), use as much abstractions as possible (contributors should not require to understand how the specific data structures are stored), and should be flexible as well (if needed, contributors should be able to access all the properties of a RS).

Reproducibility is also a very important characteristic that should be leveraged by this framework. Therefore, it should be possible to run a training and/or testing procedure in a deterministic manner (even if randomness is applied - during initialization, sampling or splitting, there should be a feasible way to provide a seed value).

1.3 Methodology

Due to the complexity of the main objective, the applied methodology will be based on incrementally developing the various building blocks, so that at the end a complete framework is built:

1. Build a Data Set module, capable of representing in-memory as well as out-of-memory recommender data sets (i.e. in-disk), using the same interface. This will allow the following modules to deal with any data set in the same way, regardless of the size.

2. Develop a Sampler module, that should be able to use various sampling techniques on any data set. This will abstract generating training batches for model training.

3. Develop a Recommender module, that will allow easy and modular integrations of various recommenders, using a common interface and structure.

4. Build an Evaluation module, allowing practitioners and researchers to do comparative analysis among multiple RSs using the relevant metrics.

5. Implement some baseline RSs (non-DL based).

6. Implement state of the art RSs (DL-based).

1.4 Contributions

The main contribution of this work is a solution to allow faster and easier development of novel, extensible, and reproducible deep learning-based recommenders, and to improve the overall training and evaluation processes.

- **Open Source Software**: *DRecPy GitHub Repository*[2]
- **Published Paper**: *DRecPy: A Python Framework for Developing Deep Learning-Based Recommenders* Colaço et al. [2020], in the 14th ACM Conference on Recommender Systems (RecSys 2020) (Late-breaking results), pages 675-680, 2020.

1.5 Document Structure

Additionally to the current introductory chapter, this work is structured as follows:

- ***Chapter 2*** (Background) introduces the various related concepts such as Recommender Systems (RSs), with its distinct types and characteristics, popular data sets to train these systems and its formats, which evaluation metrics are commonly applied, and other important concepts such as Machine Learning (ML) and Deep Learning (DL).
- ***Chapter 3*** (Related Work) briefly describes popular frameworks that support the development of RSs, as well as some examples of DL-based RSs that will be implemented on this work.
- ***Chapter 4*** (DRecPy: A Deep Learning Recommendation Framework for Python) discusses and presents a framework for developing and evaluating deep learning-based recommenders. Various points related to how to build these systems will be discussed in-depth, and a discussion on how this framework deals with these challenges will be provided.
- ***Chapter 5*** (Evaluation) presents evidences for the consistency of the proposed framework and its built-in RSs, with the results found on the related literature. It also shows an analysis of a use-case that compares the extensibility and code reusability of the proposed framework against one of the related frameworks, followed by a final in-depth comparison of features offered by DRecPy and alternative tools.

[2] https://github.com/fabioiuri/DRecPy

- ***Chapter 6*** (Conclusion) briefly explores the main conclusions of this work and provides a few directions for future work.

Chapter 2

Background

This chapter will start by explaining some of the key concepts required to understand how systems that provide recommendations can be structured, followed by describing the types of learning agents that can be applied for these tasks.

2.1 Recommender Systems

A Recommender System (RS) is a system designed to suggest items to each user, and can also be seen as a filter that greatly reduces the number of items shown to the user. Preferably, a RS will be tailored to each user's taste and behavior, in a way that the shown products maximizes the user's interest. As previously mentioned, the provided suggestions relate to various decision-making processes, such as: what item to buy, what music to listen to next, what movie to watch, just to name a few. Some of the objectives of a RS are:

- ***Increase user satisfaction***: Without a RS, each user has to do their search, by browsing the multiple alternatives and by applying successive filters, until he/she is satisfied with the selected options. This isn't the desired process, although it still happens very often. In most cases, the user might get so frustrated that he/she gives up searching for the desired item altogether, making both the user and the vendor unhappy. By employing the help of RSs to filter out most of the uninteresting items to the user, and showing first those who are predicted to be the most successful for that particular case, the decision process is simplified, making the user satisfied to not have to take the frustration journey of choosing the *perfect* item without help. As it is common knowledge, satisfied users are faithful users, that is, users that come back for future purchases.

- ***Increase the number of sold products***: Despite RSs being a huge help by speeding up certain decision-making processes, thereby avoiding potential situations in which the users get too frustrated to continue, they also help by providing recommendations of items that the user might need at that moment but wasn't aware of. From showcasing interesting promotions to compelling prod-

uct combinations, therefore increasing the total number of sales. This effect is fairly easy to test for any online retailer, through online tests where a certain percentage of the traffic gets assistance from a RS, and the remaining does not.

- ***Increase diversity of sold products***: These tools also excel at promoting item-discovery. As discussed in section 1.1, a good RS should leverage long-tail recommendations (i.e. suggest not only the most popular items but also the niche products as targeted as possible). Items with low mainstream interest and a low number of sales might get a significant increase with an appropriate RS, diversifying the products sold by each vendor.

When developing a RS, there are several factors to take into account, such as: the type of available data; which feedback types are available; how do we build a model that can learn the available and valuable patterns; how should we provide recommendations; and what is the best way to evaluate our model once it is built. With so many challenges to be tackled, a huge research community from academia and industry as been formed around these systems, and a big number of interesting and powerful ideas have emerged. There are many basic concepts that one is required to understand to be able to wrap their heads around how these systems work, and who knows, to push the boundaries of RSs research even further.

2.1.1 Types of Recommenders

There are many ways to classify a RS, but here we'll discuss each type of recommender in terms of which techniques are used for their development, by presenting some advantages and disadvantages for each.

2.1.1.1 Knowledge-based (KB)

Knowledge-based recommenders are the most algorithmically basic systems, with this technique being the oldest one for building these systems. It is solely based on domain knowledge, where rules are built and later on assessed on certain contexts (i.e. for certain items or user behaviors), using case-based reasoning processes, as detailed in Martinez et al. [2008]. Nowadays these RSs are only used for complex domains, when no other method is easy to be applicable, or does not provide valuable results due to the lack of good data (e.g. domains where the number of user interactions are low, or when this data is very noisy). A simple design for these types of systems is shown in Figure 2.1.

Even though these methods are the most simple in terms of the applied algorithms, they demand the most manual work and maintenance among all listed types. From building rules to recommend certain items, adjusting these rules whenever the inventory increases in diversity, to tweaking them over time due to seasonality trends, these techniques become very hard to scale. Another bottleneck for KB RSs is that in some domains, hiring domain-experts to help to build the knowledge base is not cheap, so there will be infrastructure costs, engineering costs, but also an added cost for these domain-experts.

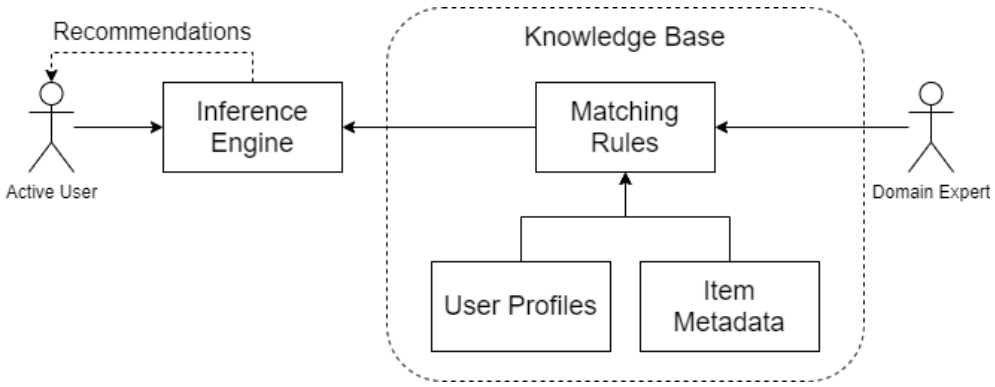


Figure 2.1: Knowledge-based Recommender: A system that depends on domain knowledge experts to build matching rules based on the existing user-profiles and item metadata.

Despite these disadvantages, these systems can provide instant results, since there is no training procedure or the requirement of acquiring data to give meaningful recommendations. They also don't suffer from the famous **cold-start** problem (i.e. not being able to infer rules or draw patterns due to missing data, as described in Safoury and Salah [2013]), since no rules are inferred on these systems. With KB-based recommenders, explanations on why certain items are being recommended to the user can be easily provided, due to the inference engine being deterministic in its use of the knowledge base.

2.1.1.2 Content-based Filtering (CB)

These RSs make their predictions based on various content features and their possible correlations, such as user characteristics (age, gender, demographics) and item features (color, size, brand, price). They are mainly based on the similarity of the content, so if a given user bought a book b_i, and a recommendation is requested to a RS, it could compare this book to every other book of the same genre and provide the one that is the most similar as the recommendation (i.e. the one that maximizes $sim(b_i, b_j)$), as shown in Figure 2.2. These systems work on the idea that if users consistently buy items that share some characteristics, then they are likely to continue to buy items sharing those features.

Despite item information being available from the start, CB systems might still suffer from a user cold start problem, that is, not having enough user information to classify whether a given item would be liked by the user or not. This mostly happens when new users join the platform. But after some interactions with the RS, the system is able to build a model of the user's preferences and provide effective recommendations. Often these approaches lead to over-specialized recommendations (Aggarwal [2016]), in which the system recommends the expected items, therefore producing little to no value. Despite these disadvantages, it is also possible to provide explainability on the recommendations generated by CB RSs.

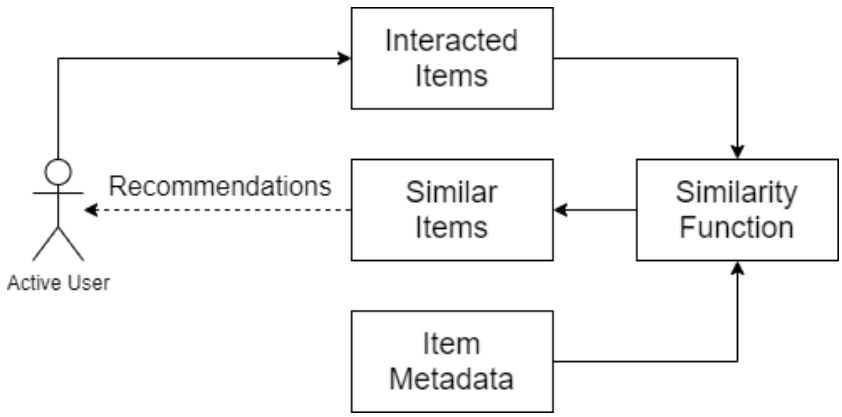


Figure 2.2: Content-based Recommender: The recommender should compute which items are the most similar with the ones previously interacted by the user, and then recommend them.

2.1.1.3 Collaborative Filtering (CF)

The previous technique is based on how to explore the user's preferences to find items that would be a good match, and collaborative filtering techniques take the idea of exploiting the available data even further, by detecting patterns on the distinct user's behaviors. But how can we know if a given user is similar to another user? A quick and naive approach would be to use the proportion of bought items that user u_i bought and that user u_j also bought. It makes sense in the way that each person has its own tastes, but it is very likely that there are also more people that share those tastes and interests, at least at some particular subject. Recommendations are then provided by the RS by taking into account the various bought items by the similar users, as depicted in Figure 2.3. This technique showed to significantly improve performance when valuable data is provided, relatively to all previously discussed types.

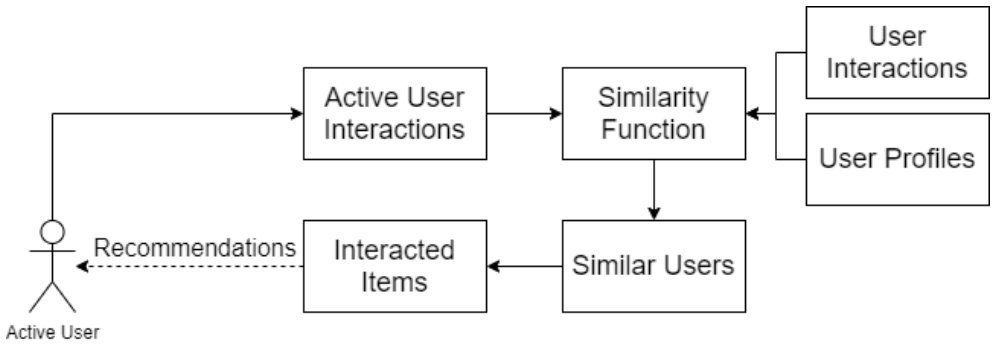


Figure 2.3: Collaborative Filtering Recommender: Similar users are computed by comparing the activity of each user to the activity of the active user. After similar users are retrieved, some items that were previously interacted by these users are recommended to the active user.

There are two main model designs used for CF based RSs:

- ***Memory-based***: These RSs are mainly based on computing similarities (item-to-item or user-to-user) and providing recommendations based on the neighbours of the target user. This obviously assumes that there is some distance metric/similarity function S that is able to capture some or most of the required relationships to provide valuable recommendations. A very popular example of such application that is broadly used as a baseline algorithm, is the user k-nearest neighbours, which for a target user u and a target item i, computes the user's k closest neighbours that have rated i and estimates the $r_{u,i}$ as a weighted average of each neighbour's rating by their respective similarity. One simple alternative would be to use the following function to estimate a rating:

$$r_{u,i} = \frac{1}{\sum_{u' \in P(u)} S(u,u')} \sum_{u' \in P(u)} S(u,u') \cdot r_{u',i}, \tag{2.1}$$

  where $P(u)$ is the set of the k-nearest users of u. Similarity functions are discussed in section 2.1.2. Despite these techniques being easy to understand and to be deployed in production systems, they have some pitfalls in terms of scalability and the computational power required is usually high (due to computing all similarity pairs). Explaining the results is usually not as easy as with traditional KB and CB methods, but this is still an achievable task.

- ***Model-based***: When more advanced techniques (i.e. machine learning / deep learning) are employed on collaborative recommendation tasks, these are called model-based CF RSs. The most traditional model-based techniques are the matrix factorization approaches, in which the main idea is to represent users and items in a lower dimensional space latent space (i.e. instead of each user being represented by a vector of size $|I|$, it is represented by a vector of size L, with $L << |I|$). The Regularized SVD (RSVG) Paterek [2007] is one example of this type of approach, and it makes predictions in the following manner:

$$r_{u,i} = \sum_{j=1}^{L} U_{u,j} V_{i,j}, \tag{2.2}$$

  where U and V are the user and item matrix respectively, and U_u is the vector of latent features for user u and V_i the vector of latent features for item i.
  Some deep learning-based work is also built around this idea, and further along this document, we'll discuss some of these existing approaches and how they work. Despite these techniques being very attractive due to their showcased performance, they are sometimes discarded because the produced recommendations are often very hard to explain.

As one can already expect, these methods suffer even more from the cold-start problem, since there must be enough interaction data to group users based on their behaviors (Burke [2000]). With large amounts of existing users and items, the sparsity of the data is expected to increase, making it harder for these algorithms to be useful - when that happens, the most used approach is to remove certain users and items with a low number of interactions (i.e. items that are almost never bought and users that almost never buy).

2.1.1.4 Hybrid

Modern proposals for RSs usually combine two types: CB and CF strategies. With CF methods being used to understand the collaborative patterns of the data and CB methods to provide mechanisms for assessing relevancies between items. In this way, we can help fight the cold-start problem of these systems, while being able to achieve even higher performant solutions at times Ma and Narayanaswamy [2018]. One thing to note is that in a production environment, it might become a more complex system to deploy. Since these systems use CF techniques, they might also not be able to provide explainable recommendations.

There are various hybridization techniques that were proposed and that indicate how to combine these different approaches Burke [2002], such as:

- ***Weighted***: Each recommendation approach is used to produce a score, and a final output is computed as a combination of these scores;
- ***Switching***: Certain situations trigger either one algorithm or the other, depending on which type is more suitable for that case;
- ***Mixed***: Recommendation lists are provided as a concatenation of the recommendations computed by each technique separately;
- ***Cascade***: Recommendation outputs from one approach are then refined by the other approach, by filtering out or discounting certain items depending on if they're recommended by the second approach or not;
- ***Feature combination***: Both approaches are combined in a single algorithm and its input is the feature combination of both;
- ***Feature augmentation***: Recommendation outputs from one approach are used as part of the input for the second algorithm.

2.1.2 Similarity Functions

Similarity functions are components widely used in CF-based recommenders and another information retrieval (IR) applications, and in the context of RSs, these are used to measure the similarity between two users or two items. The similarity value between similar vectors should approach 1, and for very distinct vectors it should approach 0 or -1, depending on the similarity measure used (e.g. the cosine similarity ranges from -1 to 1 whilst the Jaccard index ranges from 0 to 1). Although many of these functions range from -1 to 1, in practice, they usually vary from 0 to 1, since most data sets contain non-negative rating/interaction values.

Given the importance of computing similarities, this section will provide a brief description as well as the formulation for computing the similarity between two users using multiple popular functions (item similarity formulas can be easily derived from the ones shown here).

2.1.2.1 Cosine Similarity

Cosine similarity is one of the most used similarity functions, and it computes the cosine of the angle formed by two vectors, which in the RSs context, a vector contains the interaction values for each existing item (when no item exists, the interaction value is set to 0). If both vectors are equal, then the formed angle is $0°$, and the similarity value will be $cos(0°) = 1$; if the vectors form an angle of $90°$, the resulting similarity will be 0; and if the vectors are apposed, then their formed angle will be $180°$ and their similarity will be -1. This is described in Breese et al. [2013], with the following alternative, but equivalent formula:

$$COS(u, u') = \frac{\sum_{i \in I(u,u')} r_{u,i}\, r_{u',i}}{\sqrt{\sum_{i \in I(u)} r_{u,i}^2}\sqrt{\sum_{i \in I(u')} r_{u',i}^2}} \tag{2.3}$$

with $r_{u,i}$ being the interaction value/rating that the user u provided to the item i, $r_{u',i}$ the interaction value/rating that the user u' showed to the item i, $I(u)$ the set of items rated by u and $I(u, u')$ the set of co-rated items by u and u'.

There is also another cosine similarity formula that is widely applied on CF applications (Adomavicius and Tuzhilin [2005]), but generates different results:

$$COS_{CF}(u, u') = \frac{\sum_{i \in I(u,u')} r_{u,i}\, r_{u',i}}{\sqrt{\sum_{i \in I(u,u')} r_{u,i}^2}\sqrt{\sum_{i \in I(u,u')} r_{u',i}^2}} \tag{2.4}$$

The difference is very subtle, but it computes completely different values, as the denominator for this second formulation is much smaller than the first one, because it only takes into account the co-ratings between the two users ($I(u, u')$).

2.1.2.2 Adjusted-Cosine Similarity

The adjusted-cosine similarity is based on pre-processing the vectors before applying the normal cosine formula, so that the rating bias of each user does not impact the computed similarities. This is very effective because some users tend to rate most items with a very low score (users that are harder to satisfy) and others that do the exact opposite (users that are easily impressible). By applying this new method, we can compare users with very distinct rating scales without affecting the overall similarity quality.

$$ADJCOS(u, u') = \frac{\sum_{i \in I(u,u')} (r_{u,i} - \bar{r}_u)\,(r_{u',i} - \bar{r}_{u'})}{\sqrt{\sum_{i \in I(u,u')} (r_{u,i} - \bar{r}_u)^2}\sqrt{\sum_{i \in I(u,u')} (r_{u',i} - \bar{r}_{u'})^2}} \tag{2.5}$$

where the new variables $\bar{r}_u$ represent the average rating value for the user u, and $\bar{r}_{u'}$ the average rating value for the user u'.

2.1.2.3 Pearson Correlation Coefficient

The Pearson Correlation Coefficient gives the linear correlation between the two user vectors. When the computed result is 1, it means that the users have a positive linear correlation; when the result is -1, then the users have a negative linear correlation; and when the result is 0, the users have no linear correlation. This metric is also commonly found in the literature related to memory-based RSs.

$$PCC(u,u') = \frac{\sum_{i \in I(u,u')} (r_{u,i} - \bar{r_u})\,(r_{u',i} - \bar{r_{u'}})}{\sqrt{\sum_{i \in I(u,u')} (r_{u,i} - \bar{r_u})^2} \sqrt{\sum_{i \in I(u,u')} (r_{u',i} - \bar{r_{u'}})^2}}, \tag{2.6}$$

This formula is exactly the same as the one shown in equation 2.5, for the adjusted-cosine similarity, although there's a small but very important distinction: here $\bar{r}_u$ represents the average rating value for the user u on the items that were rated by both u and u', and $\bar{r}_{u'}$ the average rating value for the user u' also on the items that were rated by both u' and u. Formally:

$$\bar{r}_u = \frac{\sum_{i \in I(u,u')} r_{u,i}}{|I(u,u')|} \tag{2.7}$$

and

$$\bar{r}_{u'} = \frac{\sum_{i \in I(u,u')} r_{u',i}}{|I(u,u')|} \tag{2.8}$$

2.1.2.4 Jaccard Index

The Jaccard index does not take into account the interaction value/rating that the users provided, but instead, it only considers whether the users rated the item or not (binary value). Instead of looking to the user interaction vectors has vectors containing rating values, here we should look at them as two sets, where each set contains the identifier of the items that were rated by that respective user. If the two sets are the same (i.e. both users rated the same items, independently of the values associated with each rating), their Jaccard index will be 1; if they don't have any rating in common, then the similarity value will be 0.

$$JAC(u,u') = \frac{|U \cap U'|}{|U \cup U'|} = \frac{|U \cap U'|}{|U| + |U'| - |U \cap U'|} \tag{2.9}$$

with U being the set of items that were rated by u and U' the set of items that were rated by the user u'.

2.1.2.5 Mean Squared Differences

This is the most intuitive similarity, as it only consists of computing the multiplicative inverse of the mean of the squared difference for each co-rating.

$$MSD(u,u') = \left(\frac{\sum_{i \in I(u,u')} (r_{u,i} - r_{u',i})^2}{|I(u,u')|} + 1 \right)^{-1} \tag{2.10}$$

The multiplicative inverse is added here so that we get the expected behavior of approaching 1 when the two rating vectors are very similar, and approximating 0 whenever they are dissimilar. Summing 1 also guarantees that there's never a division by zero, given a non-empty set of co-rated items.

2.1.3 Feedback Types

We can classify a data set on the way that feedback is presented: using **explicit** or **implicit** feedback. Concretely, feedback is how the preferences are displayed by the data, to the RS.

Some examples of records with explicit feedback are ratings or like/dislike actions. This type of feedback has a powerful advantage: being fairly easy to interpret when a user likes an item or not, so the RS is able to easily understand which items were of use to that particular user. Gathering these behaviors although, is not an easy task, since this is a time-consuming process that usually requires a deviation from the preferred user-journey path. More issues that come from explicit feedback data sets are the various biases that need to be handled:

1. Some users are more lenient or strict than others, and that results in rush deviations. A way to deal with this is to do some data pre-processing by normalizing the provided feedback using normalization methods such as the Gaussian or decoupling normalization Jin and Si [2004].

2. Another consequence of using explicit feedback is that some users only provide ratings whenever they are unhappy with a product, whilst others only provide ratings whenever they enjoyed the product. Statistically, these users would always have either a low or a high rating average, and it would certainly make it hard for any system to learn their behavioral patterns.

Coming back to discuss implicit feedback, this is a more realistic representation of what goes on during a user-system interaction. Most implicit feedback data sets are based on system logs (but can also be provided as a user-item record table), a more descriptive set that can help to understand some important factors that were previously unknown. One major advantage over explicit feedback is that users don't even notice that their own system interactions are being used to improve their personalized experiences, without the need to get distracted by providing ratings on whether they liked a certain product or not. Feedback biases derived from the distinct user personalities have little effect on systems that are based on implicit data. But building reliable RSs using implicit feedback only is not always easy, and it might become a very tricky task to succeed due to the low accuracy of this type of feedback (Jawaheer et al. [2010]), and the high number of existing records having usually many distinct features.

Data Set	# Records	# Users	# Items	Sparsity	Min. Rating	Max. Rating
ML-100k	100000	943	1682	93.6953	1	5
ML-1M	1000209	6040	3706	95.5316	1	5
ML-10M	10000054	69878	10677	98.6597	1	5
ML-20M	20000263	138493	26744	99.46	1	5
Jester-4.1M	4136360	73421	100	93.6953	-9.95	10
Jester-1.8M	1842370	54905	150	95.5316	-10	10
Book-Crossing	1149780	105283	340556	99.9968	0	10
Goodreads	228648342	876145	2360650	99.9889	0	5
Netflix	100480507	480189	17770	98.8224	1	5

Table 2.1: Statistics of the discussed data sets.

2.1.4 Data Sets

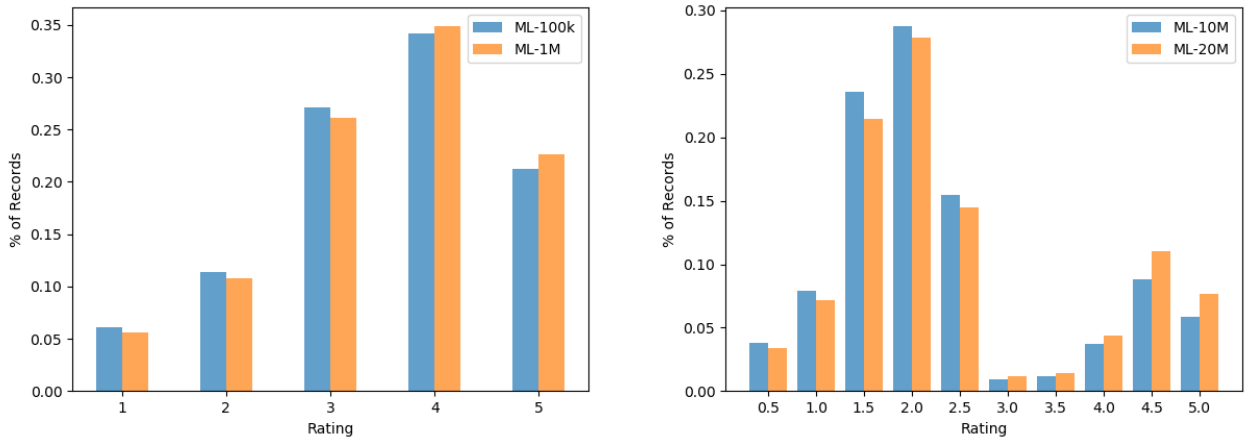
This section lists some of the most popular data sets used for training and evaluating RSs, where most are being used for the purposes of this work. For a more extensive list, please refer to the following resource: http://cseweb.ucsd.edu/~jmcauley/datasets.html.

To complement this section, there is also some statistics about these data sets in Table 2.1, showing the total number of records, unique number of users and items, the interaction matrix sparsity, as well as the minimum and maximum rating values for each discussed data set.

2.1.4.1 MovieLens

The GroupLens research group from the University of Minnesota has been continuously publishing various versions of the MovieLens (a movie recommendation service) data set Harper and Konstan [2016] - the first having 100k records, the second with 1M, the third with 10M, and the fourth one having 20M records. Each record has an associated user, item, rating, and timestamp value. These data sets already have some pre-processing applied, by removing users that have made less than 20 ratings. Some content information is also available, namely movie titles, genres, tag genomes (properties displayed in the movie such as thought-provoking, realistic, etc.) and user-generated metadata about movies. Each rating is an integer from 1 to 5.

Figure 2.4 shows how the rating values are distributed among the total number of records on each data set. Note that the versions of the MovieLens data set with 10M and 20M records have rational rating values, so these are displayed in a distinct plot. All versions have a similar distribution, but with an increasing sparsity value, which means that there is less user-item interactions relative to the size of the interaction matrix.



(a) MovieLens 100k and 1M

(b) MovieLens 10M and 20M

Figure 2.4: Bar chart of the rating values present on the MovieLens data sets.

2.1.4.2 Jester

Jester is a research project from the UC Berkeley Laboratory for Automation Science and Engineering, with two published data sets, each containing 4.1M and 1.8M joke ratings, respectively (collected from a joke recommendation service) Goldberg et al. [2001]. Since each rating is a continuous value from -10 to 10, we represent these distributions with histograms, as shown in Figure 2.5.

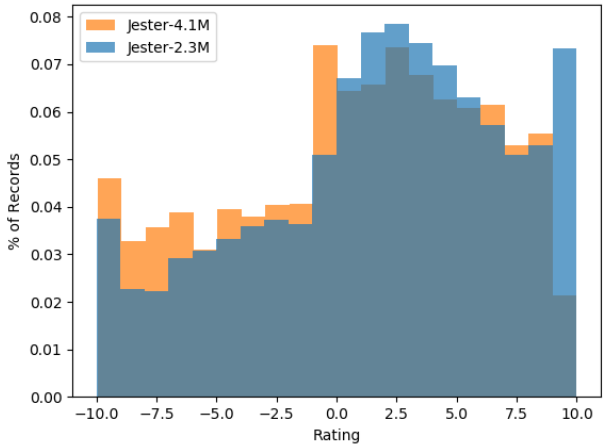


Figure 2.5: Overlapping histograms of the rating values present on the Jester data sets.

2.1.4.3 Book-Crossing

This data set results from a crawl made by a member of the Institut für Informatik of the Universität Freiburg, and contains about 1.1M explicit and implicit ratings Ziegler et al. [2005]. Each record has a user, item, and rating feature, where implicit records are marked by having the rating feature set to 0. It also comes with user info (age and location), as well as book content properties (titles, authors, year of publication, and publisher). Each rating is an integer from 0 to 10. Figure 2.6 shows a bar chart of the rating distribution, where implicit ratings are predominant, representing $\approx 60\%$ of the data set.

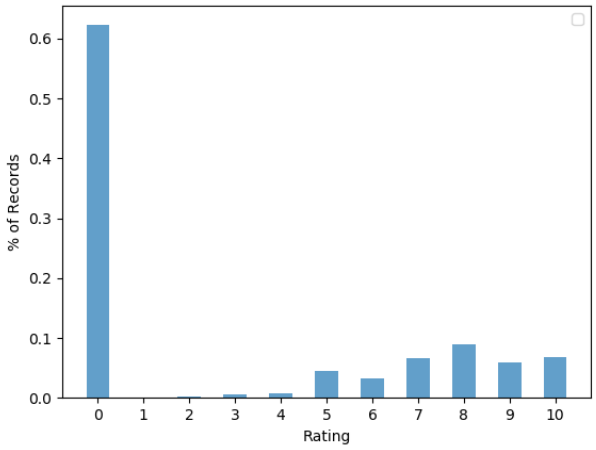


Figure 2.6: Bar chart of the rating values of the Book-Crossing data set.

2.1.4.4 Goodreads

Goodreads is a collection of huge data sets containing user ratings (about 228M), user text reviews, and book content information that was collected by the University of California San Diego Wan and McAuley [2018]. With integer ratings ranging from 0 to 5, and 0 representing implicit ratings, we show its distribution in Figure 2.7. Again, the implicit ratings make up the majority of the data set ($\approx 55\%$). This only enforces the importance of being able to extract knowledge from data with this type of feedback.

2.1.4.5 Netflix

As part of one of the greatest recommendation challenges ever made that highly promoted the advancement of the research on RSs, the Netflix Prize challenge Bennett et al. [2007], was the Netflix data set, which couldn't be left out of this list. It is an explicit feedback data set containing integer ratings from 1 to 5. Figure 2.8 shows a bar chart with the rating values and their respective frequency on the Netflix data set, which follows the distribution seen in the MovieLens data sets very closely.

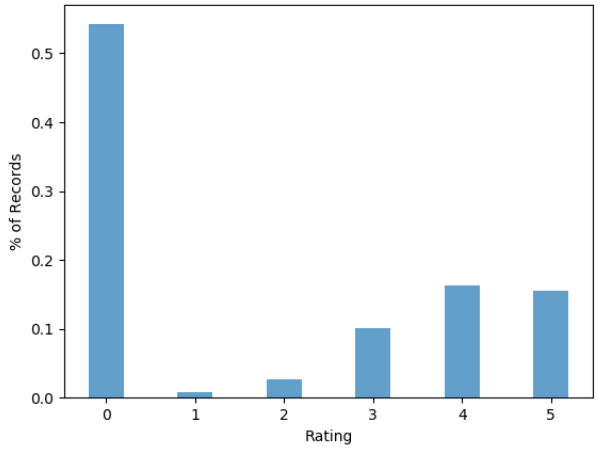


Figure 2.7: Bar chart of the rating values present on the Goodreads data set.

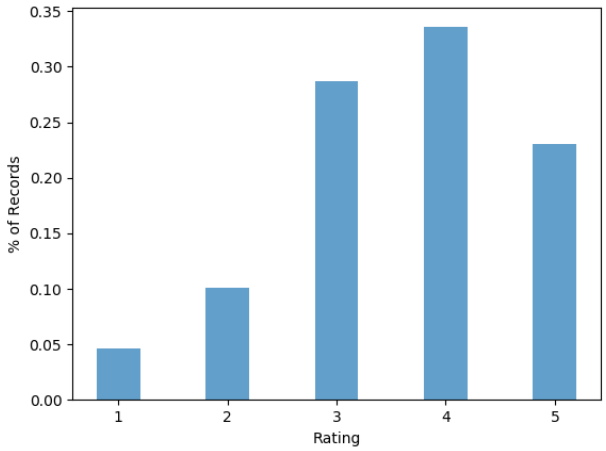


Figure 2.8: Bar chart of the rating values present on the Netflix data set.

2.1.5 Calculating Recommendations

There are two main types of outputs produced by recommender systems, which can be: scores associated with a given user-item pair; or an item-ranking from the provided items for a given user. The former is usually referred to as predictive RSs, and the latter as learning-to-rank (L2R) RSs.
Specifically, a predictive RSs would compute:

$$f(u, i) = s, \tag{2.11}$$

where s should increase as the item i is predicted to be more appropriate for user u. Comparatively, L2R RSs would compute the predicted ranking as:

$$f(u, I) = I', \tag{2.12}$$

where I' would be the predicted best item-ranking from the list of items I, with respect to user u.

Choosing which output to follow for a RS is dependent on the final purpose. Predictive RSs can also be used to rank items, although a prediction for each singular item is required first, together with a final sort by promoting items with higher scores. The L2R RSs aim to be more specific in trying to correlate items in the provided list, to compute an optimal final listing. This distinction is important because each output type should be evaluated with the most appropriate metrics. While the evaluation of predictive RSs would make more sense if the applied metrics measure the difference between two continue values (i.e. prediction scores), applying these to L2R RSs would make little sense - for these, applying IR metrics is the usual case. This matter is further discussed in section 2.1.7.

2.1.6 Sampling Methods

To understand how sampling is usually done in RSs, we start by describing what is a **positive** and **negative data-point/instance**. A positive data-point reefers to a user-interaction that has occurred, where a negative one reefers to a user-interaction that has not yet occurred. For example, (u, i) is a positive data-point when the user u interacted with the item i, and a negative data-point whenever the user u has never interacted with the item i. There is also another variation of this definition, where a negative instance is also any user-interaction with a low feedback/rating value.

When training deep learning-based RSs we must provide not only positive feedback, but also negative feedback, to avoid sampling bias as much as possible. This process is called negative sampling, and should be done so that the recommender can ingest negative feedback during prediction-time as well. The degree of the applied negative sampling is usually denoted as a hyperparameter called **negative ratio**, which can be an integer representing the number of negative instances to include for each positive point, or can also be a fraction representing the proportion of negative instances to be present in the final training data set. Usual values for this hyperparameter are between 2 and 10.

It is also important to understand the distinction between point-wise and list-wise sampling techniques. Point-wise methods sample a single independent positive or negative interaction at a time, while list-wise methods sample multiple related interactions at a time. Each RS requires one of these sampling methods to be applied: when it relies on a single-instance training, like most predictive RSs, a point-wise sampling strategy should be used; otherwise, when a RS requires multiple related interactions, such as data-points grouped by each user and sorted by timestamp (e.g. Caser Tang and Wang [2018]), a list-wise sampling strategy should be applied. An example of both strategies is depicted in Figure 2.9. To my best knowledge, at the moment no recommendation framework makes available both these sampling techniques integrated with the generation of negative instances, and having it would be very useful to simplify the steps of training and testing a RS, as well as promoting reproducibility.

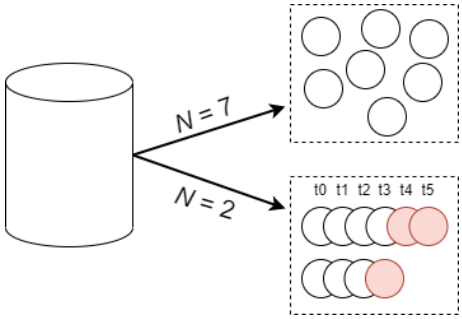


Figure 2.9: Illustrative example of sampling 7 data-points using a point-wise sampler, and 2 lists of data-points sorted by timestamp, using a list-wise sampler.

2.1.7 Evaluation and Metrics

Evaluating whether a RS is useful for a given use-case or not, requires some knowledge of the important properties for succeeding at that use-case, on how to structure the evaluation procedure, as well as the differences between each used metric. There are two main evaluation paradigms: online and offline evaluation. Online evaluation procedures offer more trustworthy results, since the RS is being used and tested against real users performing real tasks. On the other hand, these kinds of tests are not practical for most practitioners or researchers, and that justifies the wide adoption of offline evaluation methods. Offline evaluation techniques are based on evaluating the RS's performance on a set of previously recorded data (usually system logs or historical transactions). Garcin et al. [2014] shows a very clear and practical example of the difference between the results provided by the distinct types of evaluation procedures. From now on, this section will only discuss topics that are related to offline evaluation, due to the reasons stated before.

To train and evaluate a RS, one has to convert a full data set $\mathcal{D}$ into two disjoint subsets $\mathcal{D}_{train}$ and $\mathcal{D}_{test}$, by applying some train/test split method. After this splitting process has been applied, the RS is trained using $\mathcal{D}_{train}$, and evaluated on $\mathcal{D}_{test}$. The test procedure should not affect the model in any way: no weight updates are made and no hyperparameters should be adjusted. To fine-tune the model's hyperparameters, it is useful to do a second split on $\mathcal{D}_{train}$, obtaining $\mathcal{D}_{train'}$ and $\mathcal{D}_{validation}$. With this setting, the former segment should be used to train the RS, whilst the latter should be used to understand how distinct values of the hyperparameters affect the model's performance; after the training and fine-tuning steps are made, the evaluation process on $\mathcal{D}_{test}$ can take place.

To split a data set, one can use, for example, one of the following methods:

- ***Random Split***: As the name suggests, this technique just randomly splits the full data set into a train and test set, where the train set has $p\%$ of the data points of the full data set, while the test set has the remaining $(100-p)\%$. Usually, there are no guarantees that all users present on the test set will also be on the train set, so some RSs might have some problems when being evaluated with splits resulting from this technique (e.g. models that use user embeddings).

- ***Matrix Split***: This split technique samples $p_i\%$ of the total items from $p_u\%$ of the total users into the test set, while the missing $(100 - p_i)\%$ items from all the users make up the training set. Every user is present on the training set, since the $p_u\%$ sampled users are not removed from this set; only some interactions provided to $p_i\%$ items are sampled into the test set.

- ***Leave-k-out Split***: Consists on splitting, for each user present on the full data set, k of this user's interactions (might be the last k, or k randomly sampled interactions). This means that every user present on the test set, will also be present on the train set. Some other parameters might also be taken into account, such as the minimum number of user interactions required to sample the k interactions for that user, or the minimum rating value required for a data point to be included on the test set. Also worth to note that if for each user, only the last k interactions are moved to the test set, this splitting technique allows for correctly testing the behaviours of a sequence-based RS.

- ***K-fold Cross-validation***: This method is a little bit different from the previous ones, as it splits the data set into k independent sets - also called folds. Each fold is used once as the test set, and $k - 1$ times as part of the train set. That makes the train set to always be a combination of $k - 1$ distinct folds. To get the final performance of the RS, one could take the average test score obtained on each of the test folds.

Now that we understand the basics of splitting a data set for an evaluation procedure, lets talk about some metrics that are used to evaluate whether a RS is providing good and valuable predictions or not. As discussed in section 2.1.5, there are two main types of outputs that a RS can provide: a score assigned to an item (predictive) or rank a set of provided items (learn to rank). Intuitively, these strategies cannot be evaluated in the same manner, as one provides a score for an item, while the other provides a list of items (or a list of ranking scores). For that reason, several metrics are applied accordingly to the RS's output. We'll now go through several metrics and discuss them, providing some tips on which cases they should be used.

2.1.7.1 Mean Absolute Error (MAE)

This metric measures the average difference between each pair of continuous values of two vectors, therefore it is useful for evaluating predictive RS's. It's formulated as follows:

$$\text{MAE}(y, \hat{y}) = \frac{1}{n} \sum_{i=1}^{n} |y_i - \hat{y}_i| \tag{2.13}$$

Whenever it is desired the loss to be directly proportional to the obtained error (i.e. if the prediction error of the i^{th} instance is twice as large as the prediction error of the $(i-1)^{th}$ instance, then $L(y_i, \hat{y}_i) = 2 \cdot L(y_{i-1}, \hat{y}_{i-1})$), MAE is usually the suggested metric. It penalizes small and large errors proportionally, and it's impact is easy to understand.

2.1.7.2 Root Mean Squared Error (RMSE)

Another very popular metric that measures the average error of predictive RS's. While MAE uses the absolute value of the error, the RMSE takes the square of the error, making it penalize errors in a different way.

$$\text{RMSE}(y, \hat{y}) = \sqrt{\frac{1}{n}\sum_{i=1}^{n}(y_i - \hat{y}_i)^2} \tag{2.14}$$

With RMSE, large errors result in a much bigger negative contribution, when compared to smaller errors. Therefore, if the objective is to minimize large errors, RMSE is a good choice.

2.1.7.3 Precision

This is the first listed metric that targets evaluation of ranking-based RSs. It calculates the proportion of recommended items that are relevant. Assuming that y is the set of relevant items and $\hat{y}$ the set of recommended/predicted items, the precision metric is formulated as follows:

$$\text{P}(y, \hat{y}) = \frac{|y \cap \hat{y}|}{|\hat{y}|} \tag{2.15}$$

In order to evaluate top-k rankings, it is also useful to define precision at k, denoted by P@k:

$$\text{P@k}(y, \hat{y}) = \frac{|y \cap \hat{y}'|}{|\hat{y}'|}, \tag{2.16}$$

where $\hat{y}' = \{\hat{y}_i \mid i \in [1, k]\}$.
It is easy to see that constructing a contingency table (or confusion matrix) provides us with a good visualization of what are the true/false positives and the true/false negatives (see table 2.2).

	Relevant	**Irrelevant**	**Total**
Recommended	$\lvert y \cap \hat{y}\rvert$	$\lvert \overline{y} \cap \hat{y}\rvert$	$\lvert \hat{y}\rvert$
Not Recommended	$\lvert y \cap \overline{\hat{y}}\rvert$	$\lvert \overline{y} \cap \overline{\hat{y}}\rvert$	$\lvert \overline{\hat{y}}\rvert$
Total	$\lvert y\rvert$	$\lvert \overline{y}\rvert$	N

Table 2.2: Contingency table for the recommended vs. relevant items.

2.1.7.4 Recall

Recall is another metric that is useful for evaluating produced ranking sets. It computes the proportion of relevant items that were recommended:

$$\text{R}(y, \hat{y}) = \frac{|y \cap \hat{y}|}{|y|} \tag{2.17}$$

For evaluating top-k rankings, we define recall at k by:

$$\text{R@k}(y, \hat{y}) = \frac{|y \cap \hat{y}'|}{|y|}, \tag{2.18}$$

where $\hat{y}' = \{\hat{y}_i \mid i \in [1, k]\}$.

2.1.7.5 F-score

F-score, also called F-measure or F_β-score, measures the effectiveness of the recommendation by giving β times as much importance to recall, than precision. It combines the previously listed precision and recall metrics in the following manner:

$$\text{F}_\beta(y, \hat{y}) = (1 + \beta^2) \cdot \frac{\text{P}(y, \hat{y}) \cdot \text{R}(y, \hat{y})}{(\beta^2 \cdot \text{P}(y, \hat{y})) + \text{R}(y, \hat{y})} \tag{2.19}$$

Usually the most used β value is 1, which simplifies F_1 to be:

$$\text{F}_1(y, \hat{y}) = 2 \cdot \frac{\text{P}(y, \hat{y}) \cdot \text{R}(y, \hat{y})}{\text{P}(y, \hat{y}) + \text{R}(y, \hat{y})}, \tag{2.20}$$

which sets the importance of the recall to be the same as the importance given to the precision.
Again, for evaluation of top-k rankings, it is defined as:

$$\text{F@k}_\beta(y, \hat{y}) = (1 + \beta^2) \cdot \frac{\text{P@k}(y, \hat{y}) \cdot \text{R@k}(y, \hat{y})}{(\beta^2 \cdot \text{P@k}(y, \hat{y})) + \text{R@k}(y, \hat{y})} \tag{2.21}$$

2.1.7.6 Reciprocal Rank (RR)

The Reciprocal Rank is a ranking metric that takes into account only the first relevant item on a recommendation list, and it increases as we push that relevant item to the top of the list (closer to rank 1). The calculation process is very straightforward, by taking the multiplicative inverse of the rank of the first relevant recommended item:

$$\text{RR}(y, \hat{y}) = \begin{cases} \frac{1}{\text{rank}_I} & \text{if } I \text{ exists} \\ 0 & \text{otherwise} \end{cases}, \tag{2.22}$$

with I being the first relevant item present on the recommendation list $\hat{y}$. If no relevant item is present in $\hat{y}$, then I simply does not exist. Similarly to previous ranking metrics, RR@k is also a valid metric, and it is evaluated by truncating $\hat{y}$ and finding I.

2.1.7.7 Average Precision (AP)

Whenever we have relevant items sharing the same relevancy value (relevant items are of the same importance to the user), this metric is very valuable for evaluating a given recommendation list. Specifically, it increases its value as we push relevant items to the top of the recommendation list:

$$\text{AP}(y, \hat{y}) = \frac{1}{|y|} \sum_{i=1}^{n} \text{P@i}(y, \hat{y}) \cdot \mathbf{1}_y(\hat{y}_i), \tag{2.23}$$

with $\mathbf{1}_S(x)$ being the indicator function, defined as follows:

$$\mathbf{1}_S(x) = \begin{cases} 1 & \text{if } x \in S \\ 0 & \text{otherwise} \end{cases} \tag{2.24}$$

One other thing to note, and that is not taken into account in multiple implementations of this metric, is that repeated items should not be counted twice, that is, their relevancy should only be counted once (the first time they appear on the list).
As for evaluating top-k rankings, we can define AP@k$(y, \hat{y})$ as:

$$\text{AP@k}(y, \hat{y}) = \frac{1}{\min\{k, |y|\}} \sum_{i=1}^{k} \text{P@i}(y, \hat{y}) \cdot \mathbf{1}_y(\hat{y}_i), \tag{2.25}$$

In some implementations, there are different chosen normalization factors, but it was decided that the usage of $\min\{k, |y|\}$ was optimal, as this factor ensures that AP@k has a $[0, 1]$ range and "retains the property that promoting a relevant document always increases the score" Craswell and Robertson [2009].

2.1.7.8 Normalized Discounted Cumulative Gain (NDCG)

This ranking metric is slightly different from the AP, as it takes into account the distinct relevancy scores among relevant items (both items I_i and I_j might be relevant, although I_i can be more relevant than I_j). Again, the value of this metric increases as we push relevant items to the top of the recommendation list.

Obviously there is a dependency on the Discounted Cumulative Gain (DCG) value, and the normalization factor is the DCG value on the ideal recommendation list (denoted as IDCG). Let's start by defining DCG: there are 2 popular formulations, the first one being:

$$\text{DCG}(y, \hat{y}) = \sum_{i=1}^{n} \frac{r_i}{log_2(i+1)}, \tag{2.26}$$

and the second, which puts more emphasis on retrieving relevant items:

$$\text{DCG}(y, \hat{y}) = \sum_{i=1}^{n} \frac{2^{r_i} - 1}{log_2(i+1)}, \tag{2.27}$$

with r_i being the relevancy of $\hat{y}_i$ on y. This relevancy value can be, for example, a user rating. Note that for relevancy values $\in \{0, 1\}$, both versions handle the same result.

The need for a normalization factor comes when we need to compare recommendations consisting on different lenghts. For that reason, we define NDCG as:

$$\text{NDCG}(y, \hat{y}) = \frac{\text{DCG}(y, \hat{y})}{\text{IDCG}(y, \hat{y})}, \tag{2.28}$$

where $\text{IDCG}(y, \hat{y})$ is computed by the DCG on the sorted $\hat{y}$ according to each item's relevancy (items with higher relevancy should come first). This normalized version is defined on the $[0, 1]$ range, and with

that we can safely compare NDCG values for different recommendation lists.
Once again, we also define the formulations of this metric for top-k recommendation, the first version being:

$$\text{DCG@k}(y, \hat{y}) = \sum_{i=1}^{k} \frac{r_i}{log_2(i+1)}, \tag{2.29}$$

and the second:

$$\text{DCG@k}(y, \hat{y}) = \sum_{i=1}^{k} \frac{2^{r_i} - 1}{log_2(i+1)}, \tag{2.30}$$

with the normalized version defined as:

$$\text{NDCG@k}(y, \hat{y}) = \frac{\text{DCG@k}(y, \hat{y})}{\text{IDCG@k}(y, \hat{y})} \tag{2.31}$$

2.2 Machine Learning

Machine Learning (ML) is a scientific field that addresses the challenge of how to build systems that improve and evolve its behavior through experience and data ingestion. More formally, "A computer program is said to learn from experience E with respect to some class of tasks T and performance measure P, if its performance at tasks in T, as measured by P, improves with experience E." - Mitchell [1997].

There have been great advances in ML during the recent decades, and its applicability across various fields has turned it into a very attractive field. With the advances it brought on computer vision, natural language processing, speech recognition, autonomous agents, and many other applications, it is currently a very respected field, with a broad community contributing to push its boundaries even further. With the huge amounts of available data on multiple areas, and with the computational power being so strong and cheap, the use of ML techniques is very common nowadays.

The goal of these systems is to build a mathematical model that detects patterns in the provided data $\mathcal{D}_{train}$, allowing it to adjust its behavior whenever the system gives a wrong prediction. Assuming that we want to learn a specific task that can be reduced to an unknown target function $f : \mathcal{X} \to \mathcal{Y}$, then we want to find a function $g : \mathcal{X} \to \mathcal{Y}$ such that $g \approx f$ (g approximates f).

Our predictive function g is drawn from our hypothesis space $\mathcal{H}$, which the model iteratively explores using the training data set $\mathcal{D}_{train}$. There is no guarantee that $f \in \mathcal{H}$, but since we control our hypothesis space (by fine-tuning the model's hyperparameters), we can achieve very good results on many distinct problems. This hypothesis space can also be increased, by increasing the model's capacity. One thing to note is that a large $\mathcal{H}$ is not always better than a smaller one, since it might contain functions that overfit our data, and it is harder to explore due to its dimension. Most of the time it is easier and faster to try various smaller hypothesis spaces (by using domain knowledge and experience gathered from training other models), rather than exploring a very large $\mathcal{H}$ right away.

Another very important challenge that was mentioned above and that requires further formalization, is the overfit phenomena. While training a model to approximate the desired target function, one has

to be careful to make sure that the model is making good progress by learning the appropriate patterns. Whenever the model displays accurate behaviours on $\mathcal{D}_{train}$ but is not able to generalize for points outside of this set (performs poorly on $\mathcal{D}_{test}$), then the model has probably overfited the training data. Let us define E_{train} as the training error, and E_{test} as the test error. Optimal models will try to minimize these two probabilities:

$$P[E_{train}(g) > \epsilon] \tag{2.32}$$

$$P[|E_{train}(g) - E_{test}(g)| > \epsilon] \tag{2.33}$$

Minimizing equation 2.32 means that g is acquiring knowledge from $\mathcal{D}_{train}$, whilst minimizing equation 2.33 means that the model is able to generalize its predictions on $\mathcal{D}_{test}$.

Here, ϵ represents the error margin. Although that achieving a margin of 0 would be theoretically the optimal approach, in real-world applications it is never possible to achieve such a small value. There are multiple reasons for this to not be possible, but here are some quick justifications for this statement:

- Stochastic and deterministic noise present on the data set (noise introduced by unpredictable agents or by sporadic system failures);
- Not enough training data (complex tasks require more data);
- Provided data is not representative enough (for the model to be able to learn a pattern from the data, then there must be a pattern that can be learned that helps to solve the target problem);

Similarly to overfitting, underfitting is also an undesired effect that appears when the model hasn't been able to learn enough from the training data, therefore presenting very poor results. On overfitting cases, E_{train} is small while E_{test} is high, but when a model is underfitting, both E_{train} and E_{test} are high.

There are 3 major learning paradigms, and they differ on multiple aspects, from the data requirements to the training procedures, as well as possible practical applications. Lets briefly explore the characteristics of each learning type.

2.2.1 Supervised Learning

Supervised Learning is the most common type of learning, and it is currently applied to various learning systems. The data set $\mathcal{D}$ consists of pairs of observations (x_i, y_i), where x_i is the i^{th} data point and y_i corresponds to the desired output for that observation. It is called supervised due to the way the data provided to the model is structured: a desired output y is always required, so it is supervised because these desired outputs guide the model through training (as if there's a "supervisor" labeling each data point for the model).

Recommender systems are a type of practical application where this type of learning is commonly applied. When training a RS, we can provide data containing interaction pairs, where users and items

that have interacted are the input $x = (u, i)$, and the desired output is the rating that the user gave to that item $y = r_{u,i}$ (i.e. the level of satisfaction the user has with the item).

2.2.2 Unsupervised Learning

Unsupervised Learning is characterized by not having the desired outputs as part of the data set $\mathcal{D}$. Each data point consists only of the input vector x_i, and nothing more. These types of learning systems are usually applied to clustering problems (i.e. find groups that share common characteristics).

For Recommender Systems, this technique can be applied to find similar users/items, depending on the contents of each data point, as well as to learn representations for users/items.

2.2.3 Reinforcement Learning

Reinforcement Learning is the most recent learning paradigm that has been gaining more visibility and popularity due to the increasing number of experiments conducted by academia members on various distinct problems. This technique is characterized by making the learning agent explore the existing environment, transitioning between states by choosing and performing an action. The chosen action is the one that the agent ”believes” will maximize the final reward.

In the context of Recommender Systems, this approach hasn't been applied often with much success, but some interesting approaches like Zhao et al. [2018] show interesting results, with a modified Deep Q-Network (DQN) that captures both positive and negative feedback.

2.3 Deep Learning

Deep learning is a very popular and active field nowadays, whose most essential concepts come from the 1980s, and it consists of the study of how to build deep neural networks for solving complex problems. The advancements in terms of the amount of available data and the increasing availability of cheap computational power led this field to be one of the most publicly discussed. This was also due to the recent incredible achievements in various fields such as: natural language processing (NLP) techniques Radford et al. [2019], image recognition He et al. [2016], autonomous driving advancements Chen et al. [2015] and other autonomous agents Silver et al. [2017].

To better understand what deep learning is, let us start by discussing how an artificial neural network (ANN) is conceptualized: this type of network learns a function that tries to map its inputs to the desired output, using a series of linear and/or non-linear transformations of the input. An ANN is a collection of connected units - called artificial neurons - and depending on the network type, they can be organized in various layers, having at least an input layer and an output layer (with the possibility of having multiple layers in between - these are called hidden layers), or not. It is said that when an ANN has more than one hidden layer, it is called a deep neural network (DNN). Whenever a signal transverses from one layer to another, an activation function is applied, therefore transforming the provided signal. Depending on the

type of the neural network in question, the signal might pass through the same layer(s) multiple times. If the network is being trained in a supervised learning fashion, then at the last layer, the model output is evaluated with the help of a loss function, and the network's weights are updated accordingly (usually by applying gradient-based algorithms such as backpropagation), so that it can perform better on the next trials. The conjunction of the units, their connections, the chosen activation functions, and the loss function, compose the architecture of a neural network, as shown in Figure 2.10.

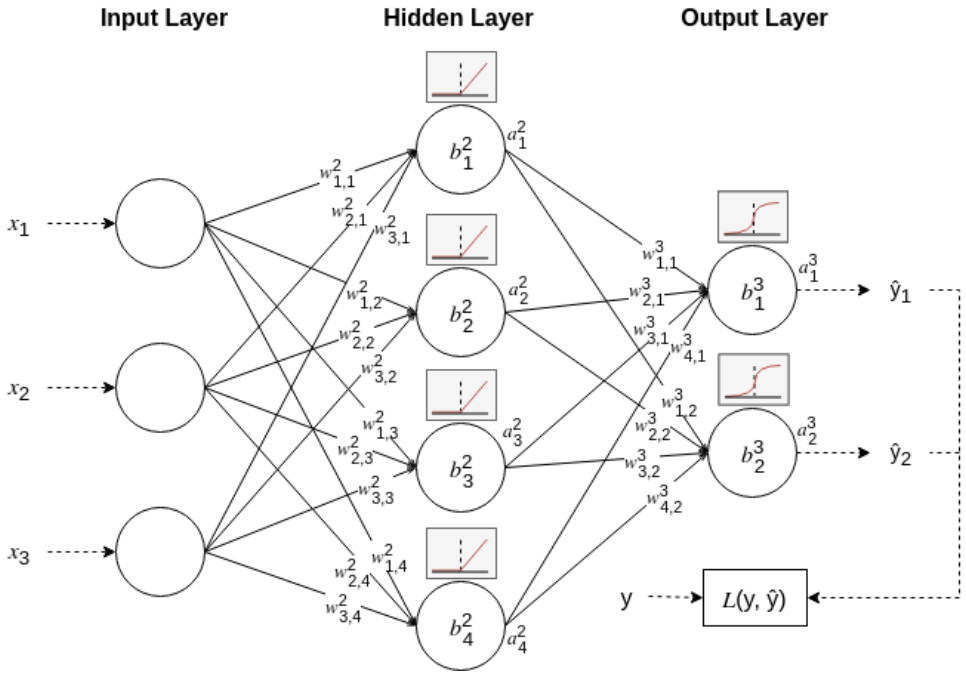


Figure 2.10: Illustration of a 2-layer Fully Connected Feedforward Neural Network with Supervised Learning: the input layer has 3 units, the hidden layer has 4 units, each using the ReLU activation function and the output layer has 2 units, each using the sigmoid as the activation function. Each i^{th} unit in a non-input layer l has a bias term b_i, a weight vector $w^l_{j,i}$ and an activation function value a^l_i. After the transformed input reaches the output layer, it is called the prediction vector $\hat{\mathbf{y}}$. The loss function is then evaluated with the predicted values and the expected values $\mathbf{y}$. Note that when we count the number of layers of a NN, we don't take the input layer into account, since this layer does not have tunable weights.

It has been proven a while ago (1989), that a feedforward[1] ANN with an unlimited number of artificial neurons in a single hidden layer, using the sigmoid activation function, can approximate any continuous function within a compact set (i.e. for inputs in a finite range) Cybenko [1989]:

$$|f(x) - g(x)| < \epsilon, \forall x \in [a, b] \ , \tag{2.34}$$

where f is the target function, g the learnt function, ϵ the error margin (where $\epsilon > 0$) and both a and b are

[1] Described further on this section.

finite.

This **universal approximation theorem** presented on Cybenko's work was a very important step towards the wide adoption of these tools. Later on, more work was developed on the idea of expanding this theorem, by analyzing and publishing proofs for more distinct and broad settings, such as Lu et al. [2017] and Hanin and Sellke [2017], which generalize this theorem for width-bounded deep neural networks using the rectified linear unit (ReLU) as the activation function.

Since there are several classes of neural networks, it is worthwhile to discuss some of the most popular ones briefly, so that the basic concepts are in mind when trying to understand how certain deep learning-based recommender models work and behave. Please note that within each class there are multiple variations, and not all will be discussed. After that, some notes and examples of activation and loss functions will be provided.

2.3.1 Classes of Neural Networks

2.3.1.1 Feedforward Neural Networks

This class of NNs is very popular for its simplistic connection-architecture and it is usually the first type of network that one tries to get familiar with. A feedforward NN is defined by having each unit from the l^{th} layer connected to at least one or all units of the $l + 1^{th}$ layer. This means that for each unit of a certain layer, there are as many corresponding weights as the number of units connected to it from the next layer.
A fully connected feedforward NN is the most popular variant, that consists on having dense layers only (every unit from each layer is connected to every other unit from the next layer), and these are usually called multilayer perceptrons (MLPs) Pal and Mitra [1992] - Figure 2.10 shows an example of a 2-layer MLP.

The fixed-sized input represents the input layer, and when passed through the first hidden layer, the produced signal always travels from the previous to the next layers of the network, without ever going back, because these architectures are acyclic. This leads us to think that each layer applies some type of processing that allows the NN to decode abstractions within the input data: more complex input intuitively requires a more complex and deep architecture. After training, these networks always provide the same prediction for the same input vectors.

2.3.1.2 Recurrent Neural Networks

Recurrent Neural Networks (RNNs) are a class of NNs that support inputs of unlimited size. Since these networks support cycles, they can store state from previous inputs even after training, making them better suited for sequential inputs, such as time series or user actions over a time period on the context of recommendation.

The signal passing through the network is not only influenced by the network's weights, but also by

hidden states that are supposed to represent context from previous inputs. With that, the produced outputs might differ even for the same input. Another key difference from feedforward NNs is that distinct parts of the input can share some weight values. This parameter sharing characteristic is what allows these NNs to be able to support inputs independently of their size.

A widely popular variant called Long Short Term Memory (LSTM) networks allows long-dependency learning. Simple RNNs are able to memorize context from small to medium-sized sequences only, but LSTMs tackle this problem by controlling which information is useful to be stored or discarded.

2.3.1.3 Convolutional Neural Networks

Convolutional Neural Networks (CNNs) are similar to feedforward NNs, but instead of using matrix multiplication, they use convolution (a mathematical operation), in at least one of the network's layers. The purpose of convolution is to do feature extraction from the input, by preserving its spatial characteristics.

Since CNNs are spatial invariant, they are useful to learn features present on data that has spatial dependencies, such as images. Feedforward NNs can also be used for these types of tasks, but they have a hard time trying to succeed, since learning spatial dependencies is harder when the input shape is not preserved (feedforward NNs require N-dimensional input vectors, while CNNs can receive $W \times H$ input matrices). Sometimes complex models use a combination of these two classes of NNs, such as LeNet LeCun et al. [1998], which uses a CNN to extract features from images, and then passes these features to a MLP to complete the classification process.

2.3.1.4 Autoencoders

An autoencoder (AE) is a class of NNs that is applied with unsupervised learning techniques and it is useful to learn representations of the provided input (representation learning). The goal of autoencoders is to be able to output their exact input, with hidden layers of reduced dimensions (usually referred to as the network's bottleneck). This forces the network to learn a compressed data representation of the original input, therefore it can be regarded as a dimensionality reduction technique.

One variation that will be discussed further on this document, is the denoising autoencoder (DAE), which instead of being trained with the complete input, it tries to reconstruct a corrupted (or distorted) input. This is useful because it might help training procedures that use noisy data, making the model noise-resistant.

2.3.2 Activation Functions

An activation function σ is one of the fundamental pieces of any NN architecture. They serve to compute the output of a single node (node's activation), by receiving the signal of each neuron that is connected to it.

The complexity of these functions highly influences the complexity of the overall network. Simple functions such as the identity function 2.35, only allows a node to go so far.

$$\sigma(x) = x \tag{2.35}$$

More complex ones such as the sigmoid function 2.36, allows the node to exhibit nonlinear behavior while also limiting the node's output to the $[0, 1]$ range, which is good to compute probabilities.

$$\sigma(x) = \frac{1}{1 + e^{-x}} \tag{2.36}$$

Activation functions should be differentiable so that the model can be trained via gradient-based optimization methods. Another good property to have is monotonicity, so that the function's error surface is convex, which helps the model to not be stuck in local minimums during training. There are various other widely used functions that exhibit distinct behaviors and characteristics, as described in Karlik and Olgac [2011].

2.3.3 Loss Functions

A loss function L is used to compute how well is the network doing on a given point. In other words, it is the method to compute the prediction error (or loss value) of the network, which is also used to calculate the gradients to update the network's weights so that it can perform better on the next predictions. It receives the network's prediction vector $\hat{y}$, as well as the expected vector y, and the goal is to minimize the loss as much as possible (note that for NNs with a single output unit, these values can be represented as a single value, or by a vector with 1 value). Sometimes we may wish to train a NN to maximize some function, and in that case, the loss function is actually referred to as the objective function.

In order for a network to be trained in a gradient-based optimization process, the loss function must be differentiable as a function of the network parameters. Whenever we want to apply a non-differentiable loss function to train a model, we have to find an approximate function that is differentiable, otherwise the network's weights cannot be updated and its behavior will never be adjusted.

One loss function that is widely used on regression tasks, is the mean squared error loss 2.37.

$$L(y, \hat{y}) = \frac{1}{n} \sum_{i}^{n} (y_i - \hat{y}_i)^2 \tag{2.37}$$

For classification tasks, some valid options could be the binary cross-entropy[2] (for binary classification tasks) 2.38, and the categorical cross-entropy (for multi-class classification problems) 2.39, which is just a generalization of the first one.

$$L(y, \hat{y}) = -(y \log(\hat{y}) + (1 - y) \log(1 - \hat{y})) \tag{2.38}$$

[2]The binary cross-entropy loss can also be referred to as log loss or logarithmic loss function.

$$L(y, \hat{y}) = -\sum_{i}^{n}(y_i \log(\hat{y}_i)) \tag{2.39}$$

Some loss functions, such as the cross-entropy loss, depend on the output unit(s) to be defined in a specific range, so we must choose this function carefully and in accordance with our network's architecture. In this case, the output should be in the $[0, 1]$ range.

If the loss function that is chosen to train a model is not suited to the task at hand, then we can end up with desirable loss values, while the model shows very poor predictive performance in a different environment. This situation can also happen in other cases, for example, when the model is overfitted to the training data.

2.3.4 Improving Model Training

Some techniques for improving the training process and the performance of NNs are presented and briefly discussed in this section.

2.3.4.1 Early Stopping

Early stopping is a regularization technique to avoid overfitting, and can also be used to speed up the training procedure. When no other form of regularization is used, it is expected that the model's loss values decrease with more training, while its predictive performance starts to decrease.

This described behavior is obviously not desirable, since it gives a false perception that the model is improving when it is actually getting worse, i.e. has a high generalization error. One way to avoid it is by employing this technique, which simply consists of verifying if the model's performance outside of the training set continues to increase, or not.

On every model training pipeline, we should start by splitting a training set $\mathcal{D}_{train}$ and a test set $\mathcal{D}_{test}$ from the whole available data $\mathcal{D}$. To apply early stopping, we will have to decrease the amount of training data that the model will be able to see, by dividing the original training set $\mathcal{D}_{train}$ into a validation set $\mathcal{D}_{val}$ and a second training set $\mathcal{D}'_{train}$. Then, the model should be trained using $\mathcal{D}'_{train}$, and for every X epochs, we compute the validation error of the model at its current state. As soon as the validation error starts to increase, we stop the training process and set the model's weights to the ones it had on the previous evaluation step.

This sounds very simple, but this exact process might not be the ideal for most situations, since the validation error curve rarely has a convex shape, and we might end up stopping the training too soon. This problem is well described and tackled in Prechelt [1998], and there's no exact way to fix it, since it will always involve a tradeoff between training speed and model generalization.

2.3.4.2 Mini-batch Training

Optimization algorithms that require the entire training set in a single batch, are called batch algorithms. In the context of NNs and DL, its usual to instead use algorithms that take a small number of training points per epoch, and these are called mini-batch methods.

In mini-batch stochastic gradient descent, it has been shown that generalization improves with smaller batches per training Wilson and Martinez [2003], with values between 1 and 32 consistently providing the best results (assuming a small learning rate is also used). This improvement happens because each gradient update has more noise with a small batch size than with a big one, which helps non-convex optimization processes to not be stuck in a local minima. Note that it is important that mini-batches are sampled randomly, so that these samples are independent, providing an unbiased estimate of the expected gradient.

This improvement comes with a somewhat intuitive cost: the runtime of the training process will increase with smaller batches, because more steps will be required to transverse the whole training set and also due to the reduced learning rate.

2.3.4.3 Pre-training

Pre-training is a technique to speed up NN training, by leveraging from prior training processes. Although not always applicable, due to reasons explained further, when it can be applied it shows a tremendous improvement in training runtime Hinton and Salakhutdinov [2006].

Whenever two tasks have some concept in common (e.g. both image recognition and object detection involve image processing), there is a possibility that pre-training can be applied. The way this is done is by extracting the trained network's weights and using them at the start of the training process of the new network. This obviously requires the new network's architecture to not be drastically different from the already trained one.

Some other forms of pre-training involve splitting an untrained network into sub-networks, and train them individually so that later we can put them together before starting the new training process. This has shown good results by allowing the training process to be faster, since the initial weights of the main network are not computed at random, but are instead inherited from these sub-networks.

Several practical experiments have shown this technique to be extremely useful, as described in McAuley et al. [2015] for image processing applications, and in He et al. [2017] in the context of recommender systems.

Chapter 3

Related Work

This chapter briefly describes some popular frameworks that facilitate the development of RSs, some of the issues identified by experimenting with them, as well as the out-of-the-box DL-based RSs that are offered by the proposed framework.

3.1 Existing Frameworks

With the growth of the popularity of these systems, there was a need to facilitate their development, and multiple frameworks for this purpose have been built. In this section, some popular ones will be presented, together with a brief discussion about their positive and negative points. A more complete comparison between the listed frameworks is shown in section 5.

3.1.1 DeepRec

This recently published Python framework Shuai Zhang [2019] implements various DL based RSs ranging from various types, and also some utilities to evaluate them. It is also a noticeable example of how to implement various models, whilst maintaining most of the notation present on each respective paper. The huge amount of work required to implement all these systems should be recognized, and it is a great help for those who are trying to understand how to practically implement some of the most complex models in this field.
With that, there are still some issues that are worth to be noticed, specifically:

- ***Splitting Functions***: Currently no data set splitting functions are offered by this framework, so these tasks must be done manually. This is one of the most important processes during the evaluation procedure, and having support for it is crucial. Implementing these functions manually can become really tiresome, especially when talking about cross-validation methodologies, which are also not supported by DeepRec.

- ***Code Structure***: The structure for implementing a RS with this framework could be optimized, for instance, by using abstract classes that allow the developers to know which methods are required to be implemented. At the moment, one has to analyze existing implementations to understand which methods to implement. The code-base is poorly structured, with little separation of concerns, having only two modules: models and utils.
- ***Deterministic Training and Evaluation***: At the moment, DeepRec does not support deterministic training nor evaluation of most implemented methods. This happens because each model that requires sampling data points for the training or testing purposes, has its own implementation of that procedure. This also means that new models that require this feature, need to implement it too.
- ***Data Set Support***: There's no custom builtin data set module, but the framework solves some of the issues by providing functions under the utils module, allowing developers to load data sets using third-party packages. Some other issues are not addressed at all, such as how to easily manipulate data sets (e.g. getting all the interactions for a specific user).

3.1.2 Surprise

Surprise Hug [2017] is the most popular recommendation Python framework listed here (with more than 3.6k stars on GitHub), developed for building and analyzing recommender systems that deal with explicit rating data. It provides some implementations of non-deep learning collaborative-filtering based RSs, such as matrix factorization and neighborhood-based methods. The provided documentation is simple and accurate, with a good code structure and support for deterministic evaluation. Building custom algorithms is very straightforward and easy to achieve task.
Despite the amazing success of this framework, there are some negative points that should be pointed out:

- ***Deep Learning Algorithms***: Surprise was built for non-DL based RSs, which means that there's no support for implementing these types of systems.
- ***Implicit Data***: As stated in the description of this framework, it does not support implicit data. With more research being done towards exploiting this type of feedback, it is a bit of a letdown that Surprise does not support it.
- ***Content Based***: This framework does not offer support for content-based RSs, since it's targeted for collaborative filtering approaches only.
- ***Multi-Columns Data Set Support***: The data set module built for Surprise does not support multiple column records, which means that RSs that take advantage of these extra columns are not supported.
- ***Out-of-Memory Data Set Support***: Data sets that are too big to fit into memory are also not supported by Surprise.

3.1.3 CF4J

CF4J Ortega et al. [2018] is a recommendation framework for the Java ecosystem, and it implements various collaborative filtering memory and model-based algorithms. All the documentation is provided via javadocs, and a framework class hierarchy is shown on the project's GitHub page.

- ***Deep Learning Algorithms***: CF4J was also built for non-DL based RSs, which means that there's no support for implementing these types of systems.
- ***Content Based***: This framework does not offer support for content-based RSs, since it's targeted for collaborative filtering approaches only.
- ***Multi-Columns Data Set Support***: CF4J's builtin data set module does not support multiple column records, which means that RSs that take advantage of these extra columns are not supported.
- ***Out-of-Memory Data Set Support***: Data sets that are too big to fit into memory are also not supported by CF4J.

3.1.4 Spotlight

Spotlight Kula [2017] is another very popular Python framework, that uses PyTorch to build various types of recommender systems, providing various representation techniques, loss functions and data set generation utilities.

- ***Content Based***: This framework does not offer support for content-based RSs, since it's targeted for collaborative filtering approaches only.
- ***Multi-Columns Data Set Support***: The builtin data set module does not support multiple column records, which means that RSs that take advantage of these extra columns are not supported.
- ***Out-of-Memory Data Set Support***: Data sets that are too big to fit into memory are also not supported by Spotlight.

3.1.5 TensorRec

TensorRec is a Python recommendation framework that also uses TensorFlow to train and make model predictions. It is very extensible on the types of embedding, loss, and prediction functions it uses, but it lacks some flexibility and other utilities for exploring more types of RSs.

- ***Splitting Functions***: Currently no data set splitting functions are offered by this framework, so they must be done manually.
- ***Data Set Support***: There's no custom builtin data set module.

- ***Content Based***: This framework does not offer support for content-based RSs, since it's targeted for collaborative filtering approaches only.
- ***Multi-Columns Data Set Support***: The data set representation used internally does not support multiple column records, which means that RSs that take advantage of these extra columns are not supported.
- ***Out-of-Memory Data Set Support***: Data sets that are too big to fit into memory are also not supported by TensorRec.

3.2 Deep Learning-based Recommenders

To better understand how DL-based recommenders work and how they usually make use of the available data, it is important to discuss a few practical examples of these types of systems. Each model discussed in this section is also implemented on the DRecPy framework, which will be discussed in chapter 4.

Each model will include a brief description of what class of NNs it uses, its layer structure, what kind of activation functions it supports between layers, and the loss function.

3.2.1 Deep Matrix Factorization

The Deep Matrix Factorization (DMF) model is introduced by Xue et al. [2017] and it is a very simple model that uses two MLPs: one for learning user's interests and the other to specialize on item's relationships, therefore the model itself belongs to the Feedforward NNs class. Its intended use is for item ranking tasks, despite using a point-wise loss function.

Assuming that $Y \in \mathbb{R}^{|U| \times |I|}$ represents the interaction matrix, where $|U|$ is the total number of users and $|I|$ the total number of items, then a prediction for the user u and item i will require the user u interaction vector ($Y_{u,*}$) and the item i interaction vector ($Y_{*,i}$). Each user interaction vector should contain the interaction value (or rating value) that the user provided to each item, and when no interaction has occurred yet between a user-item pair, then that interaction value should be set to 0. The same logic applies to computing item interaction vectors.

It is also possible to set the interaction vector' values to 1 when an interaction is found, and to 0 when none is found, but this alternative is quickly discarded in its original publication, since this option does not preserve the preference degree of a user on an item.

3.2.1.1 Layer Structure

As stated before, this network actually involves the use of two separate networks, let us denote the user MLP as MLP_U, and the item MLP as MLP_I. The motivation for this split is to specialize each network to represent each entity in a latent low-dimensional space, maintaining its relationships (user interactions and item interactions). The model's neural architecture is depicted in Fig. 3.1.

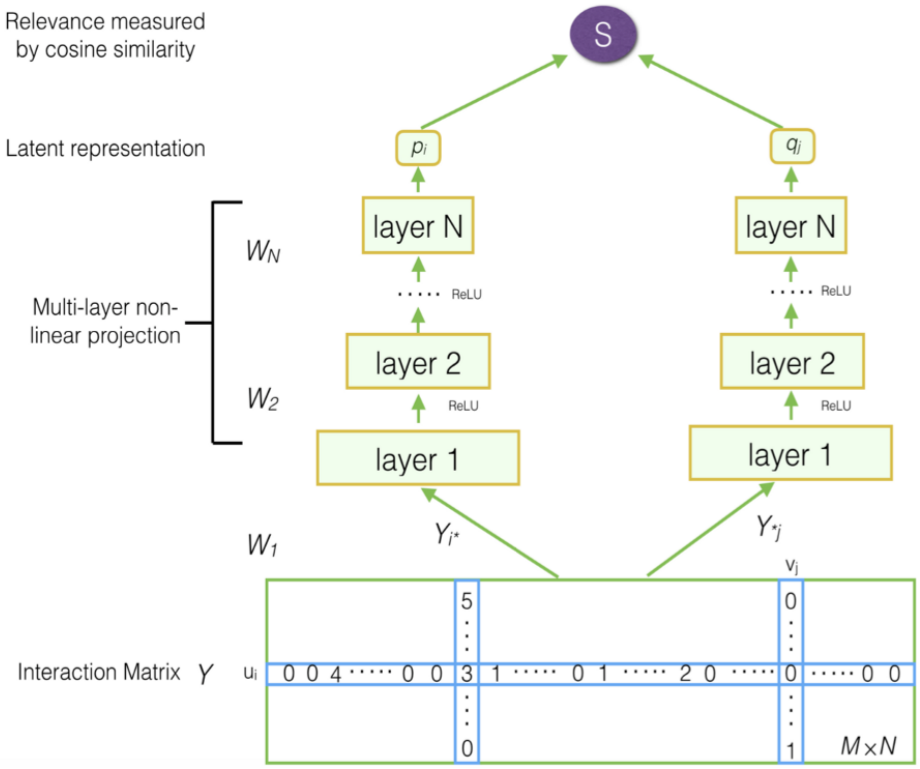


Figure 3.1: Deep Matrix Factorization: The neural architecture of the DMF model (source: Xue et al. [2017]).

The dimension of each layer of the MLP_U network does not need to match the dimension of the same layer on the MLP_I network, although the output dimension of the last layer of both networks must be the same, due to how the model's predicted value is computed. Once each input vector reaches the end of network, it is transformed into a latent vector: $p_u = MLP_U(Y_{u,*})$ is the user latent vector and $q_i = MLP_I(Y_{*,i})$ is the item latent vector.

The model predicted value $\hat{y}_{u,i}$ is computed by taking the cosine similarity of both latent vectors, as denoted in equation 2.3. Since the cosine similarity ranges from $[-1, 1]$, it is possible that negative predictions occur, so it is also necessary to set prediction values to a very small positive number (e.g. $1\text{e}{-6}$) whenever they are negative (so that the loss function remains unaffected). The final predicted value is computed as follows: $\hat{y}_{u,i} = max(cosine(p_u, q_i), 1\text{e}{-6})$.

3.2.1.2 Activation Functions

As described in the model's original publication, the activation function to be used between each layer should be the ReLU activation function. This function is denoted in equation 3.1.

$$\sigma(x) = \begin{cases} x & \text{if } x > 0 \\ 0 & \text{otherwise} \end{cases} \tag{3.1}$$

3.2.1.3 Loss Function

DMF uses a point-wise loss function called (binary) cross-entropy, as shown in equation 2.38. There are actually two possible variants that completely affect the model behavior: the DMF-ce, using cross-entropy and the DMF-nce, using a normalized cross-entropy loss. This final variant consists of scaling the desired value $y_{u,i}$ to the $[0, 1]$ range.

The goal of the optimization process is to minimize the resulting loss values by applying the backprop algorithm to get the gradients for each network and adjust their weights accordingly.

3.2.2 Collaborative Denoising Autoencoder

The Collaborative Denoising Autoencoder (CDAE) was introduced by Wu et al. [2016], and it uses a modified denoising autoencoder, that incorporates an extra node on the input layer, that changes accordingly to the user for which the prediction is being made.

Besides the extra user-specific node, the input layer also receives the corrupted interaction vector of that user. The corruption level is controlled by a model hyperparameter q, and this input is drawn from a conditional distribution $\tilde{y} \sim p(\tilde{y}|y)$, where:

$$\begin{aligned} p(\tilde{y}_u = \delta y_u) &= 1 - q \\ p(\tilde{y}_u = 0) &= q \end{aligned} \tag{3.2}$$

where $\delta = (1-q)^{-1}$, so that the corruption is not biased, due to some of the dimensions being overwritten with 0.

3.2.2.1 Layer Structure

As this model incorporates a denoising autoencoder, it must consist on three layers only: an input layer with $|I|$ nodes (which receives the corrupted input), a hidden layer that must have a reduced dimensionality K, where $K < |I|$, and then an output layer with the same dimension as the input layer. Figure 3.2 represents the model's structure, also including the user-specific node.

Note that this user node is connected to every other unit from the hidden layer, therefore we must have an additional matrix $V \in \mathbb{R}^{|U| \times K}$, where the V_u entry corresponds to the u-th user-specific node. This user latent vector is the key difference between CDAE and a simple DAE.

To compute predictions for the i-th item with respect to a given user u, we provide CDAE with the u-th user corrupted interaction vector, along with V_u, and the i-th element of the output layer corresponds to the model's prediction for the u, i pair. For out-of-training predictions, note that the provided input vector should not be corrupted.

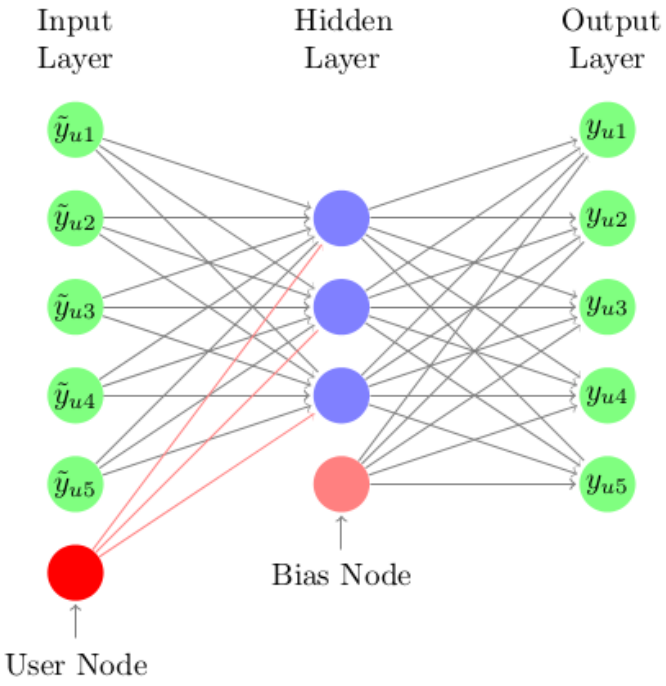


Figure 3.2: Collaborative Denoising Autoencoder: The neural architecture of the CDAE model (source: Wu et al. [2016]).

3.2.2.2 Activation Functions

The original publication discusses several options for the activation functions used by CDAE, such as the identity function 2.35, the sigmoid function 2.36 and the hyperbolic tangent function 3.3.

$$\sigma(x) = \frac{e^{2x} - 1}{e^{2x} + 1} \tag{3.3}$$

3.2.2.3 Loss Function

On the evaluation experiments presented on the CDAE publication, four variants are compared, where two distinct loss functions are used: the square loss function 2.37 and the log loss/binary cross-entropy function 2.38. It is also mentioned that there should be no reason for CDAE to not support other loss functions, provided that they are differentiable and allow for the model to adjust and improve its behavior.

3.2.3 Caser

Caser Tang and Wang [2018] is a sequence-based RS that uses convolutional layers to process the outputs of the embeddings produced by a sequence of user-interactions. Then, it passes the signal through a

feedforward neural network, to produce predictions for the next items that are most likely to be of use to the user. This RS uses a list-wise sampler, since a list of data-points is required to compute a prediction, specifically items that the user has previously interacted with.

3.2.3.1 Layer Structure

This model consists on three components: embedding look-up, convolutional layers, and fully-connected layers.

After sampling L items that a given user has interacted with before, an embeddings matrix $E \in \mathbb{R}^{L \times d}$ is computed. This matrix is used as input for two types of convolutional layers: an horizontal convolutional layer, to compute union-level patterns, and a vertical convolutional layer, to compute point-level sequential patterns. To compute the predictions, the outputs of these convolutional layers are concatenated and processed by two fully connected layers, also making use of a user embeddings vector $P \in \mathbb{R}^{d}$, and resulting in a prediction vector $\mathbf{y} \in \mathbb{R}^{|I|}$.

At prediction time, we take the T items that obtained the largest prediction values. Figure 3.3 represents the overall model architecture, separated by the three components.

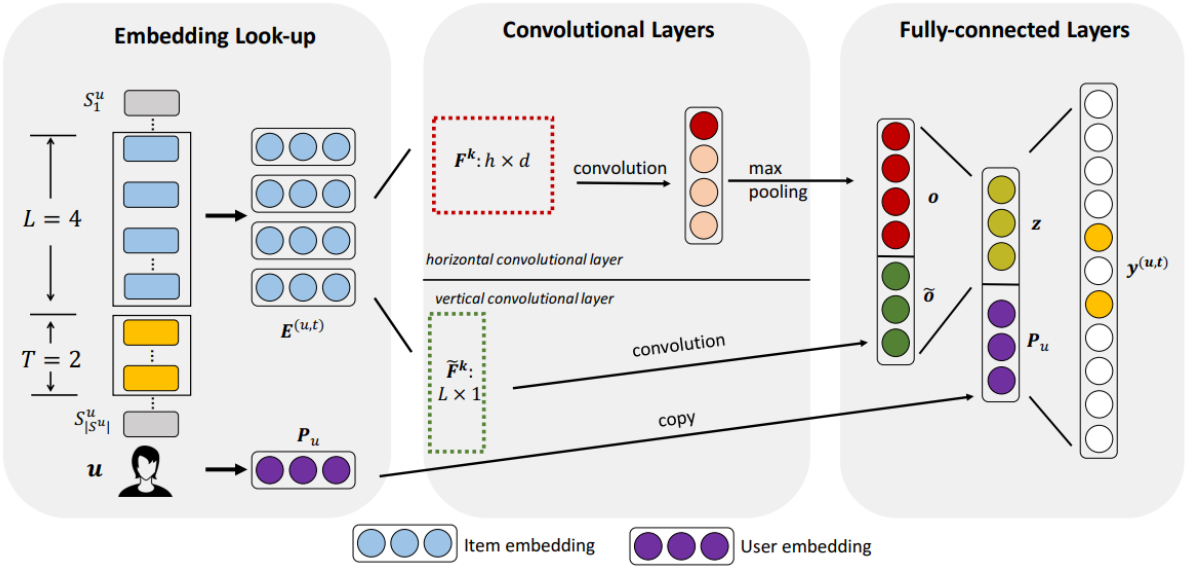


Figure 3.3: Caser: The architecture of the sequence-aware Caser model (source: Tang and Wang [2018]).

Negative feedback is introduced by defining a factor dependent on the number of target predictions T, which can be defined as $T^- = \eta T$. It is expected that the predictions of the T target items approach 1, and that the predictions of the T^- negative items approximate 0.

3.2.3.2 Activation Functions

There are two activation functions that must be defined for this model: for the horizontal convolutional layer, and for the feedforward layers. The authors evaluated multiple functions on their practical experiments, such as the hyperbolic tangent 3.3, identity 2.35, the sigmoid 2.36 functions, but the one that obtained better results was the relu 3.1 function.

3.2.3.3 Loss Function

The loss function used by Caser is the categorical cross-entropy function 2.39. As mentioned before, each prediction over a target item in T is expected to approximate 1, and all predictions over negative items are expected to approximate 0, which makes Caser work very well with the categorical cross-entropy loss function.

Chapter 4

DRecPy: A Deep Learning Recommendation Framework for Python

In order to tackle the various issues described previously, a new open-source (MIT License) Python framework - **DRecPy: Deep Recommenders with Python** - is introduced. These previously mentioned issues will now be discussed in a more granular manner so that the reader is able to understand the practical improvements achieved by using the framework. This toolkit aims to assist both researchers and practitioners of this field, by easing the development and evaluation of new models, but also by providing various state-of-the-art recommender models out-of-the-box.

A brief overview of the module structure of DRecPy is shown in Fig. 4.1.

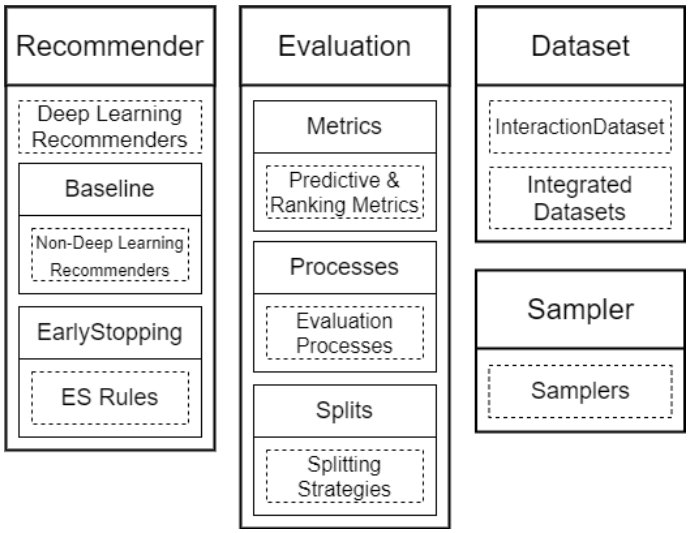


Figure 4.1: DRecPy Module Structure: DRecPy is divided in four modules: Recommender, Evaluation, Dataset and Sampler. All these modules are detailed in the following sections.

4.1 Handling Data Sets

Pre-processing data sets before training and testing a model is one of the most tedious and repetitive tasks of the current development pipeline. Even though some domain-specific processing might still be required, there are some steps that can be generalized. Implementing support for these general cases manually and individually for each developed recommender would bring an undesired time-cost overhead.

To this end, DRecPy has a module called `InteractionDataset`, which handles the import process from a CSV [1] file, allowing easy and fast filtering. It also supports loading data from a Pandas dataframe[2], one of the most popular frameworks in the Python scientific community. It is worth mentioning that this module also provides additional methods that are often not found on any particular recommender framework, such as selecting unique records based on the given columns, apply queries to the data set, return user/item interaction sparse vectors, and to save any given record state into a file. More details about the contributions of this module will become clear once we discuss each particular challenge. This also allows for other operations to be abstracted of the data set dimension, as shown in Figure 4.2.

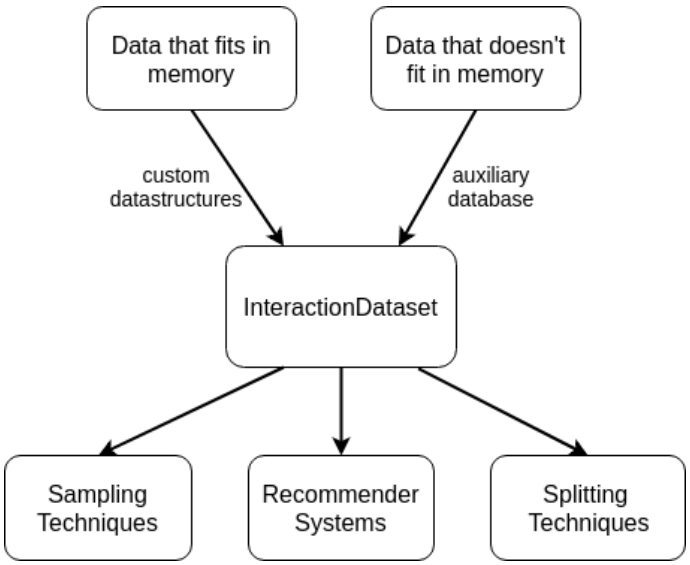


Figure 4.2: Interaction Data Set Overview: By abstracting in-memory and out-of-memory data sets in one data type, every other data-related process will be simplified, instead of requiring the implementation of specific behaviours for each of the two cases.

[1] Comma-separated values file format.

[2] https://pandas.pydata.org/pandas-docs/stable/reference/api/pandas.DataFrame.html

4.1.1 Raw to Internal Identifiers

Collaborative filtering based models use interaction data sets, containing at least (but not limited to) triplets of user, item, and interaction[3] fields. This data can be represented in a matrix format, where each field is represented in a specific column, and each record in a separate row. The format and types of the user and item columns are not controlled by the practitioner that is trying to make use of this data, so there can be a few issues when trying to use this raw representation:

- User and item columns might not be numeric[4] fields, which makes it imperative to convert them for further use on training and testing processes;
- Even numeric types can be an issue, if they are not consecutive and in a defined range, certain operations might become too time-consuming and hard to achieve, for example: embedding-based models require the users/items to be in a $[0, N]$ range; retrieving user/item vector representations for vector-based similarity measures; and to sample positive and negative interaction pairs efficiently.

DRecPy deals with these raw to internal id conversions without the need of any specific user instruction. Custom raw to internal mappings are automatically built, and whenever iterations or filters are made on a specific user or item (e.g. selecting all ratings provided by user u), these structures are used to speed up those processes. For the practitioner, the only requirement is to load the data set using this module; for the researcher/developer, the identifiers passed to the predict or rank methods are all automatically converted to internal ids, so that id conversion-related problems can be abstracted and no longer be an issue.

4.1.2 Flexible Column Structure

Most existing recommender toolkits assume that interaction data sets consist of only the 3 mentioned fields, and that leads to another common issue: how to develop RSs based on extra fields? Multiple systems based on additional interaction fields have been proven to show significant improvements, such as: session-based models Hidasi et al. [2015], sequence-based Donkers et al. [2017] and tag-based models Zuo et al. [2016]. Supporting these additional fields, as shown in Figure 4.3, such as session ids, tags, timestamps, location, used platform, etc., would bring additional support and development ease for these more complex models.

The way that was chosen to implement this feature is to request the practitioner a list of columns, where each represents a column on the data file, and then build a flexible mapping structure to support these multiple fields on each interaction. Then, whenever a record has to be retrieved, this structure is returned, instead of a fixed number of parameters. These extra fields also persist in each data subset, so that we can support filters and fast retrieval of these columns.

[3] The interaction field can be a rating provided by a user to the item, or simply a binary value indicating if the user has interacted with the item or not.

[4] Usually floating-point numbers are not frequent on these identifiers, but if these occur, they might also require conversion.

Figure 4.3: Flexible Structure: Any number of columns can be loaded and used for query purposes. It is also possible to remove columns from a data set instance.

An alternative method to implement this would be to store these extra fields in a different structure and create a mapping from ids to these values. But after a few tests, the previous alternative was chosen due to the fact that if the practitioner is loading these extra columns, then s/he will probably access them whenever a record is retrieved, so it makes sense to store these together, although an overhead cost is expected.

4.1.3 In-Memory Data Sets

For in-memory data sets, the framework uses the famous Pandas dataframe as means to store the data. Since the dataframe doesn't provide the most efficient way to iterate and query data sets, auxiliary structures are built to fasten raw to internal id conversion, iterate through the record set, as well as to access all the ratings for a given user/item. These structures map raw ids to internal ids, internal ids to raw ids, and internal ids to associated records. There is indeed an initial overhead for building these mappings, although it provides faster overall runtimes. For querying the data set using other fields (such as ratings, timestamps, or other extra fields), boolean indexing[5] is used, as it shows to improve performance significantly.

Tests were made to try to incorporate the Modin[6] dataframe, which shares most of the Pandas API but provides parallel computation by splitting and distributing each dataframe for each available CPU core. Although it looks like a very promising project, it still has some issues on low-memory machines, most likely because it's still a relatively new project (the first version was launched at 7/07/2018). Another drawback is that it still does not offer easy compatibility for Windows-based machines, and being one of the crucial objectives of this work to provide a toolkit available for all, Modin could not be adopted at the moment.

4.1.4 Out-of-Memory Data Sets

Another problem that is mitigated by this framework, is handling out-of-memory data sets, that is, data sets that do not fit into memory. At the moment, the easy alternative is to iterate through the data

[5] https://pandas.pydata.org/pandas-docs/stable/user_guide/indexing.html#boolean-indexing
[6] https://github.com/modin-project/modin

set in chunks, whenever an operation is applied. This quickly becomes a very repetitive work and slows down the whole process. To offer an alternative, faster, and more convenient way to do this, the `InteractionDataset` module will use a database engine to store the processed data set whenever it doesn't fit into memory, without bothering the practitioner with details on how it is done internally. It also automatically builds the required indexes to support fast document retrieval.

This concept of building a new sub-module that shares the same interface and behaviors as the in-memory data set, presented two challenging tasks:

1. ***How to handle data set selections***: if we have a data set instance $\mathcal{D}$, and want two new separate instances $\mathcal{D}_{u_1}$ and $\mathcal{D}_{u_2}$, each containing the records of user u_1 and u_2 respectively, we would have to copy these results into 2 separate databases (or tables). This would obviously become too slow to be viable for any practical usage, so another approach was taken;

2. ***How to create databases without requiring any action from the user***: a database needs to be created in order to store these records, but it shouldn't add any complexity layer or configuration requirements for the practitioner, so a simple and transparent database engine needs to be used.

4.1.4.1 How to handle data set selections

Lets first introduce a few concepts that will help us understand the taken approach:

- ***Main table*** **($\mathcal{T}_{main}$)**: Represents the main database table where the whole data set is loaded into. Every row in the data file has an associated record on this table;

- ***Reduced table*** **($\mathcal{T}_{red}$)**: Is a temporary table that is created whenever the number of selections is big enough that justifies the creation and insertion of a certain set of records that are present on the $\mathcal{T}_{main}$ onto a different table - this behavior is called *path compression*; an out-of-memory `InteractionDataset` instance might be using the $\mathcal{T}_{main}$ or a $\mathcal{T}_{red}$ to do operations on the data set, depending on the circumstances - this will become more clear as we progress;

- ***Active table***: Reefers to the database table that is being used by an instance. The first-ever instance to be created will always have its active table to be the $\mathcal{T}_{main}$. After a few selections, new instances will be created, and at some point, a $\mathcal{T}_{red}$ might be constructed and assigned as the instance's active table;

- ***State query***: Each instance has a state query, and whenever an operation is applied, such as a selection, instead of it being instantly evaluated, by retrieving the result set and storing them on a $\mathcal{T}_{red}$, it will instead append a new instruction to the state query of the current instance. It represents the required queries to apply to the instance' active table in order to reach its segment of the data set.

Example 4.1.1 Initially $\mathcal{D}$ has the active database table set to the $\mathcal{T}_{main}$ and its state query is empty; $\mathcal{D}_{u_1}$ and $\mathcal{D}_{u_2}$ instances also share the $\mathcal{T}_{main}$ as the active table, but with a different state query ($\mathcal{D}_{u_1}$ has a "$u = 1$" state query, whilst $\mathcal{D}_{u_2}$ has it set to "$u = 2$"). If now we do a rating selection on $\mathcal{D}_{u_1}$, a new instance $\mathcal{D}_{u_1,r_{>2}}$ is created. For this new instance, its state query is empty and a new $\mathcal{T}_{red}$ is assigned to be its active table, by evaluating the state query "$u = 1$" on the $\mathcal{T}_{main}$ (i.e. apply the parent state query on its respective active table). This example is visually described in Figure 4.4.

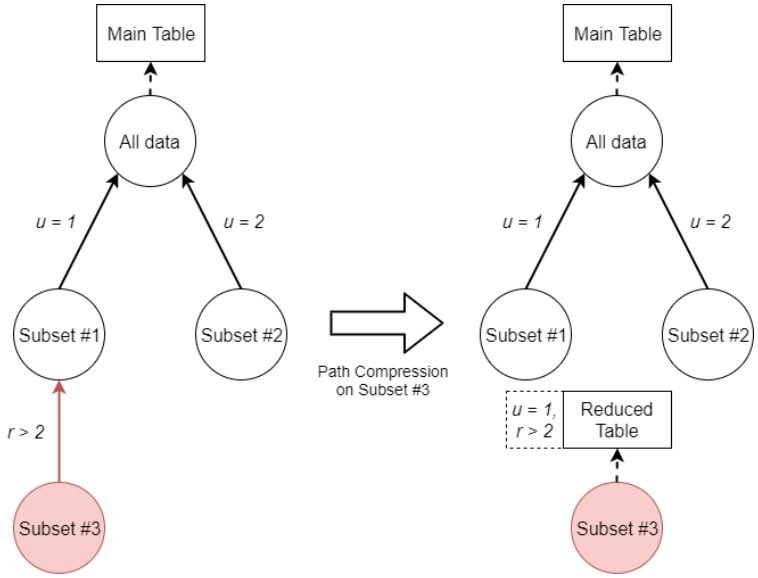


Figure 4.4: Out-of-Memory Interaction Data Set: This figure depicts the path compression example 4.1.1. After the second consecutive filter, the module decides to do a path compression on the newly created node, by migrating the state query results to a new table.

Of course that example 4.1.1 describes a very simplistic scenario, and in a real environment, the number of created instances will be much higher and most of the time the path compression process will not take place right after the second consecutive selection. The creation of reduced tables is not random, but instead based on the combination of some heuristics, such as the number of consecutive selections, the percentage of rows removed by applied filters, and the number of direct interactions with a given `InteractionDataset` instance.

4.1.4.2 How to create databases without requiring any action from the user

There are multiple popular database engines such as MySQL[7], PostgreSQL[8] and MongoDB[9] which are very optimized for read and write operations. Although, all of these require a server to be used, which is

[7] www.mysql.com
[8] www.postgresql.org
[9] www.mongodb.com

not optimal for what this framework is trying to achieve, which is a user-friendly and zero-configuration toolkit that helps to focus on the most important objective: developing and using RSs. After a brief investigation on the available tools for achieving this goal, the SQLite[10] looked like the most promising engine for this use-case, for various reasons:

- Serverless and self-contained database engine;
- Zero-configuration is required to get a database up and running;
- One of the most active developer communities, which allows it to be continuously updated and optimized;
- Stable and reliable codebase that is being maintained since the 2000s;
- It is schema-based, which is the most appropriate type for this application (since columns and its types are known and well defined from the start);
- A very popular Python module sqlite3[11] comes installed by default.

Although SQLite does not support concurrent writes (which could be useful to speed up importing processed data sets), its speed is still pretty comparable to the ones provided by other engines on similar use-cases. All of this together led to the decision of using this database engine for the support of the out-of-memory data set implementation.

As most database management systems (DBMSs), they come with features to enable atomic commits and fault tolerance, so that in case of a power failure, operating system (OS) crash, application crash, or hardware failure, the database can be restored to its last valid state (i.e. rollback operation). For most applications, this is a fundamental feature, as it ensures that the database will always be in a valid state, so the applications do not need to worry about most of these problems. This is achieved through the creation and maintenance of a rollback journal (created when a new transaction begins and deleted at the end) or of a write-ahead log (WAL) file (created when the connection is established and deleted on disconnect); both these auxiliary files allow us to recover from an invalid state to the most recent valid one.

Despite these features being enabled by default, for our use-case, there is no need to be fault-tolerant, because the data set will also be stored in another file (e.g. CSV file). For that reason, and to speed up every out-of-memory data operation, these features are disabled by setting the journaling mode to ”off” (disabled rollback journal), the synchronous flag to ”off” (does not sync data handling with the OS), and by increasing the cache size for each database from the default 2000 to 4000 KiBs (4096000 bytes). These settings improved the performance by reducing the import execution time by $\approx 15\%$ and iteration execution time by $\approx 15.5\%$.

[10] www.sqlite.org
[11] https://docs.python.org/3.4/framework/sqlite3.html

Another very important performance tuning that was made, was to disable all indexes before the initial import process, and disable internal identifier indexes before these are assigned, and re-enable them after these procedures. These changes resulted in the performance improvement of the import process by $\approx 27.8\%$ and $\approx 35.5\%$ in the process of assigning internal identifiers.

4.1.5 Data Manipulation Methods

In order to speed up some other very regular data operations used on the environment of recommender systems, the `InteractionDataset` module makes available various operations, such as computing min and max values for a given column, computing unique values, select random positive and negative interaction pairs, modify column values given a function and methods to compute user/item interaction vectors.

All these operations were implemented to support built-in features, such as samplers or recommender models, so they were moved into the data set module in order to simplify these and to provide helpful methods for practitioners and researchers.

4.2 Sampling Data Points

During model training, there is the need to sample a set of data points, denominated as the training batch. This training batch can be computed in various ways, but it usually follows one of the two alternatives discussed in 2.1.6. For that reason, DRecPy has a `Sampler` module that provides the tools required to accomplish this task. All implemented sampling strategies take into account that not all possible user-item pairs are present on the data set, and there is a parameter to control the percentage of sampled negative interactions for each positive one.

4.2.1 Point-wise Sampler

The point-wise strategy focuses on sampling a user-item pair individually. This is the most common type of sampling, where the recommender requires N independent data points and makes a prediction for each.

The implementation of this sampler not only takes into account the negative ratio passed as parameter, but also the interaction threshold to be used when deciding which data points are considered to be negative, and which are considered to be positive. When no interaction threshold is provided, the negative instances are those that are not present in the data set and the positive are those that are; when it is provided, all interactions with an interaction value/rating bellow the provided threshold are considered to be negative, and all the others are considered to be positive interactions. A seed parameter is also available, so that the sampled data points can be computed deterministically.

This sampler provides several methods: sample N independent mixed (containing positive and negative) interactions; sample one positive or negative interaction; sample one positive interaction; or sample

one negative interaction.

4.2.2 List-wise Sampler

The list-wise sampling strategy samples a set of points that are somewhat related. This is widely used for sequence-aware RSs that require $N \times M$ data points, where N represents the number of groups/sequences, and M the number of related interactions per sequence.

The implemented list-wise strategy receives an argument that represents the key to be used when grouping a set of records (the key is a set of column names), as well as an optional parameter that represents the column to be used when sorting the records inside the sampled sequence. When a sequence needs to be generated, a random group is chosen and the positive records that belong to that group are sorted. If the number of target records is provided, then each group will contain not only a sequence of positive sorted records, but also another sequence of target records, that come after the positive records according to the sort column, and a list of identifiers representing negative interactions. The number of negative interactions is always proportional to the number of target records, by a factor that depends on the adjustable negative ratio parameter.

An example would be when we need to sample N user-sessions, each containing M records sorted by timestamp, where L is the number of non-target interactions and T the number of target interactions, with $M = L + T$. When used with a negative ratio set to η, the result of running DRecPy's list-wise sampler will be a list containing N elements, where each has two sequence lists of positive interactions with size L and T respectively, and an identifier list of size ηT, with some of the identifiers of the items that the user did not interact with during that session.

4.3 Building and Training Models

Building new models with DRecPy requires extending a recommender base class (`RecommenderABC`) and implementing a few methods to initialize and train the model, as well as making predictions. Training a model requires its initialization, followed by a fit process that is highly customizable.

4.3.1 Recommender Structure

Understanding the structure of a recommender is key to be able to implement new models, or extend existing ones. DRecPy provides a base abstract class called `RecommenderABC`, that contains the structure that each recommender should follow. Once a novel recommender extends this class, it should implement a few methods that are never called by the outside user, but are instead called by the base class whenever needed. There are two types of methods present on each DRecPy recommender: generic methods, which are the ones that are implemented on the recommender base class and therefore shared between each model; and the specific methods, that should be implemented individually for each model. This is done

so that the basic workflow, as shown in Figure 4.5, is known and no unexpected behavior occurs, but also to support various out-of-the-box features on every recommender.

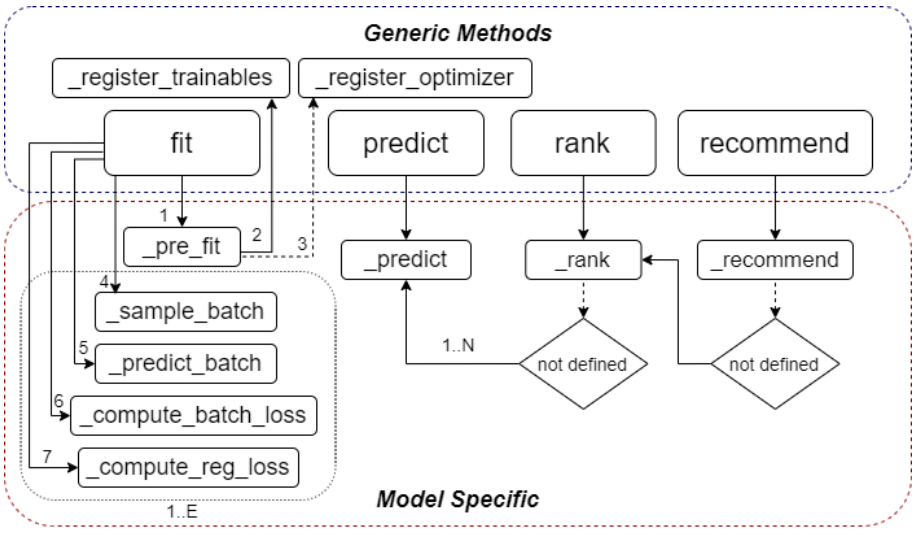


Figure 4.5: Recommender Call Workflow: Each model has generic and specific methods, where generic methods are shared between all recommenders, and specific methods should be implemented for each model. The fit workflow starts by executing the pre-fit method, to create and initialize custom data structures, and then it repeatedly calls the specific sample batch, predict batch, compute (predictive) batch loss, and the compute regularization loss methods, for each training epoch. If the model is not based on a deep learning approach, these last four steps are skipped, and only the pre-fit method is called.

Most generic methods rely on specific methods to accomplish their task, but they also take care of some details to reduce the work required for the latter methods. Not all specific methods are required, for example, the `_rank` method can make use of the `_predict` method to return the appropriate item rankings - the generic `rank` method is responsible to deal with these cases. Another example is if the specific `_recommend` method is not implemented, in which case the `_rank` method is used to rank all the items for a specific user - here, the generic `recommend` method is responsible to ensure this behavior. The most important generic methods that do not depend on specific methods are the `_register_trainables` and `_register_optimizer`, where the first should always be called during the execution of the specific `_pre_fit` method, with all the trainable variables, and the last is an optional call that should be made whenever the model implementation requires a specific optimizer to work (the base class already provides a default optimizer that uses the user-defined learning rate). The back-end tool used by DRecPy to compute gradients and apply weight updates is TensorFlow Abadi et al. [2016], one of the most popular deep learning frameworks on the Python ecosystem.

The recommender base class has the following responsibilities:

- **Abstract raw id to internal id conversion**: All identifiers that are sent to model specific methods are internal identifiers. This ensures that these specific methods do not have to deal with id

conversion, which could result in unexpected behaviors.

- **Handling invalid identifiers**: Whenever invalid user or item identifiers are passed to predict, rank, or recommend methods, these are filtered out before being passed to the specific methods (or an exception is raised, depending on the passed configurations). This also simplifies the specific model implementation, which reduces the probability of eventual errors and speeds up development.

- **Tracking losses per epoch**: The generic training process takes care of tracking the model's loss per epoch. This feature is further described in section 4.3.2.

- **Automatic Weight Updates**: During model training, all registered trainable variables are traced by a TensorFlow gradient tape, allowing gradients to be computed from the loss value returned by the recommender. At the end of each training batch, these gradients are used to update all the registered weights, improving model predictive performance.

- **Epoch callbacks and Early Stopping**: Executing a function to evaluate the model's performance on a validation set after each epoch (or for each X epochs), allows the training process to use techniques such as early stopping, described in section 2.3.4.1. More about epoch callbacks in section 4.3.3.

- **Interaction value standardization**: Utility functions are provided by this base class to standardize and re-scale a given interaction value/rating. This might be useful for models that require interaction values in the $[0, 1]$ range Xue et al. [2017].

- **Save / load models**: Functionality to save and load models is also provided, as described in section 4.3.5.

- **Progress Logging**: During the training process, messages are displayed with its progress, loss values, and other information related to the training set.

While this described structure is very simple, it still allows the creation of complex recommenders, since there is very little enforced behavior on each implemented model. Some other abstractions were also thought, like removing the sampling of training batches from the model completely, by delegating that responsibility to the recommender base class. Because some models require distinct sampling techniques, this could lead to some incompatibilities, therefore each model implementation should make use of sampling tools provided by DRecPy (or manually developing them).

The overall solution is to try to abstract all the possible steps without limiting the potential of each model (via inheritance), while providing all the tools required for each model to contain only the strictly necessary logic (via composition).

4.3.2 Loss Tracker

The base recommender takes care of storing the various losses resulting from the whole training process: losses originated by the model's objective function, as well as validation losses that might exist, computed via epoch callbacks.

At the end of the model training, a plot containing all the stored losses is displayed, allowing the developer to concentrate on model-specific challenges, instead of repeatedly creating the logic required for this type of behavior. This feature is automatically supported by using the return values from the model's `_compute_batch_loss` and `_compute_reg_loss` methods.

An example of such plot is shown in Fig. 4.6, displaying the model's loss over epoch on the top, with the validation losses (hit ratio and normalized discounted cumulative gain) on the bottom graph.

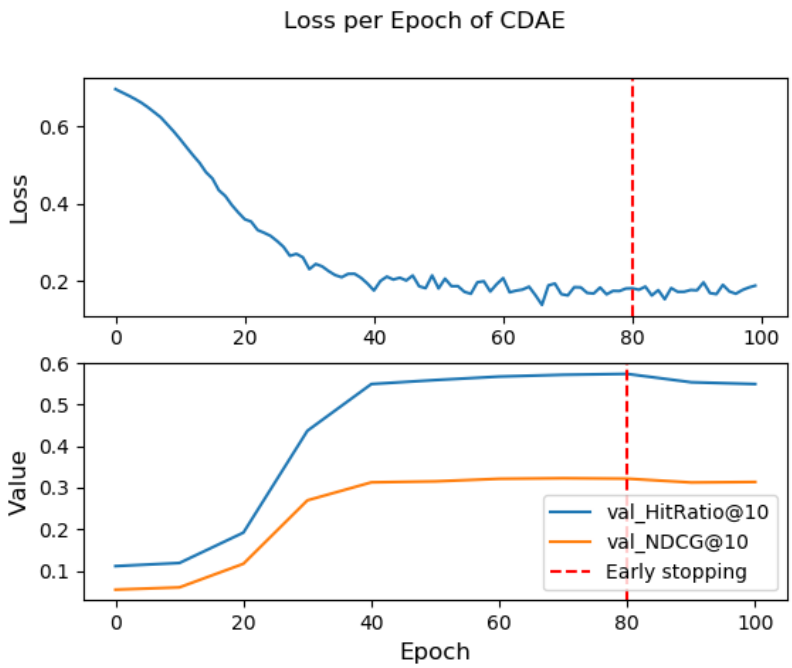


Figure 4.6: Loss Tracker Plot: The top graph of this plot shows the model's loss value for each epoch, and the bottom graph displays the computed validation losses, with the boundary computed by the early stopping rule that separates the section in which model performance increased during training, from the section that it decreased. This plot is obtained by training a CDAE recommender on the MovieLens-100k data set for 100 epochs, computing the hit ratio and normalized discounted cumulative gain metrics on the validation set for each 10 epochs, with an early stopping rule that finds the maximum validation value across the whole training.

4.3.3 Epoch Callbacks and Early Stopping

Epoch callbacks are functions that are called during the model training process, allowing greater visibility over the evolution of its performance. Although epoch losses should track the improvement level a model, that is not always the case, e.g. when the model starts overfitting the training data.

DRecPy allows the experimenter to inject custom validation functions into the training pipeline so that relevant metrics can be computed on a validation set. The frequency of these evaluations can also be tuned, allowing the user to control the trade-off between training time and introspection of the training model, i.e. more frequent computations of epoch callbacks increase the total training time. This feature also allows the experimenter to define early stopping rules based on the values of these callbacks, preventing the model from overfitting the data. These early stopping rules compute whether the model training process should be interrupted, and the epoch that the model has shown better performance. If the best-computed epoch is not the final one, then DRecPy reverts all the model's weights to those of the best epoch.

An example plot obtained by evaluating an epoch callback that computes two custom metrics (HR and NDCG) with a frequency of 10 epochs, and with an early stopping rule that never stops model training but computes the best epoch according to the best validation performance of the model, is shown in the bottom part of Fig. 4.6.

4.3.4 Extending Existing Models

Exploring possible variations of existing recommenders can lead to interesting discoveries. To further motivate this practice, DRecPy makes it straightforward to extend an existent model by adding custom functionality, e.g. adding an extra network would require to create a subclass of the existent model, override the `_pre_fit` method to register the new network as trainable and call the overridden method so that previous initializations are maintained, and then override the `_predict_batch` in the same manner, using the predictions of the previous model to pass them through the new network.

Other small adjustments are even easier, such as experimenting with new sampling strategies, or new predictive/regularization loss functions, which only require one method to be overridden. This is possible due to the structure of the call workflow that is controlled by the base recommender class. An example use-case of extending an existing model by adding custom logic will be evaluated in 5.2.

4.3.5 Saving and Loading Trained Models

Deploying and sharing a trained model is usually not supported by many frameworks. DRecPy allows saving the model's state into a persistent storage so that it can be shared, deployed, and possibly incorporated in an API to reply with recommendations for incoming requests. Although DRecPy facilitates saving and loading trained models, it currently has some limitations that are often not suitable for medium to large scale systems.

Since recommenders require access to the training data to query for user/item interactions, or other information required for making predictions, a copy of this training data is also persisted alongside its state. This forces the developer to always keep the saved model state and data set together so that it can be loaded correctly. This limitation is not ideal for large scale deployments, since these setups usually consist of multiple distributed machines, and using this approach, multiple copies of the same data set would have to be spread among distinct machines. A better approach would be to decouple the system that loads and allows queries on the data set, from the system that contains the logic to provide recommendations. This is one of the reasons that make DRecPy, in its current state, not ideal for medium to large scale applications.

4.4 Model Evaluation

To evaluate a recommender one first needs to compute a test or validation set, by applying one splitting technique, followed by applying the desired evaluation method, which computes the chosen metrics and averages them across the number of tests or validation data points. DRecPy offers an `Evaluation` module that contains all the logic to evaluate any recommender, consisting of three sub-modules: Splits, Processes, and Metrics. In the following subsections, we explain in more detail the responsibilities of each sub-module.

4.4.1 Data Set Splitting Techniques

The Splits sub-module contains all the logic to split a data set into a train and test set, or a train into another train and validation set, and currently it has three splitting methods implemented: random split, matrix split, and leave k out (see section 2.1.7).
Since these splitting functions return instances of `InteractionDatasets`, they can be used iteratively in order to compute validation sets.

Each splitting function also receives two additional arguments: the seed value and the maximum number of concurrent threads. The first argument is used in order to always compute deterministic splits, therefore improving the reproducibility of every evaluation (and training) pipeline. The second argument is used to improve performance and should be set according to the desired level of parallelization, never exceeding the number of available cores on any given machine.

Implementing a new splitting technique is also very simple since it only consists of a simple function that receives, at least, the data set in which the split will be performed, the two arguments stated before, and arguments that are specific to that technique. To leverage flexibility on the types of splitting functions that can be implemented in DRecPy, no rigid structure is enforced.

4.4.2 Evaluation Methods

To evaluate a recommender on a given test set, the Processes sub-module has three evaluation processes implemented, namely: predictive, ranking, and recommendation evaluation.

The predictive evaluation simply evaluates the model's predictive value against the expected interactions/ratings, by using the mean squared error and the root mean squared error as default metrics. The ranking evaluation compares the expected ranking of a set of items for a user, against the ranking provided by the model, while the recommendation evaluation compares the top k items provided by the model, against the ones expected for each user. The default metrics for the two last processes are: precision, recall, hit ratio, and normalized discounted cumulative gain. Custom metrics can be easily injected in order to obtain custom evaluation behavior as well. Automatic sampling of negative instances for each user can also be enabled, to support highly adopted evaluation setups Rendle et al. [2012]; He et al. [2017].

4.4.3 Evaluation Metrics

As explained before, evaluation methods make use of metrics to be able to compute how good is a recommender doing overall. For that reason, it is necessary to have a good amount of implemented and well-tested metrics, with a structure that allows creating new ones that can be incorporated in the evaluation pipeline. For that reason, DRecPy has a Metrics sub-module, which implements two types of metrics: predictive and ranking metrics.

Predictive metrics are aimed at evaluating the predictive performance of a recommender, that is, how good are the interaction values predicted by the recommenders, against the real rating provided by a user, to an item. The predictive metrics implemented so far are: mean squared error, root mean squared error and mean absolute error.

Ranking metrics evaluate how good are the recommendations provided against the relevant items. The ranking metrics already implemented in DRecPy are: hit ratio, discounted cumulative gain, normalized discounted cumulative gain, reciprocal rank, recall, precision, f-score, and average precision. All these metrics can also be computed in lists of truncated recommendations.

Chapter 5

Evaluation

This chapter focuses on the evaluation of the proposed framework, by comparing the results produced by some of the implemented recommender models, against previously published results. After evaluating the consistency of the framework, a full feature comparison will be presented, by analyzing DRecPy's functionalities against alternative tools.

5.1 Consistency with Published Results

One of the most important points for any new framework is to be consistent with the already proven results, especially on the Machine Learning field, where the final outcomes vary widely, due to different implementation details, non-fixed seed values for random generators, and other domain-specific factors. Even though it's unlikely that these tests will provide the same exact results, they shouldn't differ significantly from the previously reported ones.

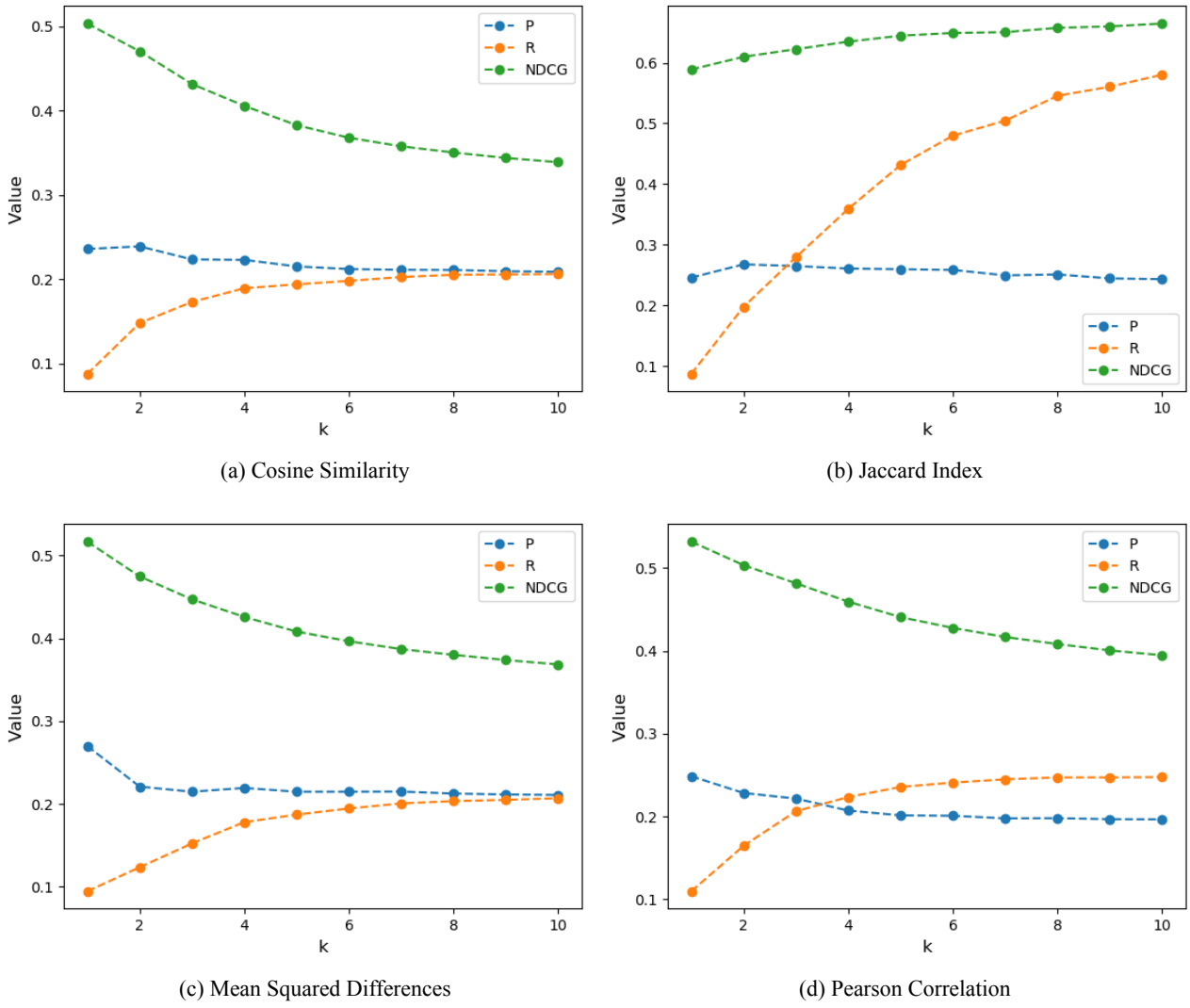
For this reason, some tests were made in order to compare the performance of the evaluation processes and recommender models provided and trained using the DRecPy framework. All the scripts built for the purpose of producing the results shown on the tests below can be found on the project's GitHub repository, specifically on the consistency evaluation examples folder (`https://github.com/fabioiuri/DRecPy/tree/master/examples/consistency_eval`).

5.1.1 Baseline Recommenders

These tests target baseline recommenders (non-DL based), and follow the evaluation process described in Barros et al. [2019], on the MovieLens-100k data set which is described in section 2.1.4.

Since not all of the similarity functions are implemented in DRecPy, a subset was selected, for training a user-based KNN model: cosine similarity, Jaccard index, mean squared differences, and the Pearson correlation. Note that similarly to the knn models mentioned on the reference publication, the ones used by DRecPy are also trained with the k parameter (number of neighbors) set to 10, without shrinkage, nor a number of minimum co-ratings to consider the similarity between 2 users (usually denoted as $m = 0$).

All the methods used for the retrieval and split of the data set were also implemented with calls to DRecPy's built-in functions. Each evaluation process applied by the framework generated a plot, which is visible in Fig. 5.1.



(a) Cosine Similarity

(b) Jaccard Index

(c) Mean Squared Differences

(d) Pearson Correlation

Figure 5.1: Performance of the UserKNN models trained on the MovieLens-100k data set.

Tables 5.1, 5.2, 5.3 and 5.4 show the reported values in Barros et al. [2019], together with the obtained values and their respective differences, for each trained model, and for three values of k: 1, 5 and 10. By comparing these results with the ones reported in the mentioned literature, they share the same trend and in general, they only vary ≈ 0.024, according to the average of the computed differences. To understand if these results are reasonable, given that the resulting data set splits are likely to become very distinct

	Precision@k			Recall@k			NDCG@k		
k	1	5	10	1	5	10	1	5	10
Reported	0.296	0.274	0.271	0.114	0.222	0.228	0.527	0.415	0.393
Obtained	0.279	0.246	0.241	0.105	0.239	0.251	0.529	0.451	0.391
Difference	0.017	0.028	0.030	0.009	0.017	0.023	0.002	0.036	0.002

Table 5.1: Performance comparison of the UserKNN with cosine similarity on the MovieLens-100k data set.

	Precision@k			Recall@k			NDCG@k		
k	1	5	10	1	5	10	1	5	10
Reported	0.377	0.301	0.284	0.149	0.418	0.572	0.623	0.645	0.678
Obtained	0.384	0.341	0.310	0.134	0.447	0.581	0.641	0.674	0.683
Difference	0.007	0.040	0.026	0.015	0.029	0.009	0.018	0.029	0.005

Table 5.2: Performance comparison of the UserKNN with Jaccard index on the MovieLens-100k data set.

	Precision@k			Recall@k			NDCG@k		
k	1	5	10	1	5	10	1	5	10
Reported	0.312	0.298	0.291	0.121	0.220	0.243	0.579	0.453	0.414
Obtained	0.289	0.256	0.242	0.098	0.248	0.262	0.554	0.451	0.397
Difference	0.023	0.042	0.049	0.023	0.028	0.019	0.025	0.002	0.017

Table 5.3: Performance comparison of the UserKNN with mean squared differences on the MovieLens-100k data set.

	Precision@k			Recall@k			NDCG@k		
k	1	5	10	1	5	10	1	5	10
Reported	0.304	0.282	0.279	0.109	0.223	0.236	0.545	0.423	0.394
Obtained	0.312	0.255	0.251	0.111	0.281	0.309	0.558	0.485	0.428
Difference	0.008	0.027	0.028	0.002	0.058	0.073	0.013	0.062	0.034

Table 5.4: Performance comparison of the UserKNN with Pearson correlation on the MovieLens-100k data set.

(they are not using the same seed values), this experiment was also run using the same tool that produced the reported results, the CF4J Java library Ortega et al. [2018], which is also designed for developing RSs. For the sake of brevity, it is only shown the results for the UserKNN recommender with Pearson correlation, which are present in Table 5.5, with an average absolute difference of ≈ 0.0411, slightly larger than the average difference displayed by DRecPy on that recommender of ≈ 0.0338. Even though

we are using the same tool, data set, recommender, splitting, and evaluation procedures, there are still some deviations due to the non-deterministic computation of data set splits offered by CF4J. With that, we can safely say that the average difference shown by DRecPy is acceptable.

	Precision@k			**Recall@k**			**NDCG@k**		
k	1	5	10	1	5	10	1	5	10
Reported	0.304	0.282	0.279	0.109	0.223	0.236	0.545	0.423	0.394
Obtained	0.296	0.256	0.247	0.106	0.286	0.317	0.570	0.505	0.444
Difference	0.008	0.026	0.032	0.003	0.063	0.081	0.025	0.082	0.050

Table 5.5: Performance comparison of UserKNN with Pearson correlation trained on the MovieLens-100k data set, using the CF4J library.

5.1.2 Deep Learning Recommenders

With the purpose of testing the whole system, more tests were developed using a deep learning-based recommender. These tests use the DMF (Deep Matrix Factorization) model described in Xue et al. [2017] and briefly discussed in section 3.2.1, following the evaluation methodology reported on the respective publication.

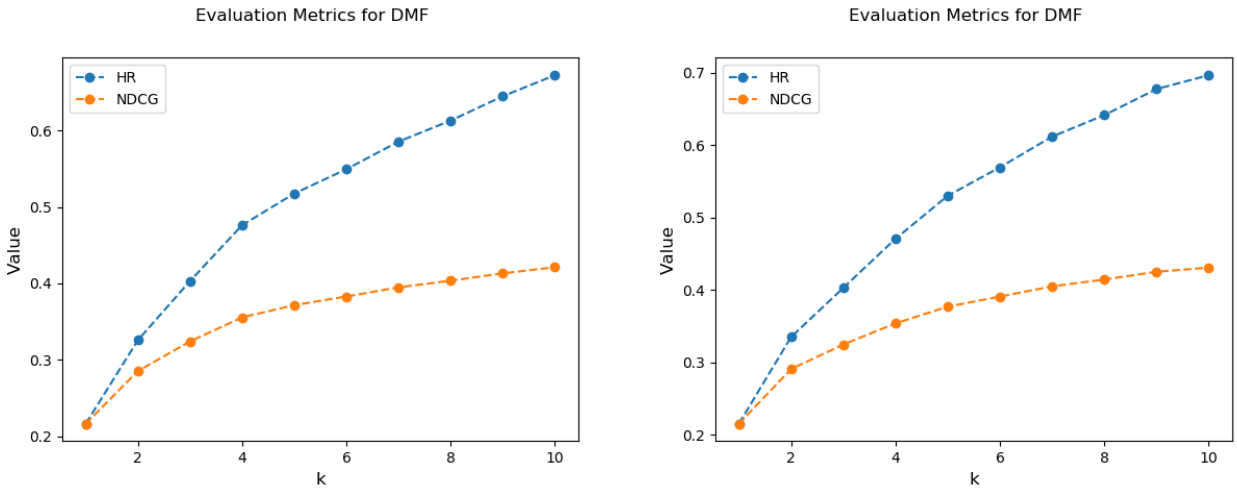
The results for the MovieLens-100k data set are shown in Figure 5.2, and as we can see, the values are very close to the reported results. The DMF-NCE (Normalized Cross-Entropy loss) variant obtained a normalized discounted cumulative gain at 10 value of 0.421 vs. 0.409 (difference of 0.012), and a hit ratio at 10 of 0.672 vs. 0.687 (difference of 0.015). The DMF-CE (Cross-Entropy loss) variant obtained a hit ratio at 10 value of 0.696 vs. 0.679 (difference of 0.017) and a normalized discounted cumulative gain at 10 of 0.430 vs. 0.405 (difference of 0.025).

Slightly better results could be obtained using DRecPy, by increasing the number of training epochs, but since the publication does not mention the number of epochs used for the model training, it's expected that the reported results are not easily reproducible. Another factor that also affects this, is that the sampled items per user for the evaluation process will also be different.

Taking all these results into account, it is clear that DRecPy can be used to reproduce experiments involving RSs, by using its built-in data set structures, split methods, model training procedures, and evaluation processes.

5.2 Extensibility and Reusability Comparison

To compare the extensibility and reusability of DRecPy against another framework built for the same purpose, a test use-case will be analyzed. This test consists on extending an existent recommender on the DRecPy and DeepRec frameworks, specifically the CDAE (Collaborative Denoising Auto-Encoder)



(a) DMF-NCE variant

(b) DMF-CE variant

Figure 5.2: Performance of the DMF models trained on the MovieLens-100k data set.

Wu et al. [2016], by adding a neural network with dimensions specified by the user during initialization, processing the output layer of the original CDAE - let us call this variant the Modified CDAE (MCDAE). This particular recommender was chosen because it is currently the only model in common that both frameworks have implemented. Note that this test is supposed to be an exercise to better understand the benefits given by these tools, and not to generate value through the creation of a novel recommender.

To evaluate the extensibility and reusability of each approach, we show the number of lines of code (LoC) of each implementation (excluding blank lines, import instructions, comments and debug logic), as well as classifications for the amount of reused logic for each common section of work. Each classification can assume a value of: *none* or *few*, when no or almost no code is reused $[0\%, 10\%]$; *partial* when some to half of the code is reused $(10\%, 50\%]$; *most* whenever a very high percentage of code is reused $(50\%, 100\%)$; or *complete* when all code is reused for that section (100%).

The results of this analysis are displayed in Table 5.6, and we can see that in terms of data set split, both frameworks implement these independently of each RS; the network initialization of DRecPy is more granular than DeepRec, since DeepRec relies on that method to initialize network's weights, define signal propagation and the loss function, while DRecPy only initializes and registers trainable variables; the sampling in DeepRec is done in a per-model basis, while in DRecPy it is only required for the model to call one of the sampling methods implemented in the Sampler module; the loss function is shared between the CDAE and MCDAE, so in DRecPy there is no need to re-implement it, while in DeepRec we must manually use the outputs of the new layers; the regularization is required to be re-implemented in DeepRec, due to the high coupling between the predictive and regularization cost functions, while in DRecPy we need to add an extra instruction to compute the regularization of the new layers and add them

to the original CDAE regularization cost; and finally, in DRecPy the evaluation methods are abstracted from model implementation, so all the code is completely reused, while DeepRec requires that each model implements the process to sample negative data points.

Reused Logic	Splitting	Init.	Sampling	Loss	Regul.	Evaluation	LoC
DeepRec CDAE	complete	none	none	none	none	most	92
DRecPy CDAE	complete	partial	most	none	none	complete	62
DeepRec MCDAE	complete	few	complete	none	none	most	30
DRecPy MCDAE	complete	most	complete	complete	partial	complete	19

Table 5.6: Extensibility and reusability comparison of the CDAE and MCDAE models implemented using the DRecPy and DeepRec frameworks.

We can verify that a structured and modular recommender highly improves code reusability, especially if this is done in an environment that provides the right tools to abstract most of the repetitive work.

5.3 Framework Characteristics

To understand what DRecPy brings to the RSs community and how is it any different from the existing toolkits and frameworks, this section lists and goes into detail about important characteristics to have in mind when evaluating the impact of any framework. This will also allow the reader to understand in which situations would be best to use this framework or, in some other cases, other frameworks more suitable for those specific situations and requirements. This section will focus on the comparison between DRecPy and the frameworks that were previously mentioned in this document, namely: DeepRec, Surprise, CF4J, TensorRec, and Spotlight.

As mentioned in Ortega et al. [2018], there are several capabilities that are important to consider for these types of software: extensibility, performance, efficiency, abstraction, flexibility, and scalability. For completion, a new capability will be introduced: replicability.

5.3.1 Extensibility

Extensibility is defined as the ability of a software to evolve from the contribution of third-party developers. These contributions can come in different forms, such as extending existing models, create new ones, implement custom metrics, and other modifications. This is important because if a software is not extensible, then its fixed structure will highly limit its usage - this is not desired for innovative work, which is the most common type of work whenever one's doing research.

DRecPy provides various abstract classes to improve extensibility and make so that new behaviors that implement a common interface are able to be passed as arguments for the builtin methods and pro-

cesses. For instance, the abstract interactions data set allows one to implement data sets that are not stored in memory nor in an auxiliary local database, but instead, they could implement functionalities for manipulating a distributed data set. These new implementations would be completely compatible with the current framework structure, since it doesn't depend on individual implementations, but instead, it depends on a common interface that is well defined from an abstract level. DeepRec allows custom data set loading using utility functions built with the sole purpose of loading data into memory. In order to support out of memory data sets (stored in disk or distributed), apart from the required implementations for that purpose, a huge amount of work would also be needed to make the framework support these new data representations. TensorRec provides utility functions to convert sparse matrices to TensorFlow data sets, but lets the user handle all the parsing work. The Surprise and Spotlight frameworks seem to use a class in order to represent a data set, but these are very limited on what they can do, since they do not allow the manipulation of data sets (filtering, modification, querying...), so implementing custom data set behaviors using these classes, would most likely be impossible. The CF4J framework implements various methods to manipulate a data set using a builtin class, so creating extra data set manipulation methods should be straightforward.

Another example could be extending a given model: DRecPy allows one to extend an existent model and override its functions, so that a new modified model can be easily built, without knowing the model's internal details. As well as DRecPy, Surprise also provides an abstract class for the base recommender model, which every other model should extend and implement. DeepRec and Spotlight, on the other hand, seem to have multiple functions that share the same signature between distinct models, but for some reason, there's no shared interface that would allow an external developer to better understand what's the required methods to implement as well as their responsibilities. TensorRec models are all constructed from a single class, which receives functions to compute user and item representations, as well as loss functions - all these functions also have an abstract class that can be extended to implement user-defined behavior, which makes this small framework to be very powerful and extensible as well. CF4J has some interfaces for models that share common characteristics, a general recommender abstract class for implementing custom models, as well as other abstract classes for implementing new metrics and similarity measures.

5.3.2 Performance

Performance evaluates how effective is a software executing a given task. By leveraging the parallelization power of modern machines, as well as trading some memory usage for reduced computations, we're able to build performant applications that are faster than if we don't take advantage of these options.

DRecPy tries to use task parallelization whenever possible, specifically in processes such as data set splitting and model evaluation, with configurable parameters to limit the maximum number of concurrent threads, which should be defined according to the machine in which they are being run. It also relies on TensorFlow to allow model training on the GPU (Graphics Processing Unit). Although there is a big

effort to reach high execution speeds, DRecPy still does not show the expected performance values due to how the call flow is arranged (many redundant calls to avoid unexpected situations). Also, increasing memory usage would show great performance improvements, but this hasn't been applied for now, since a robust structure was preferred over performance[1]. Spotlight does not use threads as DRecPy, but also allows to run its models on the GPU, by leveraging this capability from PyTorch. CF4J also uses threads to execute almost all tasks in parallel, which makes it very fast at executing a recommendation pipeline (this is also true because CF4J is a Java framework, which becomes faster because Java is a compiled language, whilst Python is an interpreted language). Conversely, DeepRec, Surprise, and TensorRec do not use any technique to speed up their processes.

5.3.3 Efficiency

Sometimes ensuring that the software minimizes as much as possible the impact it has on the memory usage of the machine where it's running, might be important. An efficient software minimizes the memory usage as much as possible to execute a given task, possibly taking longer to execute. In the context of RSs, there are multiple ways of storing user interactions: some more efficient, such as always read from disk or database whenever needed, and others less efficient, like storing a map from user ids to their interactions in memory.

The memory consumed by DRecPy highly depends on the type of interaction data set used: if the in-memory structure is used, then each data point is loaded into memory only once (and when internal ids are assigned, and if there's available memory, an optimization process is triggered and a new structure is built that maps each user and item to their respective points - doubling the memory consumption); however, if the out of memory structure is used, then all the data will be stored in disk to an SQLite database and all manipulations are done via queries and updates to this database. This allows the user to have more control over whether he/she wants to make more use of the available machine memory to speed up the data manipulations, or not. CF4J always constructs 2 arrays: one for storing user information and another for the items, where each array index corresponds to the user/item internal identifiers, so the interactions are always duplicated. Surprise also duplicates the rating information, by creating a map from user internal ids to their corresponding interactions, and another map for the items. DeepRec reads the data in the form of sparse matrices, which in terms of memory usage is very similar to loading each data point into memory once. As mentioned above, TensorRec uses the TensorFlow data set to do model training, so all data is also loaded into memory and no other structures are built. The Spotlight framework uses a list for each column of the data set: a list for storing user ids, another list for storing item ids, a new optional list for ratings, etc. - this is basically the same as storing a list with all the rows of the data set.

[1] Although some other possibilities to improve DRecPy's performance will be explored in future works.

5.3.4 Abstraction

Software abstractions are useful when they correctly model their target concept. For the abstractions provided by a recommendation framework to be useful, their representation should be intuitive and should behave like expected, such as recommender models having the common interfaces (fit, predict), data set structures that are easy to manipulate (no need to interact with internal identifiers), etc. Complex abstractions might be required to allow the user to make as much use of them as necessary.

Abstractions are widely used in DRecPy, having most of its features implemented via extension of abstract classes. Recommender models always follow the simple create-fit-predict pattern, and since we're dealing with models that target recommendation problems, they also provide functions to recommend and rank a set of items for a given user (as the developer should expect when manipulating any recommender, but oddly this was not found on any other explored framework, except TensorRec which also implements a rank function). DeepRec on the other hand, provides but does not allow the usual create-fit-predict pattern call (because of implicit and unexpected behavior, such as the need to call data preparation functions between the create and train functions or the need to build the model's neural network structure), forcing the developer to be aware of issues that should be internal concerns. Model training on the Surprise, Spotlight and TensorRec frameworks is also intuitive, following the user create-fit-predict pattern. CF4J also provides an intuitive model training, allowing one to do single or multiple predictions by passing the whole test set as a parameter.

Data manipulation on DRecPy is also intuitive, allowing the developer to select/filter all rows for a single user with a simple function call. Another negative point about DeepRec is that there's no builtin data set structure, so accessing all the rows for a given user is not intuitive, because it requires to manipulate sparse matrices. Surprise has some useful functions provided by the builtin data set module, such as verifying whether a user or item is present on the data set, but it's still limited and not intuitive, as it also forces the developer to pass identifiers as strings instead of their representation on the data set file. Manipulating Spotlight's internal data set is not easy (need to iterate multiple lists), and it also does not expose a simple and direct way to gather all interactions for a single user. On the other hand, TensorRec lacks an internal data set representation, so it's hard to think about it in an abstract way. CF4J provides an intuitive mechanism to access and iterate through the interactions of a user, by exposing functions on the data set class to do so, although it does not allow selection instructors.

DRecPy still has some progress to be done so that abstraction is improved across all the framework methods, specifically on evaluation processes and value inference. In order to offer features that work without the need to fine-tune its parameters, a lot of default values are required, which have some positive notes (e.g. faster to do experiments, less code) but also causes some issues when the user is not fully aware of those default behaviors. Good documentation does not seem to be enough for all cases, therefore in the future, all default behaviors should be analyzed, and some removed completely, forcing explicit user input.

5.3.5 Flexibility

Flexibility is the capability of a software to fit new and/or unexpected scenarios. Some examples of unexpected scenarios include accessing internal properties (which usually are not exposed) and implementing distinct processing pipelines. For enterprise applications, where the use cases are all defined from the start, the degree of flexibility should try to be limited as much as possible to avoid unexpected behaviors. But for research and exploratory projects, one would prefer the software to be highly flexible.

Python frameworks (all except CF4J) show a clear advantage here because Python attributes are accessible to all code (private attributes on Python are just a convention and are not forced during execution), therefore accessing intermediate values such as model weights is simple. CF4J tries to do its best to provide public functions that expose these private attributes, but without success, since many classes (e.g. NMF model) do not provide public methods to access user and item factors.

DRecPy also allows developers to process their data sets before executing model training, by using functions directly implemented on the interactions data set structure. Every other framework also supports modified data sets, but they should be modified before being imported using the custom data structures or functions. This introduces an unnecessary change to the processing pipeline, since instead of importing, modifying, and then training using the data, we need to first load the data into memory, modify it, import it to structures that the framework supports, and finally train the model.

5.3.6 Scalability

A scalable software is able to accommodate a processing growth to infinity, without severely impacting its performance. This is commonly referred to as software that is able to scale not only vertically (by adding extra resources to the existing machines), but also horizontally (by connecting multiple machines and combine their processing power to execute a common work). Since during the performance capability discussion we've already analyzed which frameworks use threads and GPUs to increase computation speed (which is highly related to vertical scaling), we'll only discuss about horizontal scaling during this section.

No framework mentioned here allows for horizontal scalability, apart from DRecPy (on the data set manipulation only). Although this is not implemented yet, it is already possible to implement a new interaction data set type by extending the base abstract class, that communicates with a Spark cluster to handle data manipulation tasks. This would obviously allow trained models to use massive data sets in a scalable manner, but it still might reach a point where the model itself would require to be distributed (i.e. distributed model training), and unfortunately, this is not supported by DRecPy either.

5.3.7 Replicability

Replicable software provides functionality to execute deterministic processes whenever required. This is especially important on exploratory projects, such as the ones required whenever training and evaluating

RSs.

DRecPy was built with replicability as a first priority, so all methods that depend on stochastic processes (even those present in thread-based computations) have an optional seed parameter that should be set whenever we want to be able to replicate the results in the future. DeepRec doesn't seem to expose a clear way to set the seed of pseudo-random number generators, so it is hard to ensure the replicability of an experiment. Similarly to DRecPy, Surprise, CF4J and Spotlight also support a seed parameter for all functions that depend on non-deterministic behavior, such as data set splitting processes and weight initialization. TensorRec does not provide any direct way to set seeds for random processes, but since it uses TensorFlow to generate all random values, we can use the `set_seed` function to make all these processes deterministic.

Chapter 6

Conclusion

In a world where the available options exceed the individual capacity to assimilate and compare distinct options, RSs are an essential tool to aid these individuals. It not only allows the individuals to make better choices, but also helps the platforms to be personalized and tailored to each person. Overall, this improves the user satisfaction level, and increases the number of transactions made via each platform. To create and experiment distinct RSs approaches, it is important to have tools that allow developers and researchers to do so in an environment where most requirements to start these developments are met. This not only speeds up the development and experiment processes, reduces costs, and improves consistency of obtained results and comparisons.

This dissertation describes various topics related with these systems, as well as successfully introducing a novel open-source framework called DRecPy, that is designed to help develop recommenders that are modular, extensible, and reproducible, with special emphasis on DL-based recommenders. It is shown how DRecPy is organized, how it helps tackle each challenge related with the development of RSs, and how it innovates from previous frameworks. The main differentiators of this framework can be summed up as follows:

1. Full abstraction over data set conversion, provided by the Dataset module, with a built-in import procedure, automatic id conversion and out-of-memory data sets;

2. Deterministic training and evaluation pipelines enabled by default, through the Sampler, Recommender and Evaluation modules, by using seeded pseudo-random number generators;

3. Structure for developing deep learning-based RSs, with an extensible and flexible model structure, offered by the Recommender module, through the modular call workflow where each step is replaceable;

4. Built-in support for epoch callbacks, which can be used by early stopping rules, allowing a better overview of the performance of a RSs and also working as a strategy to avoid overfitting.

Apart from these key differentiators, DRecPy also offers a rich suite of tools to split data sets, with various splitting functions, sample data points, together with the required evaluation procedures for predictive and learn-to-rank RSs.

This work also produced a scientific article in the RecSys'20 conference, containing a brief description of the architecture of the same framework.

6.1 Future Work

With DRecPy being a framework specifically tackled for deep learning-based RSs, future work must include the incorporation of more built-in state-of-the-art RSs of this type into the framework, as well as building a complete comparison between every recommender. More future work includes:

- Non-accuracy related metrics, such as diversity, novelty, and serendipity;
- Support for distributed interaction data sets, so that data sets that do not fit in memory or disk, can be seamlessly supported on every DRecPy pipeline;
- Support for fine-tuning trained RSs, allowing users to make use of complex models that are already trained, and tune them according to their data set;
- Improve the performance of the built-in interactions data set module, to boost the performance of every DRecPy process.

References

Abadi, M., Barham, P., Chen, J., Chen, Z., Davis, A., Dean, J., Devin, M., Ghemawat, S., Irving, G., Isard, M., et al. (2016). Tensorflow: A system for large-scale machine learning. In *12th {USENIX} symposium on operating systems design and implementation ({OSDI} 16)*, pages 265–283. 54

Adomavicius, G. and Tuzhilin, A. (2005). Toward the next generation of recommender systems: A survey of the state-of-the-art and possible extensions. *IEEE transactions on knowledge and data engineering*, 17(6):734–749. 13

Aggarwal, C. C. (2016). Content-based recommender systems. In *Recommender Systems*, pages 139–166. Springer. 9

Barros, M., Moitinho, A., and Couto, F. M. (2019). Using research literature to generate datasets of implicit feedback for recommending scientific items. *IEEE Access*, 7:176668–176680. 61, 62

Batmaz, Z., Yurekli, A., Bilge, A., and Kaleli, C. (2019). A review on deep learning for recommender systems: challenges and remedies. *Artificial Intelligence Review*, 52(1):1–37. 2

Beel, J., Gipp, B., Langer, S., and Breitinger, C. (2016). Research-paper recommender systems: a literature survey. *International Journal on Digital Libraries*, 17(4):305–338. 3

Bennett, J., Lanning, S., et al. (2007). The netflix prize. In *Proceedings of KDD cup and workshop*, volume 2007, page 35. New York. 18

Breese, J. S., Heckerman, D., and Kadie, C. (2013). Empirical analysis of predictive algorithms for collaborative filtering. *arXiv preprint arXiv:1301.7363*. 13

Brynjolfsson, E., Hu, Y. J., and Smith, M. D. (2006). From niches to riches: Anatomy of the long tail. *Sloan Management Review*, 47(4):67–71. 2

Burke, R. (2000). Knowledge-based recommender systems. *Encyclopedia of library and information systems*, 69(Supplement 32):175–186. 11

Burke, R. (2002). Hybrid recommender systems: Survey and experiments. *User modeling and user-adapted interaction*, 12(4):331–370. 12

Chen, C., Seff, A., Kornhauser, A., and Xiao, J. (2015). Deepdriving: Learning affordance for direct perception in autonomous driving. In *Proceedings of the IEEE International Conference on Computer Vision*, pages 2722–2730. 28

Colaço, F., Barros, M., and Couto, F. M. (2020). Drecpy: A python framework for developing deep learning-based recommenders. In *Fourteenth ACM Conference on Recommender Systems*, pages 675–680. 4

Craswell, N. and Robertson, S. (2009). *Average Precision at n*, pages 193–194. Springer US, Boston, MA. 25

Cybenko, G. (1989). Approximation by superpositions of a sigmoidal function. *Mathematics of control, signals and systems*, 2(4):303–314. 29

Davidson, J., Liebald, B., Liu, J., Nandy, P., Van Vleet, T., Gargi, U., Gupta, S., He, Y., Lambert, M., Livingston, B., et al. (2010). The youtube video recommendation system. In *Proceedings of the fourth ACM conference on Recommender systems*, pages 293–296. ACM. 1

Donkers, T., Loepp, B., and Ziegler, J. (2017). Sequential user-based recurrent neural network recommendations. In *Proceedings of the Eleventh ACM Conference on Recommender Systems*, pages 152–160. ACM. 47

Garcin, F., Faltings, B., Donatsch, O., Alazzawi, A., Bruttin, C., and Huber, A. (2014). Offline and online evaluation of news recommender systems at swissinfo. ch. In *Proceedings of the 8th ACM Conference on Recommender systems*, pages 169–176. 21

Goldberg, K., Roeder, T., Gupta, D., and Perkins, C. (2001). Eigentaste: A constant time collaborative filtering algorithm. *information retrieval*, 4(2):133–151. 17

Gomez-Uribe, C. A. and Hunt, N. (2016). The netflix recommender system: Algorithms, business value, and innovation. *ACM Transactions on Management Information Systems (TMIS)*, 6(4):13. 1

Hanin, B. and Sellke, M. (2017). Approximating continuous functions by relu nets of minimal width. 30

Harper, F. M. and Konstan, J. A. (2016). The movielens datasets: History and context. *Acm transactions on interactive intelligent systems (tiis)*, 5(4):19. 16

He, K., Zhang, X., Ren, S., and Sun, J. (2016). Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 770–778. 28

He, X., Liao, L., Zhang, H., Nie, L., Hu, X., and Chua, T.-S. (2017). Neural collaborative filtering. In *Proceedings of the 26th international conference on world wide web*, pages 173–182. 34, 59

Hidasi, B., Karatzoglou, A., Baltrunas, L., and Tikk, D. (2015). Session-based recommendations with recurrent neural networks. *arXiv preprint arXiv:1511.06939*. 47

Hinton, G. E. and Salakhutdinov, R. R. (2006). Reducing the dimensionality of data with neural networks. *science*, 313(5786):504–507. 34

Hug, N. (2017). Surprise, a Python library for recommender systems. `http://surpriselib.com`. 36

Jawaheer, G., Szomszor, M., and Kostkova, P. (2010). Comparison of implicit and explicit feedback from an online music recommendation service. In *proceedings of the 1st international workshop on information heterogeneity and fusion in recommender systems*, pages 47–51. 15

Jin, R. and Si, L. (2004). A study of methods for normalizing user ratings in collaborative filtering. In *Proceedings of the 27th annual international ACM SIGIR conference on Research and development in information retrieval*, pages 568–569. ACM. 15

Karlik, B. and Olgac, A. V. (2011). Performance analysis of various activation functions in generalized mlp architectures of neural networks. *International Journal of Artificial Intelligence and Expert Systems*, 1(4):111–122. 32

Kula, M. (2017). Spotlight. `https://github.com/maciejkula/spotlight`. 37

LeCun, Y., Bottou, L., Bengio, Y., and Haffner, P. (1998). Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, 86(11):2278–2324. 31

Lu, Z., Pu, H., Wang, F., Hu, Z., and Wang, L. (2017). The expressive power of neural networks: A view from the width. In Guyon, I., Luxburg, U. V., Bengio, S., Wallach, H., Fergus, R., Vishwanathan, S., and Garnett, R., editors, *Advances in Neural Information Processing Systems 30*, pages 6231–6239. Curran Associates, Inc. 30

Ma, Y. and Narayanaswamy, M. B. (2018). Hierarchical temporal-contextual recommenders. 12

Manouselis, N., Drachsler, H., Vuorikari, R., Hummel, H., and Koper, R. (2011). Recommender systems in technology enhanced learning. In *Recommender systems handbook*, pages 387–415. Springer. 1

Martinez, L., Barranco, M. J., Pérez, L. G., and Espinilla, M. (2008). A knowledge based recommender system with multigranular linguistic information. *International Journal of Computational Intelligence Systems*, 1(3):225–236. 8

McAuley, J., Targett, C., Shi, Q., and Van Den Hengel, A. (2015). Image-based recommendations on styles and substitutes. In *Proceedings of the 38th International ACM SIGIR Conference on Research and Development in Information Retrieval*, pages 43–52. 34

Mitchell, T. (1997). *Machine Learning*. McGraw-Hill. 26

Ortega, F., Zhu, B., Bobadilla, J., and Hernando, A. (2018). Cf4j: Collaborative filtering for java. *Knowledge-Based Systems*, 152:94–99. 37, 63, 66

Pal, S. K. and Mitra, S. (1992). Multilayer perceptron, fuzzy sets, classifiaction. 30

Paterek, A. (2007). Improving regularized singular value decomposition for collaborative filtering. In *Proceedings of KDD cup and workshop*, volume 2007, pages 5–8. 11

Prechelt, L. (1998). Early stopping-but when? In *Neural Networks: Tricks of the trade*, pages 55–69. Springer. 33

Radford, A., Wu, J., Child, R., Luan, D., Amodei, D., and Sutskever, I. (2019). Language models are unsupervised multitask learners. *OpenAI Blog*, 1(8). 28

Rendle, S., Freudenthaler, C., Gantner, Z., and Schmidt-Thieme, L. (2012). Bpr: Bayesian personalized ranking from implicit feedback. *arXiv preprint arXiv:1205.2618*. 59

Safoury, L. and Salah, A. (2013). Exploiting user demographic attributes for solving cold-start problem in recommender system. *Lecture Notes on Software Engineering*, 1(3):303–307. 9

Schafer, J. B., Konstan, J., and Riedl, J. (1999). Recommender systems in e-commerce. In *Proceedings of the 1st ACM conference on Electronic commerce*, pages 158–166. ACM. 1

Shuai Zhang, Yi Tay, L. Y. B. W. A. S. (2019). Deeprec: An open-source toolkit for deep learning based recommendation. 35

Silver, D., Schrittwieser, J., Simonyan, K., Antonoglou, I., Huang, A., Guez, A., Hubert, T., Baker, L., Lai, M., Bolton, A., et al. (2017). Mastering the game of go without human knowledge. *Nature*, 550(7676):354. 28

Tang, J. and Wang, K. (2018). Personalized top-n sequential recommendation via convolutional sequence embedding. In *Proceedings of the Eleventh ACM International Conference on Web Search and Data Mining*, pages 565–573. 20, 41, 42

Wan, M. and McAuley, J. J. (2018). Item recommendation on monotonic behavior chains. In *Proceedings of the 12th ACM Conference on Recommender Systems, RecSys 2018, Vancouver, BC, Canada, October 2-7, 2018*, pages 86–94. 18

Wilson, D. R. and Martinez, T. R. (2003). The general inefficiency of batch training for gradient descent learning. *Neural networks*, 16(10):1429–1451. 34

Wu, Y., DuBois, C., Zheng, A. X., and Ester, M. (2016). Collaborative denoising auto-encoders for top-n recommender systems. In *Proceedings of the Ninth ACM International Conference on Web Search and Data Mining*, pages 153–162. 40, 41, 65

Xue, H.-J., Dai, X., Zhang, J., Huang, S., and Chen, J. (2017). Deep matrix factorization models for recommender systems. In *IJCAI*, pages 3203–3209. 38, 39, 55, 64

Zhao, X., Zhang, L., Ding, Z., Xia, L., Tang, J., and Yin, D. (2018). Recommendations with negative feedback via pairwise deep reinforcement learning. In *Proceedings of the 24th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*, pages 1040–1048. 28

Ziegler, C.-N., McNee, S. M., Konstan, J. A., and Lausen, G. (2005). Improving recommendation lists through topic diversification. In *Proceedings of the 14th international conference on World Wide Web*, pages 22–32. ACM. 18

Zuo, Y., Zeng, J., Gong, M., and Jiao, L. (2016). Tag-aware recommender systems based on deep neural networks. *Neurocomputing*, 204:51–60. 47