FACULDADE DE ENGENHARIA DA UNIVERSIDADE DO PORTO

Software library for stream-based recommender systems

Fernando André Bezerra Moura Fernandes

Mestrado Integrado em Engenharia Informática e Computação

Supervisor: Prof. João Nuno Vinagre Marques da Silva

Cosupervisor: Prof. Luís Filipe Pinto de Almeida Teixeira

August 8, 2020

Software library for stream-based recommender systems

Fernando André Bezerra Moura Fernandes

Mestrado Integrado em Engenharia Informática e Computação

August 8, 2020

Abstract

Traditionally, a machine learning algorithm is able to learn from data, given a previously built and treated data set. One can also analyze that data set, using data mining techniques, and draw conclusions from it. Both of these concepts have numerous world-wide applications, from medical diagnosis to speech recognition or even search engine queries. However, it is assumed that the dataset is stationary. This is not necessarily true with modern data, as many of today's applications are required to receive and process large number of observations from incoming data streams within strict time frames. Therefore, there is a need for techniques to mine and process these data streams, on a limited time period with high accuracy and effective scaling as data grows. One specific use case of analyzing and predicting future conclusions from given data, are recommendation systems. Several online services use recommender systems to deliver personalized content to their users. In many cases, recommendations are one of the most effective traffic generators in such services. The problem lies in finding the best small subset of items in a system that matches the personal preferences of each user, through the analysis of a very large amount of historical data. One can distinguish solutions between perfect ones or statistically similar ones. Due to the large amount of data available, the decision to reprocess all the data every time new data arrives, would not be feasible so, incremental algorithms are used to process incoming data and keeping the recommendation model updated. This problem gets more attention due to the fact that there is no software library which facilitates the investigation and development as well as evaluation mechanisms on this field of work. Current options are micro-libraries which are task-specific and make them hard to extend. This work presents a library that implements incremental approaches to recommendation and mechanisms to evaluate them, including the visualization of results. This dissertation provides implementation details, along with a demonstration with real world data and a discussion of results.

Resumo

Tradicionalmente, um algoritmo de machine learning é capaz de aprender com dados, considerando um e conjunto de dados tratado. Também é possível analisar esse conjunto de dados, usando técnicas de data mining, e tirar conclusões sobre ele. Ambos os conceitos têm inúmeras aplicações em todo o mundo, desde diagnóstico médico, a reconhecimento de fala ou mesmo consultas a motores de busca. No entanto, assume-se que o conjunto de dados é estacionário. Isso não é necessariamente verdade com os dados modernos, pois as aplicações modernas recebem e processam milhões de novas instâncias em uma fração de tempo limitada. Desta forma, são necessárias técnicas para extrair e processar esses fluxos de dados, em um período de tempo limitado, com boa precisão e dimensionamento eficaz à medida que os dados aumentam. Um caso de uso específico de analisar e prever conclusões futuras com dados fornecidos são os sistemas de recomendação. Vários serviços on-line usam sistemas de recomendação para fornecer conteúdo personalizado para seus utilizadores. Em muitos casos, as recomendações são um dos meios geradores de tráfego mais eficazes nestes serviços. O problema está em encontrar o melhor pequeno subconjunto de itens em um sistema que corresponda ao pessoal preferências de cada usuário, através da análise de uma quantidade muito grande de dados históricos. Podem-se distinguir entre soluções perfeitas ou estatisticamente semelhantes. Devido à grande quantidade de dados disponíveis, a decisão de reprocessar todos os dados sempre que novos dados chegassem, não seria viável, portanto, algoritmos incrementais são usados para processar dados recebidos e mantendo o modelo de recomendação atualizado. Esse problema chama mais atenção devido ao fato de não haver nenhuma biblioteca de software que facilite a investigação e desenvolvimento e que possua mecanismos de avaliação neste campo de trabalho. As opções atuais são micro-bibliotecas demasiado específicas talhadas para uma tarefa específica aquando da sua construção o que as torna difícil de estender. Este trabalho apresenta uma biblioteca que implementa abordagens incrementais para recomendação e mecanismos para as avaliar, incluindo a visualização de resultados. Esta dissertação fornece detalhes de implementação, juntamente com demonstrações com dados do mundo real e discussão dos resultados.

Acknowledgments

I would like to thank my father, mother and brother for being my role models in my personal growth and supporting my academic formation without question since i was born.
To all my friends and girlfriend for the long-run so far.
My gratitude to the great help of Prof. João Vinagre and Prof. Luís Teixeira.

Fernando André Bezerra Moura Fernandes

"We can only see a short distance ahead,
but we can see plenty there that needs to be done."

Alan Turing

Contents

List of Figures

List of Equations

Abbreviations and Symbols

UB - User Based
IB - Item Based
CF - Collaborative Filtering
MB - Model Based
MF - Matrix Factorization
TF - Term Frequency
IDF - Inverse Document Frequency
CB - Content Based
SGD - Stochastic Gradient Descent
ALS - Alternate Least Squares
LSH - Locality-sensitive-hashing
SW - Sliding window
IMF - Implicit matrix factorization
EMF - Explicit matrix factorization
EMFP - Explicit matrix factorization with matrix preprocessing
LF - Latent factors
LR - Learning rate
REG - Regularization factor
IIBN - Implicit item-based neighborhood
IILSMH - Implicit item-based locality-sensitive min-hashing
PERMS - Number of permutations
IUBN - Implicit user-based neighborhood
IUBC - Implicit user-based clustering
IULSMH - Implicit user-based locality-sensitive min-hashing
EUBN - Explicit user-based neighborhood
EUBC - Explicit user-based clustering

Chapter 1

Introduction

1.1 Context

The increasing usage of the Web by the everyday user on the numerous web applications scattered around the globe provides a tremendous amount of information to the companies who own these web applications. This information is collected through the **feedback** provided by users, in the form of **ratings**, by their actions. For example, for a company like *Amazon.com* feedback could consist of a product purchase or product page click. Web businesses provide a huge quantity of content for users to watch, read, click or buy, to the point where there is so much content available, that the regular user has some trouble deciding what to do next: *"What film do I watch?"*, *"What game do I buy?"*, *"Which food do I order?"*. This is where **Recommender Systems** come into action. They are capable of inferring user preferences and then make personalized recommendations based on those preferences. They do so, by analyzing *user-item* interactions, processing them and then returning results based on those interactions. These recommender systems are extremely valuable to companies because they provide an objective way of boosting sales with their acquired data. In 2006, Netflix released a large movie rating dataset and proceeded to challenge data mining, machine learning and computer science experts to build their own recommender system and gave a set of prizes to the ones who performed better than the Netflix one, by certain amounts [9].

1.2 Motivation

Traditional recommender systems work the same way as a **Machine Learning** model, on the way they use data. They iteratively consume a **dataset** and form a recommendation model used to provide recommendations for users. However, due to an increase in internet trafficking this is becoming more and more unsustainable. For companies such as *Google*, *Spotify* or *Facebook*, **data streams** come every second with website clicks, song requests or page likes which will increase the size of the dataset. If the dataset had to be reprocessed every time a new data stream arrived, it would be impossible to provide in real-time updated recommendations because there would not be enough computer power or time to do it. We can clearly see that it is not a scalable solution.

Some companies choose to do these expensive computations offline, and periodically update their recommendation model, keeping in real-time recommendations, but this scheme does not allow to keep recommendations accurate and updated since a lot of information is transferred between updates which is not being used [27]. Current incremental approaches are not vast, neither they are diversified and the ones that exist are task-specific. So, there must be another way of creating highly scalable recommender systems capable of dealing with incoming data streams keeping in real-time and updated recommendations of items to users. An effective solution for this problem is the adoption of **incremental learning approaches**. Each time a new data stream comes to the system, it can be observed as a rating and the new rating will be a function of the previously calculated ratings and the newest one. This way the model does not have to be recalculated again since it was calculated before. Thus, it would be useful for a **library** which aided the development of incremental recommender systems for highly scalable solutions to exist, and be imported by a third party as a project dependency. This would allow the development of new algorithms, the comparison between them, and since they should be built in a generic way, this would allow their appliance in real-world applications.

1.3 Objectives

The main goal of this dissertation is to build a scalable cross-platform **library** which contains a collection of algorithms who can form recommendations based on an incoming data stream. Therefore, the library:

- Must be able to efficiently receive data streams and **incrementally** update the recommendation model.
- Must have a collection of different incremental algorithms that make different types of recommendations which are applicable in different enterprise scenarios.
- Must be **extensible**.
- Must be built in a **generic** way so that it can be easily used for different application domains.
- Must be **lightweight**, not having too many dependencies.
- Must have mechanisms to measure algorithm **efficiency** and **accuracy**.
- Must be **tested**.
- Must be **open-sourced**.

1.4 Document Structure

The rest of this document will explain in detail theory around recommender systems, the newest incremental trend and the planned solution. Firstly we will divide recommender systems into **Collaborative Filtering** models and **Content-based Models** and explain the differences. Then, we will discuss the **Incremental Recommendation** algorithms which is not yet a very thoroughly researched field. After this, a solution to this problem will be explained with the implemented library **Increc**. With regard to the implementation, there is also a chapter discussing the results obtained of the implemented approaches. Then, there is a final chapter for conclusions and limitations of the proposed work.

Chapter 2

Traditional Recommendation Systems

Recommender systems are a type of machine learning business application which provides suggestions of **items** to **users** [30]. We understand *items* as products of a business, books for an online bookshop, music on *Spotify* or videos on *Youtube*. *Users*, naturally, are these platforms customers who generate traffic with their actions and intents. Usually, a recommender system has two types of tasks:

1. Predict a rating of a user to a certain item
2. Find out the *top-k* items for a targeted user (or *top-k* users for a designated) item.

These tasks are gravely affected by the **Long Tail Problem**.

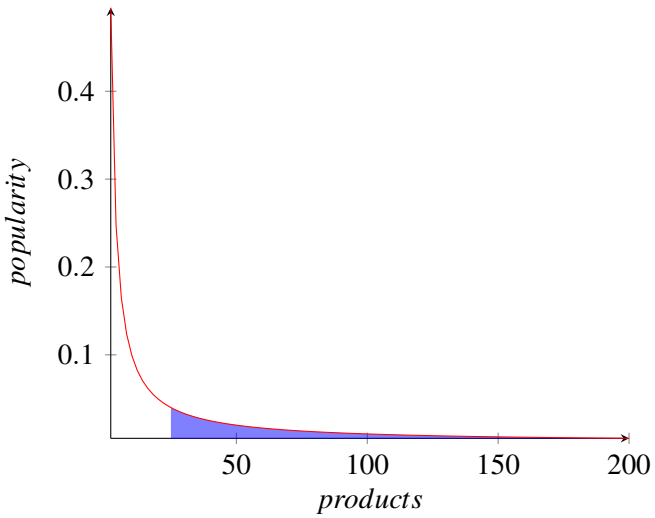

Figure 2.1: The long tail problem.

Statistically speaking, the *tail* of a distribution of values is the portion with very large or very small values compared to the rest. A distribution is called **long-tailed** if it contains values which are far from the main body of the data [43]. Whilst physical stores can offer thousands of products on shelves to a customer, online retailers have millions of them in their databases. A physical

newspaper can print several dozen articles per day, while online news services offer thousands. The offer is much bigger on online platforms. By looking at figure 2.1 the vertical axis represents popularity and on the horizontal axis there are placed items which are ranked according to their popularity. Physical stores provide the items on the uncolored area, and online stores provide all the range of items on the horizontal axis, including the items in the blue area. Product catalogs can be extremely large, however, only a small percentage of the products is responsible for traffic generation. Recommender systems are capable of recommending items from the long-tail to specific users. It is not possible to recommend all items to individual users neither enterprises can expect users to have heard of each of the items they might like [19]. Furthermore, it is also not a good option to keep recommending hyped items to the users since it will pose no interest to them if we do it repeatedly. There are two essential types of recommendation models:

1. **Collaborative Filtering Systems**
2. **Content-Based Systems**

These two types of systems will be explained in the next sections.

2.1 Collaborative Filtering Systems

Collaborative Filtering is a type of recommendation technique which attempts to match similar users or similar items by looking at users' feedback and then making recommendations based on these matches [37]. They use different forms of feedback provided by the users. We can distinguish these forms as two types of feedback:

1. **Explicit** feedback
2. **Implicit** feedback

Explicit feedback relates to directly provided information by the users and a good example of this are movie ratings on an evaluation system. On *IMDb*, registered users can rate movies on a *1-10 star* rating system and on the *Google Play Store* the same system is applied on a 1-5 scale for the available apps. *Implicit* feedback consists of a set of user actions through which the system generates conclusions. If a user clicks on several gaming mouse pads on a website, or even buys one, it means he is interested on this type of product. These types of systems attempt to match similar items by viewing closely view/rated items by users. They can also do the same for users, matching them by tracking and calculating users which saw the same items. When analyzing collaborative filtering, one must discuss how data is modeled in these types of systems. These systems use a *user-item* **ratings matrix**, where each **row** represents a user, each **column** represents an item, and the cells are filled with the rating of a user to a certain item.
The purpose of collaborative filtering is predicting the unknown ratings on the *user-item* matrix because the observed ratings are highly correlated among users and items [6]. They do this by identifying patterns in the observed interactions between users and items.

$$\begin{bmatrix} & item_1 & item_2 & item_3 & ... & item_k \\ user_1 & 1 & ? & ? & ... & rating_{1k} \\ user_2 & 3 & ? & 3 & ... & rating_{1k} \\ user_3 & ? & ? & ? & ... & rating_{1k} \\ ... & ... & ... & ... & ... & ... \\ user_n & rating_{n1} & rating_{n2} & rating_{n3} & ... & rating_{nk} \end{bmatrix}$$

Figure 2.2: The user/item matrix.

There is one problem with this matrix, which is its **sparsity**. Notice that on 2.2 most of the cells are not filled with any value(marked with *"?"*). This happens because there are no ratings by most users to most items. When someone has a *Netflix* account, he/she only watches a limited amount of movies or TV series. When people go on *Medium*, they only click a limited amount of links of articles to read them. When people shop on *Amazon.com*, they only buy certain items or browse for related ones, not the entire catalogue. This way, in *collaborative filtering* we deal with very sparse vectors of users and items, which in turn have high dimensionality. This results in problems related to machine learning models reliability and accuracy [14].
Another main concern with *Collaborative Filtering* algorithms has to do with **Cold-Start**. This problem occurs when it's not possible to generate accurate recommendations because of an initial lack of ratings [32]. Let's imagine a recent user has entered a system. Since we have no previous information regarding his/her feedback, there is no feasible way of matching him up with other users. The same principle applies to recent items, since no user got the opportunity to interact with it in the first place, which makes it difficult to get matched with other items. This has a great impact when measuring model accuracies in this field since is is very difficult to generate recommendations for new users or include new items in these recommendations.
The next sections will describe two different types of collaborative filtering models:

1. **Neighborhood-Based**
2. **Model-Based**

2.1.1 Neighborhood-Based Collaborative Filtering

Neighborhood-based approaches to collaborative filtering build a machine learning model consisting of neighbors for each user or item in the system. They are also known as **Memory-Based**. The recommendations are generated for a user depending on his/hers neighbors. Neighborhood-based methods can be viewed as generalizations of nearest neighbor classifiers in the machine learning literature [6].
Using a *user-item* matrix of m users and n items, where r_{ik} represents the rating of the *ith* user to the *kth* item, each row or column can be represented as a vector:

- The *ith* user u can be represented as $u_i = \{r_{i1}, r_{i2}, r_{i3}, ..., r_{in}\}$ and $||u_i|| = n$
- The *kth* item i can be represented as $i_k = \{r_{1k}, r_{2k}, r_{3k}, ..., r_{mk}\}$ and $||i_k|| = m$

Neighbors are calculated using previously computed similarities between pairs of users and items. If we represent items and users as previously mentioned we gain new possibilities to infer their similarities. One of these possibilities is using the well known **Jaccard similarity** or **Jaccard Distance**, represented in figure 2.3 which is defined as the size of the intersection divided by the size of the union of the sample sets [19].

$$JaccardSimilarity(A,B) = \frac{|A \cap B|}{|A \cup B|} \tag{2.1}$$

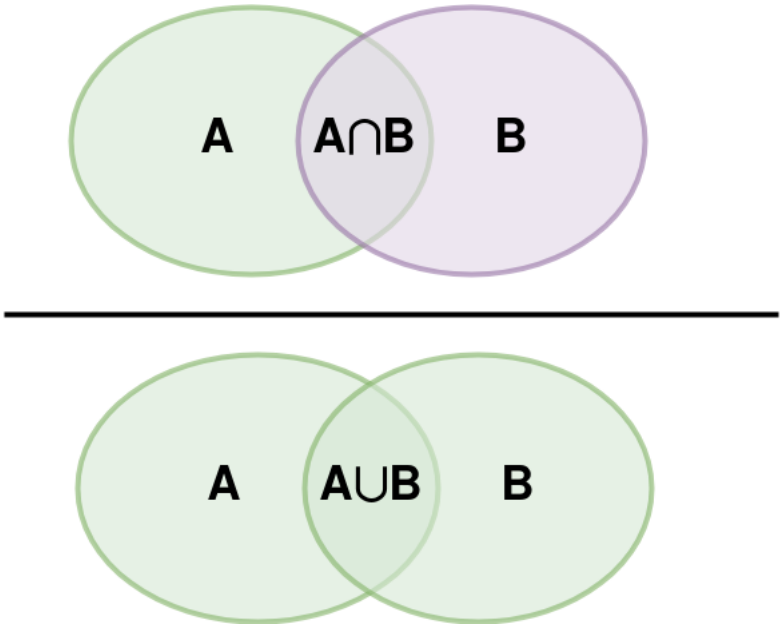

Figure 2.3: Jaccard Similarity.

This measure is specially useful for *implicit feedback* since every *user/item* consists of a vector of binary ratings, causing the *jaccard similarity* to measure the relative number of common items rated by both users. However, the actual values of *non-binary* ratings are not taken into account [23]. We could ignore values in the matrix and focus only on the sets of items rated. If the ratings matrix only reflected purchases, this measure would be a good one to choose. However, when ratings are more detailed, the Jaccard distance loses important information [19].
Another type of common similarity measure is the **cosine similarity** which measures the cosine of the smallest angle between the two vectors [33], as we can see in 2.4.

$$cos(\theta) = \frac{A.B}{||A||||B||} \tag{2.2}$$

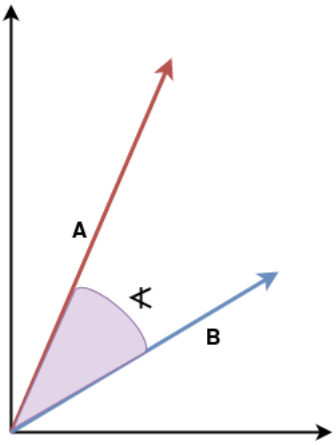

Figure 2.4: Angle between two vectors.

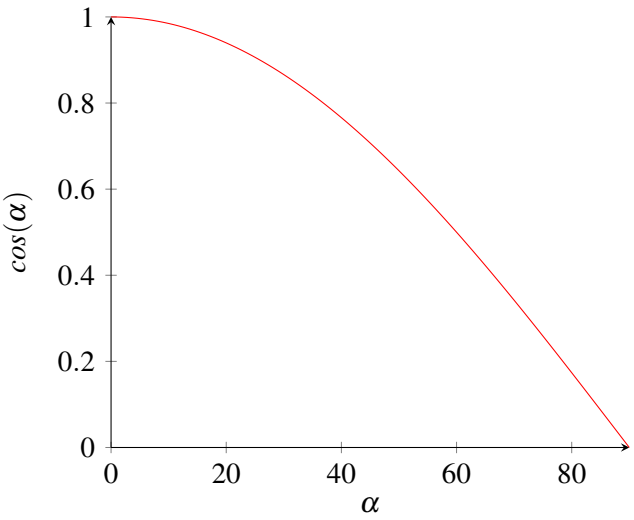

Figure 2.5: Cosine function.

We are only dealing with angles from the range *[0, 90]*. As we see from figure 2.5, as the angle increases, the cosine function decreases until it reaches 0. This means that higher angles between two vectors representing users or items will result in lower similarities between them.
Another used correlation measure is called the **Pearson Correlation Coefficient**, which measures the strength and direction of linear relationships between pairs of continuous variables. For 2 sample variables, A and B of the same size n:

$$\rho(A,B) = \frac{\sum_{i=1}^{n}(A_i - \overline{A}) \times (B_i - \overline{B})}{\sqrt{\sum_{i=1}^{n}(A_i - \overline{A})} \times \sqrt{\sum_{i=1}^{n}(B_i - \overline{B})}} \tag{2.3}$$

For recommender systems A and B will be representing user or item vectors where each element is a rating. If we use population terms for these samples the numerator can be viewed as the

covariance and the 2 denominator can be viewed as the **variances**. The *Pearson correlation coefficient* measures linear correlation between two vectors [35], and its values range from *[-1, +1]* where +1 means total positive linear correlation, -1 means total negative correlation and 0 means no linear correlation, as explained in 2.6.

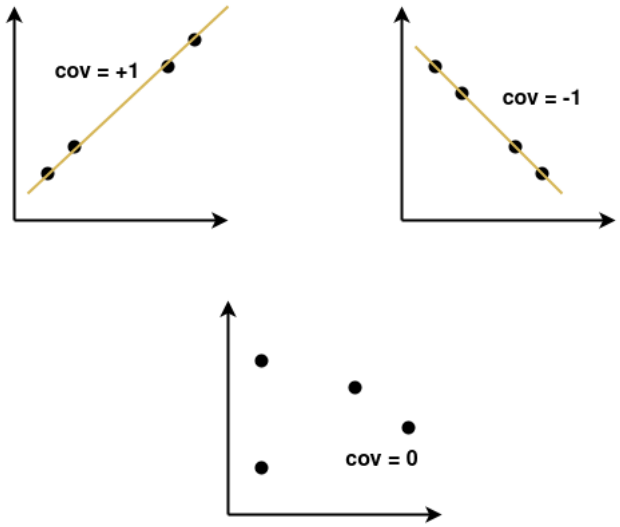

Figure 2.6: Pearson Correlation Coefficient.

These similarities' heuristics when calculated for pairs of *users/items* help build a neighborhood model to process recommendations. Essentially, neighborhood-based models are divided into two categories: The **user-based** and **item-based** models. On the next sections we will discuss methods of recommendation using both types of neighborhood based collaborative filtering, their main advantages and disadvantages and ways of tackling the neighborhood problem.

2.1.1.1 User-based collaborative filtering

In this approach, the neighborhood model is constructed using similarities between pairs of users. The most common way of doing this is through the Pearson correlation coefficient. Lets assume two users, u and v and let their respective list of indexes of rated items be I_u and I_v.

$$\mu_u = \frac{\sum_{k \in I_u} r_{uk}}{|I_u|} \tag{2.4}$$

$$Pearson(u,v) = \frac{\sum_{k \in I_u \cap I_v} (r_{uk} - \mu_u).(r_{vk} - \mu_v)}{\sqrt{\sum_{k \in I_u \cap I_v} (r_{uk} - \mu_u)^2} \sqrt{\sum_{k \in I_u \cap I_v} (r_{vk} - \mu_v)^2}} \tag{2.5}$$

In equation 2.5, μ_u represents the mean rating of a user u, and r_{uk} represents the rating of user u to item k. The *Pearson* correlation acts as the similarity between user u and v and has to be calculated for the target user and all the other users. Through the neighborhood of a user u we can make inferences to the predicted rating for an unrated item i by u. We do this through the ratings

of other users in the neighborhood of u which rated i. When we are specifying a target for which to predict a rating for a user, the ratings of the set of users with an higher pearson correlation vary significantly, so we must return a **weighted average** of the **mean centered** ratings as a predicted rating. This has to do with the fact that different users use the rating scale differently, or interpret the values in the scale in a different way.

$$s_{ui} = r_{ui} - \mu_u \tag{2.6}$$

The mean rating of the target user is then added back to this prediction to provide a raw rating prediction. Let $N_u(j)$ be the set of the k users with the highest Pearson correlation with user u who specified ratings for item j

$$\hat{r_{uj}} = \mu_u + \frac{\sum_{v \in N_u(j)} Sim(u,v).(s_{vj})}{\sum_{v \in N_u(j)} Sim(u,v)} \tag{2.7}$$

Note that, when using the *Pearson correlation coefficient* the ratings are computed over the *co-rated* items of the designated pair of users, in equation 2.5 represented by $I_u \cap I_v$. Several other variants of similarity function are used in practice. Items with an higher predicted rating are then recommended to the user.
One variant of this method is to use the cosine function on the raw ratings rather than the mean-centered ratings [6].

2.1.1.2 Item-based collaborative filtering

In *item-based collaborative filtering*, we focus on using pairwise item similarities to build a neighborhood for each item in the system. For items, it is common to use the *cosine similarity* for computing similarities. Assuming U contains the set of users in the system and U_i, U_j represents the users who rated item i and j respectively:

$$Cosine(i,j) = \frac{\sum_{u \in U_i \cap U_j} r_{ui} \times r_{uj}}{\sqrt{\sum_{u \in U_i \cap U_j} r_{ui}^2} \sqrt{\sum_{u \in U_i \cap U_j} r_{uj}^2}} \tag{2.8}$$

Like before, we should center these item's ratings in terms of a mean, getting a mean-centered rating. Here, we will use the item's average rating.

$$s_{ui} = r_{ui} - \mu_i \tag{2.9}$$

$$AdjustedCosine(i,j) = \frac{\sum_{u \in U_i \cap U_j} s_{ui} \times s_{uj}}{\sqrt{\sum_{u \in U_i \cap U_j} s_{ui}^2} \sqrt{\sum_{u \in U_i \cap U_j} s_{uj}^2}} \tag{2.10}$$

If we define N_i a set of *top-k* items most similar to i for which the user has provided a rating, the predicted similarity is:

$$\hat{r_{ui}} = \frac{\sum_{j \in N_i} AdjustedCosine(i,j) \times r_{uj}}{\sum_{j \in N_i} |AdjustedCosine(i,j)|} \tag{2.11}$$

If we can predict a rating of a user to an item, then we can recommend for a user, the items whose predicted score is the highest between the different candidates.

2.1.1.3 Item-based vs User-based

Neighborhood-based methods start with an **offline** phase and proceed to an **online** phase. In the offline phase, the similarity values of pairs of users(or items) are computed to form peer groups and store these groups for each user (or item). These peer groups are the *top-k* more similar elements (items or users). In the online phase, predictions are formed based on the peer groups and similarities as we've seen before. Let *k'* be the maximum number of specified ratings by a user, and *p'* be the maximum number of specified ratings to an item. In this sense, *k'* is the maximum running time for computing the similarity between a pair of users, and *p'* is the maximum running time for computing the similarity between a pair of items. For *user-based* methods, the process of determining the peer group of a target user may require $O(m \times k')$ time. Therefore, the offline running time for computing the peer groups of all users is given by $O(m^2 \times k')$. For item-based methods, the corresponding offline running time is given by $O(n^2 \times p')$. Plus, every time there is a rating update, a user is added, or a new item appears in the system, the entire neighborhood has to be recalculated because there is no way of knowing beforehand, that if with an update to an element, the updated element leaves the neighborhood of another element or enters another, or if its neighbors change.

Item-based recommendations rely on the users own ratings and then recommend items similar to the ones he intends to like. Inherently, since users own ratings are very indicative of items he might like, item-based methods often provide better recommendations and are more likely to be accurate. However, of course, this accuracy issue depends on the dataset at hand. In terms of **processing time**, it is obvious that if a system has more users than items then user-based algorithms will take a longer time to run than item-based ones and vice-versa. However, generally speaking, in online businesses the amount of users is normally higher than the number of items which has some consequences. This leads to pairs of users having a small number of *co-rated* items, but pairs of items having a larger number of users who *co-rated* them. And since the neighborhood has to be recalculated every time a new user or a new item is added, and since new users are added more frequently, user-based methods spend more time recalculating neighborhoods for their users that item-based methods do for their items.

2.1.1.4 Locality-Sensitive Hashing

As mentioned in section 2.1.1.3, a problem with common neighborhood-based approaches has to do with their scalability as data grows. Since classic algorithms need to verify each user or item, to calculate the *k-nearest-neighbors*, it does not scale well with large amounts of data, so it is not a very good option for stream-based recommender systems. One way of solving this problem is to use **ANN** (Approximate Nearest Neighbor) search. Like the name suggests, instead of calculating neighbors based on a heuristic we come up with a way of approximating the *"true"* neighborhood model with another, creating a tradeoff between accuracy and performance. One technique to perform *ANN* is called **Locality-Sensitive Hashing**. *LSH* attempts to create a way with which there is a high probability of similar elements getting hashed to the same *"bucket"*. *Locality-Sensitive Hashing* defines a series of hashing functions which compose an hashing family. Using *AND-CONSTRUCTION* [8], we can construct a random hashing function which hashes each element to a bucket. Elements in the same bucket form a candidate pair. This means that, if we have k hashing functions, $h_1, h_2, ..., h_k$, elements, u and v are a candidate pair if for every $i \in [0,k]$, $h_i(u) = h_i(v)$. To see neighborhoods of users or items we hash these elements and check other users or items with the same hash, that is, mapped to the same bucket.

ALGORITHM 1
Locality-Sensitive Hashing

$e \leftarrow element$
$H \leftarrow HashMap()$
$F \leftarrow [f_1, f_2, f_1, ..., f_n]$
for $f \in F$ **do**
  $H[f(e)] \leftarrow e$
end for

2.1.1.5 Clustering

Another possible way of solving the high complexity of the offline phase of the neighborhood-based collaborative filtering mentioned in section 2.1.1.3 is by **clustering** the matrix. The idea behind this is to reduce the number of elements to calculate similarities. Whether we decided to use *user-based* or *item-based* the technique is based on clustering rows or columns and then calculating the peer groups of users using only their clusters as a comparison. One common algorithm for clustering is the **k-means clustering**. Assuming c is a map of centroids to a set of rows in that cluster, g a set of centroids and M the ratings matrix.
The clusters are sets of users(for *user-based*) or items(for *item-based*), and the centroids are the mean values of these sets of users or items respectively. The increase of efficiency results in a decrease in accuracy because the closest set of neighbors inside the cluster has lower quality than the ones who would be in an entire dataset. This tradeoff varies according to the number of clusters one uses to wish for the *k-means*. As we increase the number of clusters, since clusters are smaller we improve efficiency, but we reduce the accuracy. This section could have been covered

ALGORITHM 2
K-Means Clustering

$c \leftarrow 0$
$old_g \leftarrow 0$
$g \leftarrow$ set of random generated centroids
while $g \neq old_g$ **do**
  $old_g \leftarrow g$
  for $u/i \in M$ **do**
    $centroid \leftarrow min(g, i, measureDistance)$
    $c[centroid] \leftarrow i$
  end for
  for $(centroid, cluster) \in c$ **do**
    $g.remove(centroid)$
    $g.add(mean(cluster))$
  end for
end while

in section 2.1.2, since it is considered to be a **model-based** approach, however, since if fitted within one of the ways the neighborhood approach could be changed to perform more efficiently it was positioned here.

2.1.2 Model-Based Collaborative Filtering

To avoid the main drawbacks of *neighborhood-based* methods which are related to their intense space and time usage as data grows making them non-scalable there is another type of *collaborative filtering* approach. **Model-Based Collaborative Filtering** attempts to compress a huge dataset into a model and create a referencing mechanism to provide recommendations out of that model [10]. In one way, like in machine learning, this type of collaborative filtering builds a model summarizing the data available separating the **model-building** phase from the **rating-prediction** phase. In traditional machine learning, as we are going to see, this can be closely related to **Classification**. In another way, there is no clear separation from what is a independent variable or not, there is no separation from training set or testing set, and there is no separation between features and data instances. These types of models have some advantages over *neighborhood-based* because they are more **space-efficient**, provide an higher **training-speed** and avoid **overfitting**. Over the next sections we will describe a number of *model-based* methods for *collaborative filtering*.

2.1.2.1 Naive-Bayes

The Naive-Bayes model works best for categorical data on which ratings belong to a set of l possible categorical values $r \in \{v_1, v_2, ..., v_l\}$. For instance a rating could be *Like*, *Neutral*, or *Dislike*, as presented in figure 2.7.

user_id	item_id	label
1	1	Like
1	2	Dislike
2	3	Dislike

Figure 2.7: Ratings table.

For collaborative filtering, if we are predicting a rating given by a user u to an item j, then the we would like to determine the probability of user u rating j with one of the values he has rated items with, represented by I_u. In other words, $P(r_{uj} = v_s | I_u)$. According to the **Bayes Theorem**:

$$p(r_{uj} = v_s | I_u) = \frac{P(r_{uj} = v_s) \times P(I_u | r_{uj} = v_s)}{P(I_u)} \tag{2.12}$$

The expression $P(r_{uj} = v_s)$ can be the fraction of users who rated the jth item with v_s, including only the ones who actually rated item j. The term $P(I_u | r_{uj} = v_s)$ is determined using a **naive assumption**, assuming conditional independence between all the ratings:

$$P(I_u | r_{uj} = v_s) = \prod_{k \in I_u} P(r_{uk} | r_{uj} = v_s) \tag{2.13}$$

Here, $P(r_{uk}|r_{uj} = v_s)$ is computed as the the fraction of users who rated the *kth* item as r_{uk} given that they rated the *jth* item as v_s. In the end the value of v_s with an higher probability value is returned as a predicted rating.

2.1.2.2 Decision Trees

Decision Trees consist of a series of hierarchical partitioning of the data space into decisions, which are known as the **split criteria**. The construction of the tree is a recursive process, which starts at the root node, and from there on, at each node, an item's attribute is picked as the split attribute. The choice of the split attribute is through a heuristic, and there are many different ways of doing this. The recursive process continues until all the items in the node's set share the same target attribute value or the number of items reaches a certain threshold [24].

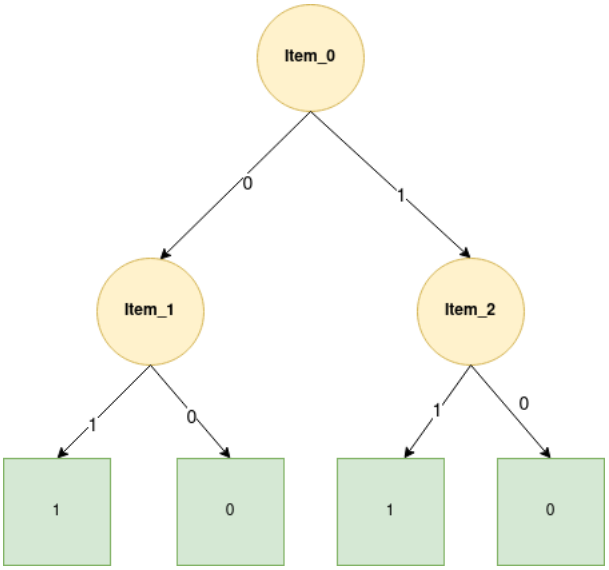

Figure 2.8: Decision Tree for a predicted rating.

Regarding *collaborative filtering*, user feedback is used for each item in the system, and a decision tree is built for each item which **consumes a considerable amount of memory**. The feedback on a targeted item is considered to be a decision for the final prediction, as we can see in figure 2.8. When talking about recommender systems we need to take into account that there is no clear difference between dependent and independent variables. So for if we are predicting a rating of an item j for a user u, all the items in the system become independent variables except j whose predicted rating will depend on the ratings of u for the items in the tree.

2.1.2.3 Latent Factor Models

These models leverage well-known dimensionality reduction methods to fill in the missing entries [6]. By rotating the axis system, pairwise correlations are removed, and the reduced and rotated representation can be estimated from the original incomplete *user-item* matrix. They are also

known as **Matrix Factorization** methods, and combine row and column correlations in one shot to estimate blank ratings.

A geometric perspective

For a geometric perspective on what latent factors are, in terms of recommendations, lets assume we have a ratings matrix with m rows and n columns, $R : \mathbb{N}^{m \times n}$. For a geometric representation of items, we would have m axis and the points' coordinates would be the ratings users gave to those items. So, suppose we had 3 items, all correlated, between each other, the 3 dimensional scatter plot could be represented as a line. This also means that the rank of the matrix is equal to 1. In this case, only one rating would need to be specified so that the other 2 could be predicted, by intersecting the given rating hyperplane with the line. This line is called the **latent vector**. We can generalize this and say if the R matrix has a rank p, such as $p << min(m,n)$, the predicted ratings will be the intersection of the *p-dimensional* hyperplane with the **latent vectors**. We attempt to represent all the points scattered in the domain space as an approximation projected on the line, as we can see in figure 2.9.

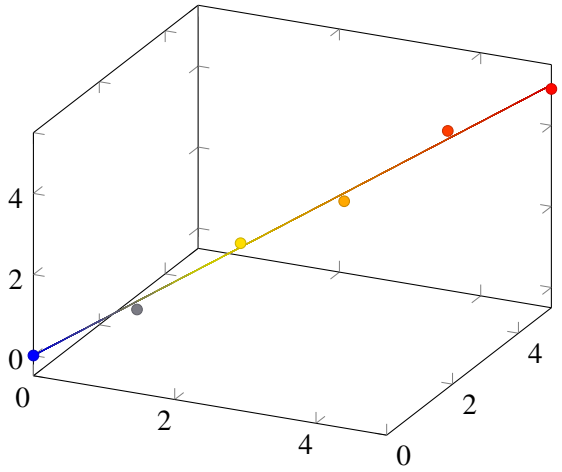

Figure 2.9: Latent factor models - Geometric view.

An algebraic approach

Any matrix $R \in \mathbb{N}^{n \times m}$, where n is the number of rows and m is the number of columns, and $rank(R) << min(m,n) = k$ can be expressed as:

$$R = UV^T, U : \mathbb{N}^{m \times k}, V : \mathbb{N}^{n \times k} \tag{2.14}$$

We can interpret the columns of U as the k basis vectors of the column space of R and the rows as the coefficients of the basis vectors of the row space of R. In the same way, the rows of V contain the coefficients which combine the basis vectors of U into the column space of R. As the rank of

the matrix grows, the equation turns into an approximation:

$$R \approx UV^T \tag{2.15}$$

The absolute error is calculated by $||R - UV^T||$ where $R - UV^T$ is the **residual matrix**. We can see that the *user-item* interactions are now going to be represented as inner products in this space [22]. That being said, it is worthwhile to formulate a valid error expression. The **squared error** is calculated using the absolute error to the power of 2: $||R - UV^T||^2$. Taking this expression, the **mean squared error** is computed by dividing the *squared error* by the n observed ratings:

$$MSE = \frac{||R - UV^T||^2}{n} \tag{2.16}$$

.

Introducing the concept **l2-regularization**, which penalizes large weights in the error function to lower model variance [3]:

$$L2 = \frac{||R - UV^T||^2}{n} + \lambda \times (||U||^2 + ||V||^2) \tag{2.17}$$

.

The term λ is the regularization term.

Matrix Factorization

Following the previous idea, each column of the result of the factorization is a **latent vector** whose dimensionality is equivalent to the number of **latent factors**. The rows in U are known as **user factors** and the rows in V are known as **item factors**. These last 2 factors represent the affinity of users and items respectively towards the concepts on the original R matrix. So if we are trying to predict a rating of a user i to an item j:

$$\hat{r_{ij}} \approx U_i.V_j^T \tag{2.18}$$

However, we are assuming that the factorized matrix R was not sparse and was entirely filled with ratings. This is not true in real case situations so, the question lies on how to factorize sparse matrices. Before jumping into that let us just make a note on the special case of **explicit feedback**. This type of feedback causes some issues due to the differences in the popularity of items and rating scales of users. Therefore, it is important to create a **preprocessed** matrix to be factorized by normalizing the ratings.:

$$normalized\_r_{ua} = r_{ua} - 0.5 \times \overline{r_u} - 0.5 \times \overline{r_a} \tag{2.19}$$

Stochastic Gradient Descent for Matrix Factorization

Stochastic Gradient Descent is an algorithm which intends to optimize an objective function in a smooth way making use of the gradient function of the error associated with the objective function, known as the **loss function**. Let S be a set of *user-item* pairs that have been rated in the rating matrix R, where each pair is *(i,j)*, $i \in \{1,2,3,...m\}$, $j \in \{1,2,3,...n\}$, and $S = (i,j) : r_{ij} \neq null$

ALGORITHM 3
Stochastic Gradient Descent for Matrix Factorization

$U \leftarrow random$
$V \leftarrow random$
while $not(convergence)$ **do**
  $R_S \leftarrow randomize(S)$
  for $(i,j) \in R_S$ **do**
    $e_{ij} \leftarrow r_{ij} - \overrightarrow{u_i}.\overrightarrow{v_j}$
    $\overrightarrow{u_{tmp}} \leftarrow u_i + \alpha \times (e_{ij} \times \overrightarrow{v_j} - \lambda \times u_i)$
    $\overrightarrow{v_{tmp}} \leftarrow v_j + \alpha \times (e_{ij} \times \overrightarrow{u_i} - \lambda \times v_j)$
    $\overrightarrow{u_i} \leftarrow u_{tmp}$
    $\overrightarrow{v_j} \leftarrow v_{tmp}$
  end for
end while

On the pseudocode, α is the learning rate, also known as the step size. This parameter is set in a way to influence the convergence of the algorithm. With a larger value, bigger steps are taken at every iteration and the algorithm might **diverge**. Lower values make the algorithm have an higher probability to **converge**, with an higher amount of iterations. The **convergence criterion** can be understood as when the rating error has reached an acceptable value or when a maximum number of iterations has passed. At each iteration the algorithm does one passage through the dataset, updating the values of the factorized matrices according to a learning rate and an error. λ stands for the regularization factor which is used to avoid overfitting. When the algorithm ends, we will have two matrices, U and V, and can predict ratings as previously discussed. For data stream mining it is common to implement algorithms in a way that it can be distributed among different independent clusters. This is generally difficult to do for *SGD*, however, there is a proposal to do so in [13], with the exploit of the summation form of the loss function at each iteration by computing the local gradients in parallel and summing them.

Alternate Least Squares for Matrix Factorization

Alternate least squares attempts to minimize the *l2-regularized error* by fixing a set of latent vectors(e.g matrix U) and treating it as a constant and updating the other set and then alternating the fixed set to the opposite(V) and updating the other one. We keep doing it back and forth until **convergence**. Assuming we are predicting a rating of a user i to an item i, let us try to minimize

the **l2-regularized error** with *ALS*:

$$L(R) = ||R_{ij} - U_i V_j^T||^2 + \lambda \times (||U_i||^2 + ||V_j||^2) \tag{2.20}$$

The gradient function $\nabla(L(R))$ is computed as:

$$\frac{\partial L(R)}{\partial U} = R_i V_j (V_j V_j^T + \lambda I)^{-1} \tag{2.21}$$

$$\frac{\partial L(R)}{\partial U} = R_j U_i (U_i U_i^T + \lambda I)^{-1} \tag{2.22}$$

Let S be a set of *user-item* pairs that have been rated in the rating matrix R, where each pair is *(i,j)*, $i \in \{1,2,3,...m\}$, $j \in \{1,2,3,...n\}$, and $S = (i,j) : r_{ij} \neq null$

ALGORITHM 4
Alternate Least Squares for Matrix Factorization

```
U ← random
V ← random
while not(convergence) do
  for i ∈ 0...m do
    for j ∈ R(i, j) do
      U_i ← R_j V_j (V_j V_j^T + λI)^{-1}
    end for
  end for
  for j ∈ 0...n do
    for i ∈ R(i, j) do
      V_j ← R_i U_i (U_i U_i^T + λI)^{-1}
    end for
  end for
end while
```

2.2 Content-Based Recommender Systems

One problem with collaborative filtering has to do with the **cold start** concept. When a product is inserted into the system, it initially has no ratings whatsoever by any user, making it impossible to recommend that product. **Content-Based** recommendation instead of focusing on all the ratings by different users across the platform, makes use of the set of attributes that can describe an item, and try to match users to items they have liked in the past. They acquire these attributes by looking at data from item descriptions, which are common in large scale web platforms, such as *Shopify* whose stores provide descriptions for their products. These recommender systems also analyze user feedback, being it either *implicit* or *explicit*, but only for the user whose recommendations are being calculated for. Then a **user profile** is built mapping user feedback to item attributes. An advantage of content-based models is, as previously mentioned, when it is used for *cold-start scenarios* with items, partially alleviating the problem. However, *cold-start scenarios* with users are still a problem, and disregarding ratings from other users may lead to a repetitive recommendation of the same set of products. This is explained by the fact that the obvious choices of recommendation may be items that the user has already consumed, or at least has already seen, since the set of attributes for the items are similar. Recommending always the same items will pose little surprise to the user. Now, a series of phases that build up to be a content-based recommender system will be described.

2.2.1 Unupervised feature selection

Normally, items are represented with unstructured documents(plain text), so one should apply feature extraction techniques for items to representable as a **Bag-Of-Words** or as a multidimensional vector. However, before that, the text has to be preprocessed and cleaned, with a series of techniques listed below:

- **Remove Stop Words** remove a set of useless words such as *"a"*, *"at"*, *"an"*, which appear very frequently in the text and are not very specific for the item at and.
- **Tokenize the text** by breaking down the document's text into different words and representing a document in memory by its words. One example is to divide the document by the space character and getting the words that way.
- **Stemming the words** which means to get the base word and removing the derivational affixes.
- **Lemmatization** by removing the inflectional ending of the words and getting the root word ensuring it belongs to a language.

After these modifications, each word will be transformed into a vector-space representation which will be composed by the word itself and its **frequency**. The document will be represented as a bag of words which is a set of vector-space representations of keywords. Words are known in

information retrieval systems as **terms**. The raw frequency is not used, instead, firstly an **inverse document frequency** is calculated due to the existence of more common words who were not eliminated when removing stop words, which have a high frequency and are statistically less discriminative. Assuming n as the number of documents in a document collection, or as the number of item descriptions(if there is one per item, this will mean the number of items), and n_i the number of items in which the term i occurs:

$$IDF_i = log(n/n_i) \tag{2.23}$$

Then, we can use the raw term frequency or a normalized form of it, using *log* or *squared root* functions to calculate the **Term Frequency - Inverse Document Frequency** of the word in the document. Assuming x_i the raw frequency of the *ith* term:

$$TF - IDF_i = f(x_i) \times IDF_i \tag{2.24}$$

Where $f(x_i) = x_i$ or, using a normalized version $f(x_i) = log(x_i)$, $f(x_i) = \sqrt{x_i}$.

2.2.2 Supervised Feature Selection

In the previous section, we discussed **unsupervised feature selection** with the removal of stop words and the calculation of *tf-idf* values for the keywords. These methods do not take into account user feedback, so now we will describe a set of useful techniques that do so.

2.2.2.1 Gini Index and Entropy

The **Gini Index** is commonly used for binary or ordinal ratings. Assuming a specific term w, t as the number of possible ratings the documents can have and $\{p_1(w), p_2(w), ...p_t(w)\}$ as the fraction of items rated with one of those t possible values:

$$Gini(w) = 1 - \sum_{i=1}^{t} p_i(w)^2 \tag{2.25}$$

The values of the previous equation has a range of $[0, 1 - \frac{1}{t}]$ and lower values suggest an higher discriminative power for the term. **Entropy** is very similar in principle to the Gini index except that information-theoretic principles are used to design the measure.

$$Entropy(w) = -\sum i = 1^t p_i(w) log(p_i(w)) \tag{2.26}$$

2.2.3 User Profiles and Filtering

Firstly let us assume we have a set D of documents who can be interpreted as textual descriptions. Some of these items are labeled(rated) by the active user and some of them are not. The objective is to be able to label the ones which are not yet labeled with predictions. To do so, a model needs to be computed for every user, differing from matrix factorization, where we had a global model for all the users. Now we will explore a series of algorithms that can be used for content-based filtering.

2.2.3.1 Nearest Neighbor Classifier

Lets imagine we have two documents $\overrightarrow{x} = \{x1...x_d\}$ and $\overrightarrow{y} = \{y1...y_d\}$ composed of normalized term term frequencies calculated using either tf-idf or other supervised techniques already discussed. In $\overrightarrow{x}$ the similarity between them can be calculated as $cos(\overrightarrow{x}, \overrightarrow{y})$. For each document whose label we are trying to predict, we can calculate their k-nearest neighbors and then average their ratings using a **weighted average** and return the value as a prediction. The problem with this technique is its complexity, which is linear. We need to determine the nearest neighbors of all the documents whose labels are not known, D_u, and the calculations grow linearly as the amount of documents whose ratings we know, D_l , grows: $O(|D_u| \times |D_l|)$. One way of solving this is through **clustering**.

2.2.3.2 Naive-Bayes Classifier

For this section, we will assume the **Bernoulli** model for data representation where labels belong to the $0, 1$ set. Lets assume we have a user u and a document $\overrightarrow{x}$, where $\overrightarrow{x} = \{x_1...x_d\}$ What we are trying to compute is the $P(u \text{ liking } \overrightarrow{x}|x1...x_d)$. Since we are dealing with *binary* ratings we can express the option of user u towards item $\overrightarrow{x}$ by, *u(x)*. So we transform the previous expression into $P(u(x) = 1|x1...x_d)$.

$$P(u(x) = 1|x1...x_d) = P(u(x) = 1) \times \prod_{i=1}^{d} P(x_i|c(x) = 1)$$
$$P(u(x) = 0|x1...x_d) = 1 - P(u(x) = 1|x1...x_d)$$

Here, $P(u(x) = 1)$ is the fraction of instances where the user has liked an item. For **overfitting** purposes we can use **Laplacian** smoothing:

$$P(u(x) = 1) = \frac{|u(1)| + \alpha}{|u| + 2 \times \alpha} \tag{2.27}$$

Here, $P(x_i|u(x) = 1)$, is the fraction for the number of times when a user has liked an item and the *ith* feature of that item takes the value of x_i. This computation can also be processed with Laplacian smoothing. In the end the predicted rating should be the one with an higher probability.

2.2.3.3 Decision Trees

In section 2.1.2.2, we explored the idea of using decision trees for collaborative filtering. For content-based recommenders, the features of each of the items are used to build a model that explains the user's preferences, so the information gain of every feature is used as splitting criteria. In [39], they remember the well-known **ID3** tree algorithm. Lets consider $A = \{a_1, a_2, ..., a_n\}$, as a set of attributes set, $V = \{v1, v2, v..., v_i\}$ as a set of sets of values for those attributes, where v_{ij} represents the *jth* possible value for the *ith* attribute. In the case of collaborative filtering, A could be the items in the system, V could contain the values (binary or not) which users have rated that item with, and for content-based recommender systems, A could be item's attributes (color, genre, etc..) and V the set of possibilities for those attributes. To construct the tree:

ALGORITHM 5
Decision Trees

if v_i contains always the same value **then**
  $Tree.append$(Leaf with that value)
else
  $a_{best} \leftarrow$ attribute for which $E(a)$ is the min
  for $v_{best,i}$ **do**
    $Tree.append$(Node from a_{best} to recursive(v_{best}))
  end for
end if
return Tree

Chapter 3

Incremental Recommendations

Incremental algorithms are becoming a serious need for online processing, due to their inherent nature of being able to process incoming data streams incrementally and updating models. **Incremental learning** aims at building a model that adapts to incoming and dynamic change and reducing the cost of model update and retraining [34]. This makes *incremental learning* specially useful for **data stream mining**. In this chapter, we will discuss a series of incremental algorithms directly applied in the recommendation field, as well as some related work about other potentially useful incremental algorithms. Such as in 2 we will divide the following sections according to their specific field on recommendation systems. All the approaches about to be mentioned are located in the **Collaborative Filtering** area of *recommendation systems*.

3.1 Neighborhood-based

3.1.1 Item-based

For both of the approaches we are about to see, they focus on the usage of *implicit ratings*, so an item i could be represented as $\overrightarrow{i} = \{1,0,0,1,.....\}$. In both approaches the pairwise similarity is calculated using *cosine similarity*:

$$cos(\overrightarrow{i}, \overrightarrow{j}) = \frac{|\overrightarrow{i} \cap \overrightarrow{j}|}{\sqrt{|\overrightarrow{i}|}\sqrt{|\overrightarrow{j}|}} \tag{3.1}$$

In equation 3.1, $|\overrightarrow{i} \cap \overrightarrow{j}|$, represents the number of users who co-rated items i and j, and $\sqrt{|\overrightarrow{i}|}$ stands for the **L1 norm** or **Manhattan Distance** of $\overrightarrow{i}$, or the number of users which rated item $\overrightarrow{i}$.

The first example we are going to look at is [25] which uses the concept of **activation weight**, representing this weight of an item i as $W(i)$. Let $Q_a(i)$ be the items in the neighborhood of i seen by user a and $Q(i)$ the items on the neighborhood of i. For an active user, u_a, we calculate every

activation weight for the items he has not seen yet:

$$W(i) = \frac{\sum_{j \in Q_a(i)} S(i,j)}{\sum_{j \in Q(i)} S(i,j)} \tag{3.2}$$

Then, we recommend the items with an higher activation weight. They add incrementalism by also storing another matrix(besides S), denoted as Int with the numbers of users who evaluate each pair of items. So, if we represent a data stream I_a which holds items a user a has rated:

ALGORITHM 6
Incremental IB CF

for Pairs $(i,j) \in I_a$ **do**
  $Int_{ij} \leftarrow Int_{ij} + 1$
  $S_{ij} \leftarrow cos(\overrightarrow{i}, \overrightarrow{j})$
end for

Another two approaches were used in [20], the **Dynamic Index** and a **Parallel** procedure of the dynamic index algorithm based on the **MapReduce** paradigm. Their *dynamic index algorithm* uses an inverted index to identify affected pairs of items when a new data stream arrives, making use of data sparsity. User binary preferences are called **pageviews**. They introduce the concept of recommendable items, which consist on the set of items that can be recommended at a given point in time because they gather a set of characteristics(i.e, stock). To incrementally update these sets they store in a data structure, a matrix of $n \times m$ items containing all *item-pair* intersections at a given time t, $M_t \in \mathbb{N}^{n \times m}$, and a array of items **l1-norms** $N_t \in \mathbb{N}^n$. In algorithm 7, L_n repre-

ALGORITHM 7
Dynamic Index

for $(u,i,s) \in P_t$ **do**
  for $j \in L_r[u]$ **do**
    $M_{i,j} \leftarrow M_{i,j} + 1$
  end for
  if $i \in R_t$ **then**
    for $j \in L_n[u]$ **do**
      $M_{i,j} \leftarrow M_{i,j} + 1$
    end for
    $L_r[u] \leftarrow L_r[u] \cup \{i\}$
    $N_i \leftarrow N_i + 1$
  else
    $L_n[u] = L_n[u] \cup \{i\}$
  end if
end for

sents an inverted index of users to non recommendable items and L_r stands for the same but to recommendable ones. From this algorithm they concluded that there is an independence between the contributions of different users to the similarity of two items. This is ideal, because we can

now separate each user in a different *map-procedure* and then the *reduce-process* will consist on summing the different matrices M and vectors N.

3.1.2 User-based

In this section we will focus on the work proposed in a paper [27], on which the authors propose an interesting way of incrementally updating the user similarities, making use of the Pearson Correlation terms. Remembering the formula in 2.3, in the following expressions, A refers to the **covariance** and C and D refer to the **variance**.

$$\begin{aligned} A &= \frac{B'}{\sqrt{C'}\sqrt{D'}} \\ B' &= B + e \\ C' &= C + f \\ D' &= D + g \end{aligned} \tag{3.3}$$

Considering a new rating incoming as a tuple (u_a, i_a, r_{u_a}):

ALGORITHM 8
Incremental UB CF

if u_y had already rated i_a **then**
  if u_a had never rated i_a **then**
    $e \leftarrow (r_{u_a,i_a} - \overline{r'_{u_a}})(r_{u_y,i_a} - \overline{r_{u_y}}) - \sum_{h=1}^{n'}(\overline{r'_{u_a}} - \overline{r_{u_a}})(r_{u_y,i_h} - \overline{r_{u_y}})$
    $f \leftarrow (r_{u_a,i_a} - \overline{r'_{u_a}})^2 + n'(\overline{r'_{u_a}} - \overline{r_{u_a}})^2 - 2\sum_{h=1}^{n'}(\overline{r'_{u_a}} - \overline{r_{u_a}})(r_{u_a,i_h} - \overline{r_{u_a}})$
    $g \leftarrow (r_{u_y,i_a} - \overline{r_{u_y}})^2$
  else
    $e \leftarrow (r'_{u_a,i_a} - r_{u_a,i_a})(r_{u_y,i_a} - \overline{r_{u_y}}) - \sum_{h=1}^{n'}(\overline{r'_{u_a}} - \overline{r_{u_a}})(r_{u_y,i_h} - \overline{r_{u_y}})$
    $f \leftarrow (r'_{u_a,i_a} - r_{u_a,i_a})^2 + 2(r'_{u_a,i_a} - r_{u_a,i_a})(r_{u_a,i_a} - \overline{r'_{u_a}}) + n'(\overline{r'_{u_a}} - \overline{r_{u_a}})^2 - 2\sum_{h=1}^{n'}(\overline{r'_{u_a}} - \overline{r_{u_a}})(r_{u_a,i_a} - \overline{r_{u_a}})$
    $g \leftarrow 0$
  end if
else
  if u_a had never rated i_a **then**
    $e \leftarrow -\sum_{h=1}^{n'}(\overline{r'_{u_a}} - \overline{r_{u_a}})(r_{u_y,i_h} - \overline{r_{u_y}})$
    $f \leftarrow n'(\overline{r'_{u_a}} - \overline{r_{u_a}})^2 - 2\sum_{h=1}^{n'}(\overline{r'_{u_a}} - \overline{r_{u_a}})(r_{u_a,i_h} - \overline{r_{u_a}})$
    $g \leftarrow 0$
  else
    $e \leftarrow -\sum_{h=1}^{n'}(\overline{r'_{u_a}} - \overline{r_{u_a}})(r_{u_y,i_h} - \overline{r_{u_y}})$
    $f \leftarrow n'(\overline{r'_{u_a}} - \overline{r_{u_a}})^2 - 2\sum_{h=1}^{n'}(\overline{r'_{u_a}} - \overline{r_{u_a}})(r_{u_a,i_h} - \overline{r_{u_a}})$
    $g \leftarrow 0$
  end if
end if

When calculating the similarities they split the Pearson Correlation into 3 factors, *B'*, *C'* and *D'*. All these factors are calculated using the previous ones plus the increments e, f and g. If we

consider the similarity between two users, we must consider their *co-rated* items for calculations, represented as *n'*. This way, the covariance between two users u_a and u_y:

$$covariance(u_a, u_y) = \sum_{h=1}^{n'} (r_{u_a,i_h} - \overline{r_{u_a}}) \times (r_{u_y,i_h} - \overline{r_{u_y}}) \tag{3.4}$$

The variance of an user a is calculated as:

$$variance(u_a) = \sum_{h=1}^{n'} (r_{u_a,i_h} - \overline{r_{u_a}})^2 \tag{3.5}$$

In algorithm 8, you may notice the usage of "'" next to some variables, which are meant to represent the new values after the update. For instance, $\overline{r'_{u_a}}$, stands for the new average rating of user a after the new processing of the incoming rating.

3.2 Model-based

3.2.1 Matrix Factorization

When we talked about matrix factorization in 2.1.2.3, we did not address the issue of what would have to be done, if a new rating by a user would arrive. The algorithm trains a model in order to minimize an L_2**-regularized** squared error for the values in the *user-item* matrix which are known. If we factorize a ratings matrix R into 2 matrices U and V, then we want to minimize this error as

$$min_{U,V} \sum_{(u,i)\in D} (R_{ui} - U_u.V_i^T)^2 + \lambda(||U_u||^2 + ||V_i||^2).$$

Remember that $U_u \times V_i^T$ was the predicted rating $\hat{R}_{ui}$, and λ acts as the **regularization** term, which is used to avoid **overfitting**. We described a stochastic gradient descent approach for factorizing the ratings matrix, however the discussed algorithm would need to be repeated every time a new update is made. In [44] the authors propose a subtle alteration which can deal with incoming data streams more efficiently. Their work was based on **positive only feedback** so in the pseudocode below the error calculation was changed and adapted to not only positive feedback. Using the notation we previously used when describing the latent factor models, when an incoming set D of incoming data stream pairs *user-item (i,j)*, the authors **Incremental Stochastic Gradient Descent** is as follows:

ALGORITHM 9
Incremental Matrix Factorization

for $(i, j) \in D$ **do**
  $e_{ij} \leftarrow r_{ij} \overrightarrow{u_i}.\overrightarrow{v_j}$
  $\overrightarrow{u_i} \leftarrow \overrightarrow{u_i} + \eta(e_{ij} \times \overrightarrow{v_j} - \lambda \times \overrightarrow{u_i})$
  $\overrightarrow{v_j} \leftarrow \overrightarrow{v_j} + \eta(e_{ij} \times \overrightarrow{u_i} - \lambda \times \overrightarrow{v_j})$
end for

The main difference is the elimination of the outer loop of the algorithm, causing only one passage on data per incoming data stream and the elimination of shuffling.

3.3 Prequential Evaluation

One question arises when thinking about a method to evaluate these incremental machine learning algorithms. As previously discussed, these algorithms are dynamic and do not focus on a batch of data, instead deal with live dynamic data stream scenarios. A solution to this problem lies in the field of **Prequential Evaluation**, which creates a learning curve, monitoring the evolution of learning as a live process [11]. *Prequential Evaluation* is a method of evaluating each element of the data stream before being processed by an algorithm, which causes the first elements of the (**timely ordered**) stream to cause an impact on the error rate since the model has known very few instances when evaluating them. The machine learning model is not stationary, and will change as data grows. The prequential error can be calculated using a sliding window at time i using:

$$P_w(i) = \frac{\sum_{k=i-w+1}^{i} e_k}{w} \tag{3.6}$$

This is useful in order to avoid influences of errors at the beginning of the processing when little information was known beforehand, and we can visualize how well the algorithm is performing for the latest known information. In the equation, e_k represents the error.

Chapter 4

Software library for stream-based recommender systems

The main purpose of this work lies in this chapter which will explain the design and contents of the library: **increc**.
Increc is meant to be an open-source software library, designed in the field of **incremental learning** to deal with data streams for recommendation purposes. This chapter contains explanations regarding the implementation of *increc's* algorithms as well as used test metrics, evaluators, data structures, and existent dependencies. This chapter will also make reference to choices related to the chosen tech stack and how the library is being packaged and deployed. *Increc* focuses on the implementation of incremental **Collaborative Filtering** algorithms in contrast to **Content-Based** ones. This is explained due to the inherent simplicity in generalizing *Collaborative Filtering* problems and the difficulty in doing the same for *Content-Based* ones. *Collaborative Filtering* can be used every time there are user and item interactions that can be modeled in some specific way. This is more generic than *Content-Based* problems where *user/item* profiles might need to be built out of text information, or image/video pixels, or even audio feedback, which makes the problem more domain-dependent.

4.1 Problem Definition

Before any further discussion, we should first look at the problem at hand and try to visualize how that affects the overall architecture. When a platform has an online recommender system, it receives a stream of timely ordered data, which in turn will be incrementally processed by the system. This data includes information regarding user actions, and it can be represented by a tuple $(user_{id}, item_{id}, value)$ rating, where $user_{id}$ represents the user, $item_{id}$ is the item identifier and the *value* is the rating's preference for the item, typically expressed by a numerical value within a defined range. When considering **implicit feedback**, that value does not exist, and every pair of user/item is interpreted to be a positive interaction which, in *increc* is represented by the value **1**. Therefore, there must be a clear distinction and separation in the library between algorithms

regarding the type of input they receive. The first problem which immediately appears is that naturally, we need to dynamically scale the used data structures to accommodate new users and items. This problem is related to **memory**. A dynamic data structure is needed to be able to cope with this problem. The second problem is related to **scalability**. As the number of items and users grows, so does the number of necessary computations to keep recommendations updated which after some time can be a problem. In this chapter, we will explore how different algorithms react to this problem. Another problem arises in **measuring** the efficiency and accuracy of the recommendation model. This means we need to verify how much time an algorithm spends at processing an incoming rating and making a recommendation as well as if the recommendations are plausible or not, by including some evaluator within the library. The final problem is related to **visualization** because it makes sense to have some sort of mechanism to visualize how each algorithm evolves through time.

4.1.1 Related work

Regarding the purpose of this work, it is worth mentioning the existence of several recommendation libraries that already exist and represent state-of-the-art knowledge in this field. The first example is called **LibRec** [15]. *LibRec* is written in Java and implements a set of non-incremental algorithms for rating-prediction and item ranking in the collaborative filtering scenario. Another example is **My Media Lite** [12], written in *C#*, also used in collaborative filtering for rating prediction and item prediction with positive-only feedback. Another example is **recommenderlab**, [17], a package written in the *R* language which besides containing several collaborative filtering algorithms, also contains hybrid recommendations. Finally, a note on the usage of deep learning for recommender systems, which does not fall in the purpose of this document, in May 2020 **Nvidia** announced **NVIDIA Merlin** which is a framework that aims to provide fast feature engineering and high training throughput to enable fast experimentation and production retraining of deep learning recommender models [1]. The field of incremental recommendation has a lot of research and development to be done, but incremental algorithms are not new to machine learning. In [38], a new algorithm **ID5R** was proposed to incrementally update decision tree models, which can be useful for decision tree algorithms on content-based recommendation. Another example is [31], where they propose new ideas for incrementally updating probabilities in the Naive Bayes context. There is a lot of background on incremental algorithms, which with some tuning can be used for the recommendation field. Since the main topic of this work is building a software library, it is worth mentioning the libraries on the field of incremental algorithms. We can start with **MOA**, *Massive Online Analysis* which is related to the *WEKA* project and written in *Java*. *MOA* has a collection of Machine Learning algorithms that focuses on data stream mining as well as tools for evaluation and benchmarking. Similar to *MOA*, we also have another implementation in *Python* called **Scikit-Multiflow**, following the same philosophy as the well-known **Scikit-Learn**. Finally, it is of course worth mentioning the libraries in the field of incremental recommendation. **Flurs**, [36], is a *Python* library, used for online recommendation, with incremental collaborative filtering

algorithms. Another example is **FlowRec** [28], built on top of *Scikit-Multiflow* with baseline recommendation algorithms and the usage of prequential evaluation. However, the problem with the existent libraries is the scarcity of implemented approaches. There is no library which allows the development and evaluation of stream-based recommender algorithms and this work aims to fill that hole. Current approaches are too much task-specific and not easy to extend.

4.1.2 Architecture

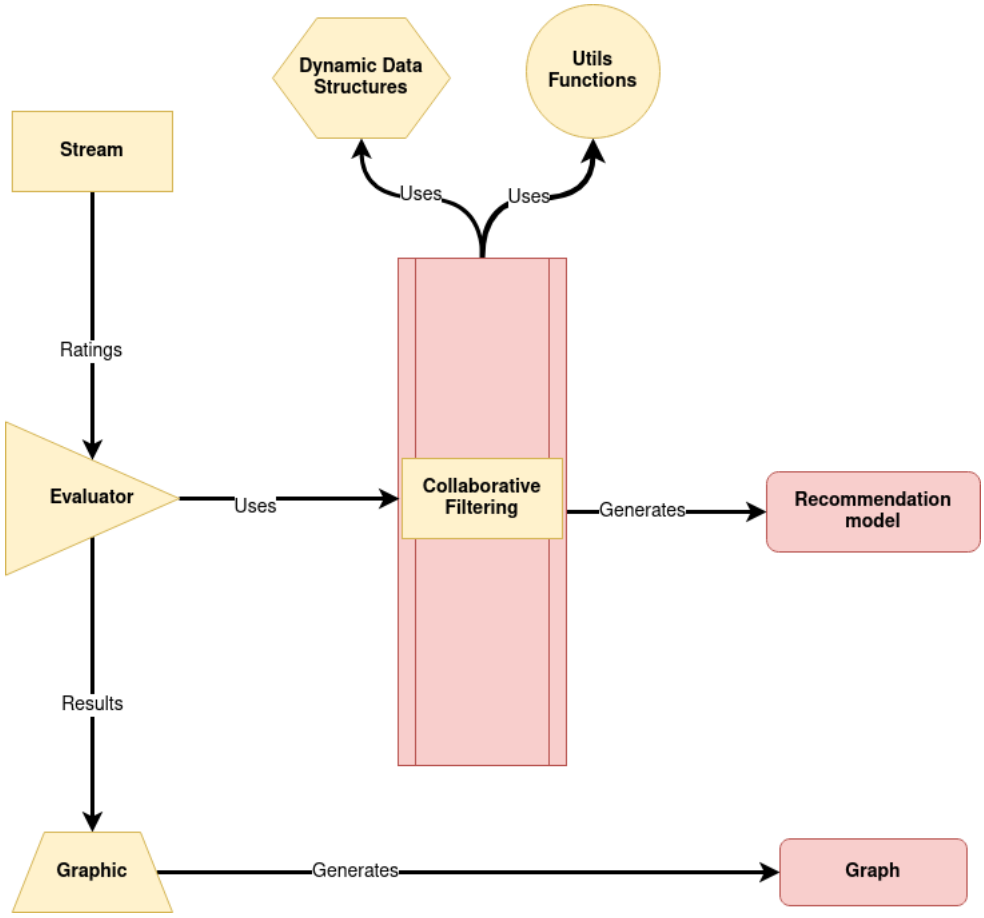

Figure 4.1: Increc's architecture diagram.

As modeled in figure 4.1, *Increc* is designed around 6 main modules: **Stream**, **Evaluator**, **Graphic**, **Algorithms**, **Data Structures** and **Utils**.

- The **Stream** module contains methods for generating data streams, namely through file consumption.
- The **Evaluator** module contains solutions which tackle the measurement problem.
- The **Collaborative Filtering** module contains implementations of several algorithms.
- The **Data Structures** module contains some classes which act as data structures used by the algorithms.

- The **Graphic** module contains strategies related to graph constructions for later analysis.
- The **Utils** module contains utility functions shared across modules.

There are two more modules in the code **Examples** and **Test**. The first one includes working examples of how to use the contents of the library. The second module refers to unit tests required for continuous integration.

4.1.3 Why Python?

When starting *increc* a choice of programming language was needed and so **Python** was chosen. This section aims to explain why.
It is true, that being an interpreted language has its performance disadvantages when compared to other languages such as **C++**, however, *python* has a sort of syntactic sugar which makes it very easy to understand and perform tasks. This is specifically useful for complex machine learning tasks which are in turn easier to implement due to the language's simplicity. Besides this, *python* already has some useful powerful libraries which make current machine learning tasks easier to implement, the most common one being **numpy**. Another advantage being *lingua franca* for data science and machine learning.
Increc was written using **Python3**.

4.1.4 Design patterns in action

Increc was built around the **Composite** design pattern for **OOP**. Since we are dealing with hierarchical relationships between objects, due to the inherent division between algorithm types and the feedback which they are expected to receive which in turn leads to the need to compose objects into a tree structure this design pattern is very useful. Even though *Composite* is meant to be used with implemented *interfaces*, in *Python* we do not have that functionality, so the workaround was to use **multiple inheritance**. In practice, each final algorithm is represented by its own class and extends the parent classes it needs for its supposed behavior. In example, the implicit matrix factorization algorithm which uses stochastic gradient descent extends the implicit matrix factorization class as well as a stochastic gradient descent one. If we wished to implement a matrix factorization algorithm which dealt with implicit feedback and used alternate least squares to train the decomposed matrices we would create a class for ALS and a class for the final algorithm which would extend ALS and the implicit matrix factorization class.

4.2 The algorithms module

This section explains in detail the implementation of *increc's* algorithms as well as its design choices. The implemented algorithms include:

1. Explicit user-based neighborhood

2. Explicit user-based clustering
3. Implicit user-based neighborhood
4. Implicit user-based clustering
5. Implicit item-based neighborhood
6. Implicit item-based clustering
7. Implicit item-based locality-sensitive min-hashing
8. Implicit user-based locality-sensitive min-hashing
9. Implicit matrix factorization
10. Explicit matrix factorization with matrix preprocessing
11. Explicit matrix factorization without matrix preprocessing

Each algorithm is represented by a class with its name and due to similarities between some of them, some superclasses were implemented to avoid code duplication.

The CollaborativeFiltering class

Every collaborative filtering algorithm needs to store a *ratings* matrix and must keep track of the model's users and items. The *ratings* matrix is a **Dynamic Array** which will be explained in section 4.3.1, but it can be understood, for now, as an array of arrays where each array represents a row, which in turn represents a **user vector** and each element of the array represents the values of the ratings of the user to the items represented by the **indexes** of the array. Every column, represents an **item vector**. For example, the *7th* array at the *5th* position holds the value of the rating of the *7th* user to the *5th* item. Since there is the possibility that these algorithms can be used intermittently, *increc* provides the possibility to receive previously computed models, to avoid repeated initial computations. The function **_init_model** checks if the model is empty returning an initial computed model with the callback function provided as a parameter if that is true. Since we are dealing with dynamic environments we must keep track of all the users and items that are required for computations at a given time. This class stores these identifiers in two sets, one for users and another for items, and this is a great help to prevent unnecessary computations of items and users which do not yet exist in the system. This class is extended by all the implemented algorithms and is not supposed to have objects directly instantiating it. Its constructor expects to receive the *user-item* matrix. For standardization purposes, each algorithm is expected to have a **new_rating** method which receives a tuple parameter of the form *(user_id, item_id, value)*. For implicit algorithms, *value* is omitted and assumed to be equal to **1**. Besides this, a **recommend** method is also implemented which expects to receive the user to process recommendations for, the number of items to recommend, and an optional parameter which states if repeated items can

be included or not(defaults to *false*). *Explicit* feedback algorithms also contain a **predict** method which, like the name suggests, predicts a user's rating o an item.

4.2.1 Matrix Factorization algorithms

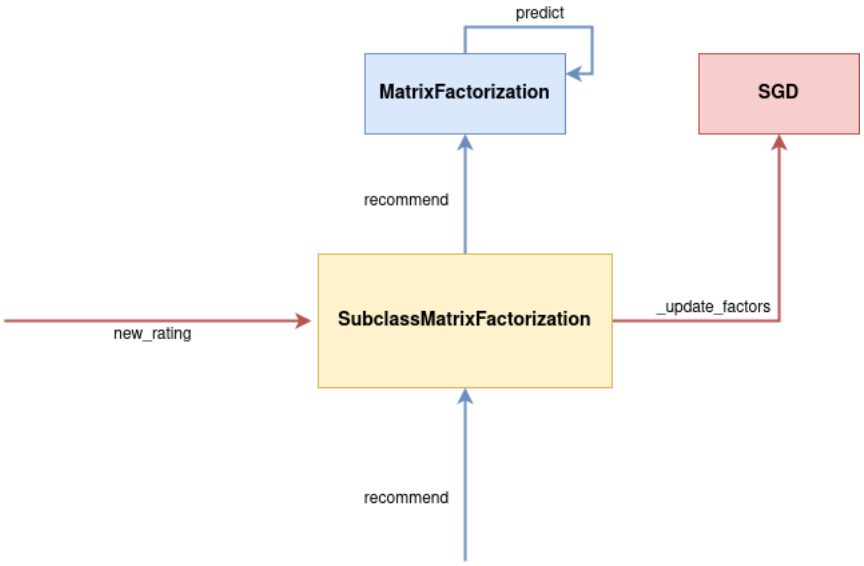

Figure 4.2: Matrix Factorization implementation diagram.

In this section, we will discuss the implementation of matrix factorization methods making a distinction on implicit and explicit feedback and the algorithm through which matrix factorization is obtained. As explained in 4.2, there is a clear separation between the implicit and explicit class and the used algorithm for updating the U and V matrices. For notation purposes, the U matrix is used for the user latent factors and the V matrix is used for the item latent factors.

The MatrixFactorization class

There is a generic class named **MatrixFactorization** which extends *CollaborativeFiltering* and holds code to initialize the decomposed matrices, U and V, through the **_init_u_v**, **_init_u** and **_init_v** functions as well the prediction and recommendations logic, with the **predict** and **recommend** functions respectively. Its constructor expects to receive the standard collaborative filtering parameters as well as possible pre-computed U and V matrices and the number of latent factors to be used on the decomposed matrices. As explained before in section 2.1.2.3, the goal of matrix factorization is to decompose the *ratings* matrix in 2 other matrices reducing the columns and rows high dimensionality to a number of latent factors. This way, the predicted rating of a user i to an item j results in the inner product of the *ith* row of matrix U and the *jth* column of matrix V.
The decomposed matrices are instances of **DynamicArray**, and are initialized with random values in the range $[0, 1]$ before training. After the initialization, the **_initial_training** function takes into account all the ratings in the *user/item* matrix to update the factors.

The **predict** function receives a *user* and an *item*, selects the respective row and column from the U and V matrices and returns the inner product of these two vectors using **numpy's inner** function. The **recommend** function is used by all subclasses and sorts the items according to a heuristic defined in each subclass and is passed along as a function parameter. In *increc* matrix factorization procedures use **SGD** to update the U and V factors.

The SGD class

This class intends to define the way the U and V decomposed matrices are updated, according to the **Stochastic Gradient Descent** algorithm, as explained in section 2.1.2.3 and adapted in section 3.2.1. Its constructor expects to receive a function that returns the default value for new elements of the matrices, which is important because we are dealing with dynamic environments and dynamic data structures, a **learning rate** and a **regularization factor**. The updates are done using the **_update_factors** function which expects to receive a *user*, an *item* and the associated error. This updates the row associated with the user on U, the column associated with the item on V. For this implementation, the versatility of *numpy arrays* was useful since it made vector operations, when updating U and V factors, easier to implement and more efficient during runtime.

4.2.1.1 Implicit Matrix Factorization

The MFImplicitSGD

This class extends **MatrixFactorization** and **SGD**. Its constructor accepts the standard *Matrix-Factorization* parameters. Regarding this implementation, it is worth mentioning the **new_rating** and **recommend** functions. When processing a new rating the error is calculated as $1 - prediction$, which is passed as a parameter to the **_updated_factors** function. The matrix is updated and the user and item will be added to the respective sets. The overloaded **recommend** function uses the parent's **recommend** function as in section 4.2.1 with a heuristic defined as recommending the products whose prediction is closest to 1.

4.2.1.2 Explicit Matrix Factorization

Regarding **Explicit Matrix Factorization**, it is worth mentioning that there are two different implementations. The main difference is the usage of matrix **preprocessing**.

The MFExplicitSGD class

This class extends **MatrixFactorization** and **SGD**. Its constructor accepts the standard *Matrix-Factorization* parameters. The procedure is the same as 4.2.1.1, however the error is calculated using the target rating actual value, as $actual\_value - predicted$. The overloaded **recommend** function uses the parent's **recommend** function as in section 4.2.1 with a heuristic defined as recommending the products whose rating prediction is the highest.

The PreprocessMatrix class

This class extends **MatrixFactorization**. Its constructor accepts the default parameters, plus an array of user and item **average ratings**, as well as a **preprocessed ratings matrix**. The **_init_user_avg** and **_init_item_avg** functions initialize the respective arrays, returning a **DynamicArray** whose elements are the average rating of users and item respectively.
The **_init_preprocessed_matrix** initializes the preprocessed matrix by normalizing ratings through user and item average ratings as referred in section 2.1.2.3, returning a **DynamicArray** with normalized values. The **predict** function makes use of the **_predict_prep** function which just returns the superclass **predict** function results and then removes the previous normalization.

The MFExplicitPrepSGD class

This class extends **PreprocessMatrix** and **SGD**. The main difference is in the **new_rating** which, for every new rating it updates the user and item average ratings, the preprocessed matrix and the U and V factors based on the previous updates.

4.2.2 Neighborhood-Based algorithms

This section will be dedicated to the implementation of neighborhood-based algorithms for online recommendations on *increc*. The general flow of these algorithms is to find neighbors of users and items and to recommend items based on those neighbors for a user. There is a distinction between standard neighborhood approaches and procedures that use element clustering, which helps reduce the complexity of the neighbors calculation. There is also a locality sensitive hashing implementation for both user and item-based methods.

4.2.2.1 Standard neihborhood approach

The standard neighborhood approach in *increc* computes the entire neighborhood at every iteration.

The NeighborhoodCF class

Since all of the standard neighborhood-based collaborative filtering algorithms implemented in *increc* have this common ground it made sense to create a common neighborhood class, named **NeighborhoodCF** which deals with neighborhood-related logic. Its constructor accepts the number of neighbors, as well the neighborhood of each element whether it is a user or an item. The **_init_neighborhood** function calculates the neighborhood for each element (user or item) and receives the candidates as a parameter. The returned neighborhood is a **DynamicArray** object where each element is the return value of the **_neighborhood** function for each element. This function returns the **k-nearest-neighbors**. In *increc*, *knn* is implemented in the **knn** function in the **utils** module, using the *Python* built-in **sorted** function. *Python's sorted* function uses a strategy implemented by Tim Peters in 2002 [7] and has a complexity of **O(n log(n))** for average and

worst cases. *Knn* receives the candidates, a function which is a heuristic to sort elements by and the number of elements to return.

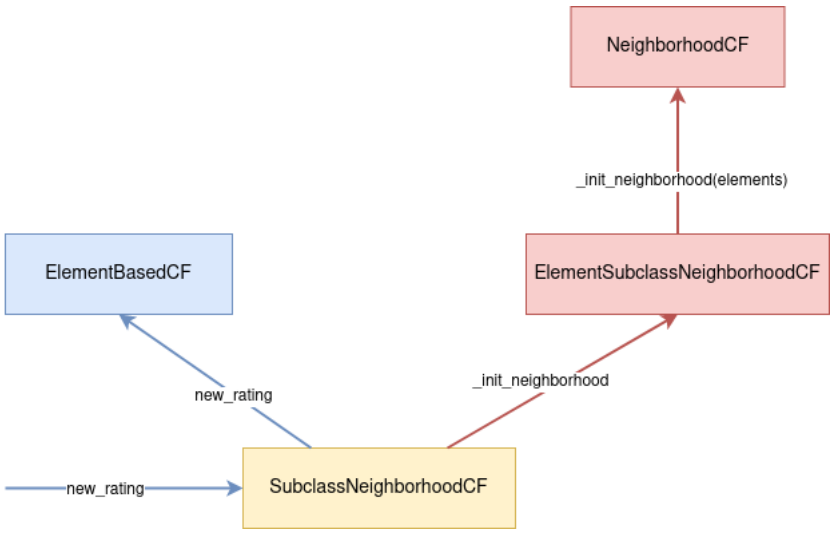

Figure 4.3: Neighborhood collaborative filtering implementation diagram.

There is a clear distinction between *user-based* and *item-based* methods which are divided between two different classes: **UserNeighborhood** and **ItemNeighborhood**. Their goal is just to specify which element set are neighbors going to be computed for, the item set fot *ItemNeighborhood* and the user set for *UserNeighborhood*. All these classes are meant to only concern themselves with neighborhood-related logic. There is also a distinction, seen in figure 4.3, between classes which calculate element-wise similarities and classes which compute neighbors using those similarities.

4.2.2.2 Clustering approach

The clustering approach incrementally separates the element sets into clusters in order to reduce the neighborhood search only for the clusters where each element is mapped to. The incremental approach follows the work of [16].

The Clustering class

This class is implements an incremental version of **k-means clustering** as described in section 10. It extends the **NeighborhoodCF** class and besides the default parameters it also expects to receive a **similarity threshold**, the **clusters**, the **centroids** and a **cluster map**. The **_init_centroids** function generates a random centroid from the element set (which can be the users or the items), returning a list of size 1, containing the initial centroid. The **_init_clusters** function returns a list of sets, which are the clusters. Finally, **_init_cluster_map** returns a *Python's* dictionary which maps every element to the cluster they are currently in. The neighborhood is calculated for every cluster, which is done by the **_init_neighborhood** function, which iterates through each cluster

calling the **_init_neighborhood_cluster** function to each cluster. This last function returns a **DynamicArray** whose elements are the neighbors of each element in the cluster. The clusters and neighbors are updated through the **increment** function. This function receives an identifier(user or item), and checks the similarity between that element and all the centroids. It then maps the element to the *closest* centroid, that is, the centroid whose similarity between the element provided as a parameter is the highest. If this similarity is not higher than the threshold, then this element becomes a new centroid.

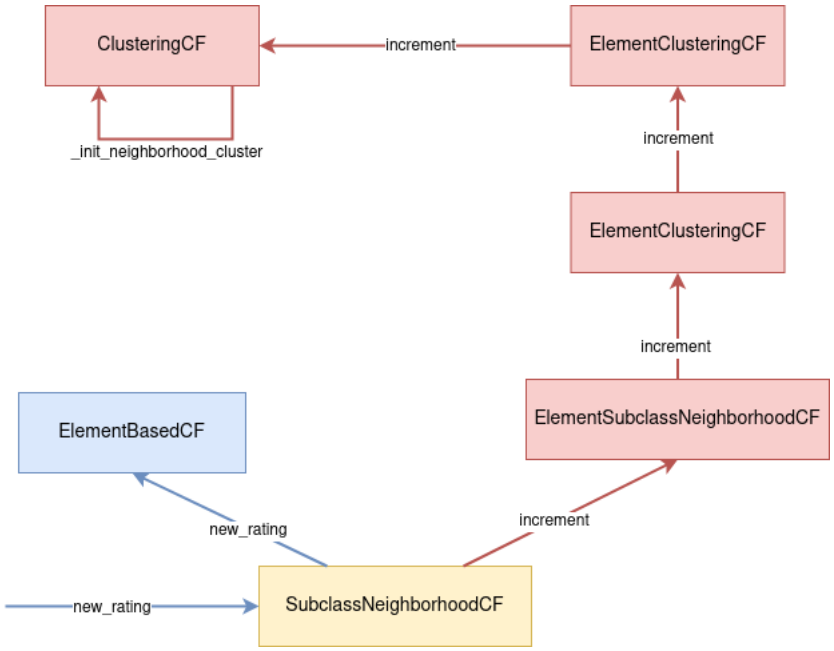

Figure 4.4: Clustering collaborative filtering implementation diagram.

ALGORITHM 10
Incremental clustering

$centroid \leftarrow maxSimilarity(centroids, element)$
if $similarity(centroid, element) < threshold$ **then**
  $clusters[centroid].add(element)$
  $clusterMap[element] \leftarrow centroid$
else
  $centroids.add(element)$
end if

Like before, there is a clear separation between item and user logic, with the **ItemClustering** and **UserClustering** classes.

4.2.2.3 Locality-sensitive hashing

This section is dedicated to the implementation of locality-sensitive min hashing for users and items.

The LSHMinHash class

The **LSHMinHash** which extends *CollaborativeFiltering* and is responsible for dealing with common logic of the locality-sensitive min-hashing algorithm for both users and items. Its constructor accepts the standard collaborative filtering algorithms as well as a **signature matrix**, **buckets**, **number of permutations** and **number of bands**. It follows the same heuristic explained in section 2.1.1.4 but with some differences from [8], since the authors of the paper follow a different family of hashing functions for cosine similarity to describe their LSH scheme. The hashing function family is defined through the **MinHashing** technique. This technique is used to estimate the *Jaccard Similarity* [45] and consists on returning the first not-null element in a vector. In the **_calculate_signatures** function the *ratings matrix* is shuffled according to the number of permutations parameter, and for each permutation the **_generate_signature** function iterates over each element calling the **_min_hash** function which returns the index of the first user which rated that item (for items), or the index of the first item which a user has rated (for users) making it the resulting hash. Note that when we permute a matrix, for items we just shuffle the order of the user vectors, but for users we shuffle the order of the items for every user vector. The permutations are done by using the **permutation** function from the *numpy* library and every subclass defines how are they processed. In this implementation, the signature matrix, represents the hashing of each element(user or item), where each column represents an element and each row has the hashing value for a specific permutation defined by a hashing function. It is an object of the **DynamicArray** class. The next part of the algorithm is related to the number of bands parameter. Each *element-hash* vector is grouped by the number of bands provided, and each resulting **band-group** forms a bucket. Each element possessing that *band-group* is mapped to the same bucket. The grouping is done by the **_group_by_bands** function. We can see that as we increase the number of permutations, we will increase the runtime of the algorithm and the length of the hashing vectors, which will increase the probability of matching items. If we increase the number of bands we will get greater quality in the elements that are mapped to the same bucket decreasing the number of **false-positives**, which are items who are mapped to the same bucket but are not similar to each other. However, if we increase this parameter too much, the number of **false-negatives** will also increase, which means similar elements will not be mapped to the same bucket. When the **new_rating** function is called the *ratings* matrix is updated and the item and user are added to their respective sets. We update the signature matrix on the column of the element provided in the argument, and then recalculate the buckets for that element. This whole process is explained in figure 4.5.

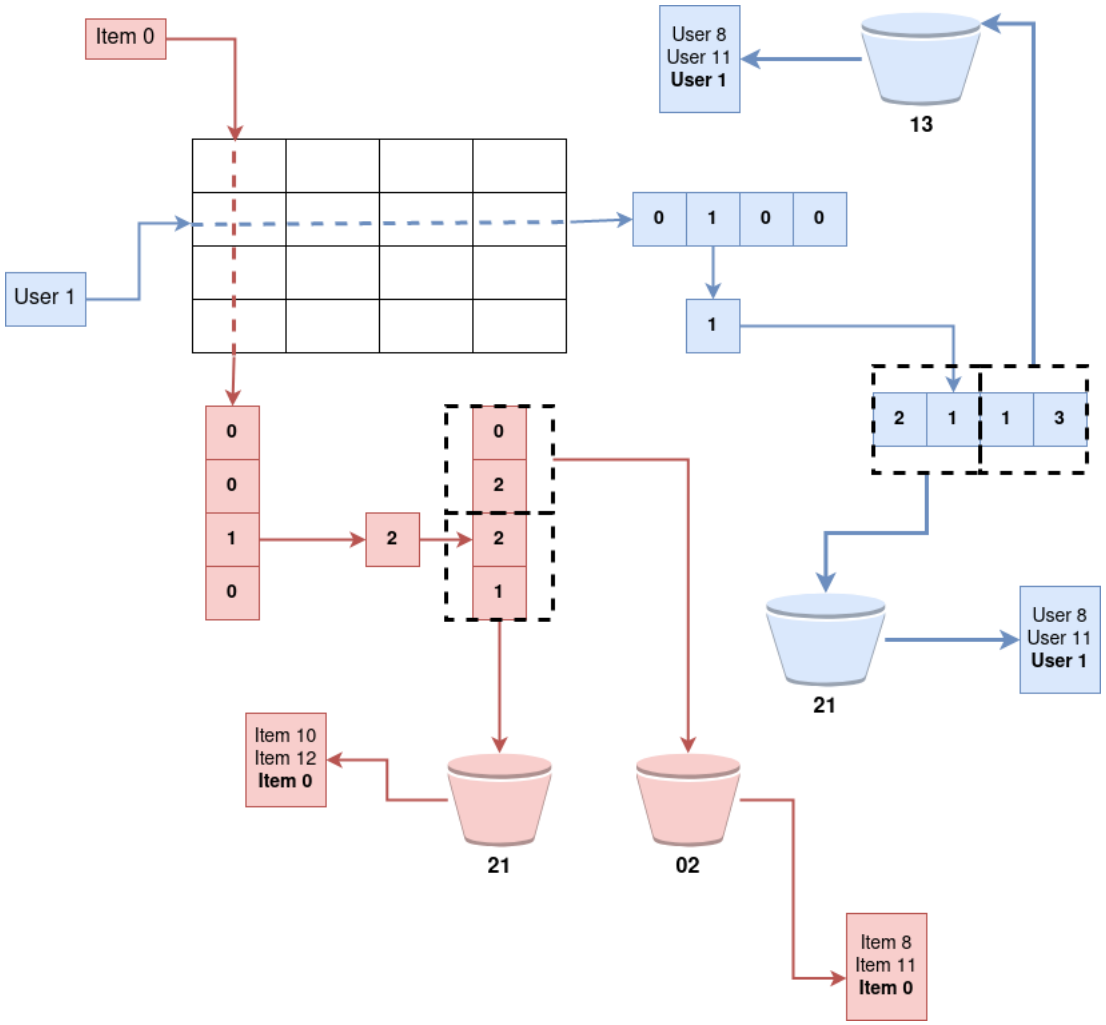

Figure 4.5: Locality-sensitive min-hashing draft in Increc.

If we consider m for the length of each element vector (user or item), and p as the number of permutations, the signature matrix computation has $O(m \times p)$ complexity. Taking n as the number of elements (users or items) and b as the number of bands, the bucket group computation has $O(n \times b)$ complexity.

4.2.2.4 User and Item approaches

In section 4.2.2, we described how neighbors are being calculated, however we have not talked about how similarities are being calculated. We should take a step further and discuss the implementation of how similarities between pairs of elements are calculated which then are used to perform neighborhood computations. In this section, we will address the implementation of the user-based collaborative filtering methods, for explicit and implicit feedback as well as an implicit item-based algorithm. Lets start with the user-based approach.

The UserBasedCF class

Both algorithms share a common superclass named **UserBasedCF** which deals with shared logic regarding similarities initialization and **co-rated** items. This class extends **CollaborativeFiltering** and its constructor expects to receive the *user/item* matrix, and a matrix of **co-rated items**. The **_init_co_rated** function returns a *co-rated* matrix by computing co-rated items for each pair of users. The *co-rated* model is stored inside a **Symmetric Matrix** object and represents the identifiers of items which have been rated (implicitly or explicitly) for each pair of users. To avoid duplicate memory usage, the *Symmetric Matrix* class was used since the*co-rated* items of the user pair $(user_n, user_m)$ are the same as the user pair $(user_m, user_n)$. To update the co-rated items matrix, the **_update_co_rated** function is used, which accepts a user, a item and a comparison function as a parameter. The **_init_similarities** computes a similarity matrix. This class calls the **_init_similarity** function for every pair of users, which defines the computation of the similarity for every subclass and, therefore must be written and declared in each subclass of *UserBasedCF*. Similarities are also stored inside a **Symmetric Matrix** for the exact same reason, however, the explicit and implicit user-based have different ways of calculating the similarity between two users, so we will dig deeper on this further.

The UserBasedExplicitCF class

The *explicit* version was implemented following the work of [27], mentioned in section 3.1.2 but not taking into consideration rating updates, which means cases where a user updates a rating of an item previously rated by that user. This decision was made because, during implementation and research, the formulas proved to be doubtful and lead to wrong calculations and computations (see A). It extends **UserBasedCF** and expects to receive its parents' parameters as well as a **similarities matrix** and the users' **average ratings**. The **_init_avg_ratings** function computes the users' average ratings which is a **DynamicArray** objects whose elements' indexes coincide with the user id's and each position has the respective user average rating. Like previously mentioned in section 4.2.2.4, this class needs to implement the **_init_similarity** function. In *UserBasedExplicitCF*, this function computes pairwise similarities using the **pearson_correlation_terms** function that is located on the *utils* module, and returns 4 terms: the **covariance** between the 2 variables, the **variance** of the 2 variances, and the actual similarity pearson correlation value. The actual similarity makes use of the **pearson_correlation** function also defined in the *utils* module. These terms include the covariance of the ratings of both users, and the variances of the ratings of each user to the co-rated items. Since the variance depends exclusively on the items *co-rated* by each pair of items we cannot store variance per user, and must store per pair of users, two computed variances, one per each user. The variance of a user will differ from the pair of users we are analyzing because *co-rated* items are different. Because we increment these terms at every iteration, instead of calculating them all over again they are stored in a dictionary for each pair of users. Due to the inherent symmetry of this issue, also correlated to the similarity matrix, we will only store these terms one time per pair of users. This causes an issue if we want to know the variance of the

ratings of a certain user for a specific user pair. To solve this issue the variances were stored inside a class which deals with this problem. This class is named **PairVariances** and will be explained here since its use case is only adequate for this algorithm. *PairVariances* checks the identifiers of the users in question and returns the variance depending on their absolute values. It receives two variances, the first and the second, the first variance corresponds to the user whose identifier has a bigger absolute value in the pair. The **new_rating** function, updates the similarities, using the **_new_rating** function which accepts a user, an item and a value, and calculates the term increments for each pair of user where the user is present, using the **_new_rating_terms** function. The **predict** function, such as the names suggests, calculates a possible rating of a user to an item, by verifying the user neighborhood and returning the average of the ratings of the users in the neighborhood for that item. This method is used on the **recommend** function, while sorting the n items with the highest predicted rating.

The UserBasedImplicitCF class

The *implicit* version was implemented following the work in [25] and located inside the **UserBasedImplicitCF** class. This class extends **UserBasedCF**, and besides the default parameters, it also receives a **similarities matrix**. As described in section 4.2.2.4, the **_init_similarity** function is defined to return a pairwise similarity between two users. Similarities are calculated using the **cosine_similarity** function located in the *utils* module. This function receives the number of *co-rated* items between the two users and the total number of items. This implementation is much simpler, since it only depends on incrementing the number of co-rated items at every iteration. The **_update_similarities** expects to receive a user, and updates every pairwise similarity on which that user is included. Every time the **new_rating** function is called, the user and item are added to the respective storage sets and the ratings matrix is updated. The *co-rated* matrix is also updated by checking all the other users who rated the specific item and the similarities between that user and all the other ones are recalculated. After this, the neighborhood of all users is also recalculated. The **recommend** function checks items which the user has rated, and then recommends the ones with an higher **activation weight**. This measure is calculated with the **_activation_weight** function.

The ItemBasedImplicitCF class

The approach used to implement the item-based neighborhood collaborative filtering is similar to the **Dynamic Index** explained in [20]. The big difference is related to the usage of **recommendable** items which the authors take into account in their original paper. *Increc* does not take into account the notion of *recommendable* or *non-recommendable* items because of generalization purposes. Of course, it helps to limit the number of computations necessary for item similarities and neighborhoods, however this is highly related to business logic and so, it is a matter of responsibility of the people using *increc* has a dependency. The **ItemBasedImplicitCF** class extends

CollaborativeFiltering, accepting the default parameters as well as a **matrix of item intersections**, an array of **items' l1 norms** and a dictionary which is the **inverted index** of users to items. The **_init_intersections**, computes the intersections matrix which holds the number of users who co-rated each pair of items. The *item intersections* is a *SymmetricMatrix* because the number of users who have rated the tuple, $(item_a, item_b)$ is the same as the number of users who rated $(item_b, item_a)$. Intersections are updated using the **_update_intersections** function. The items *l1-norms* array, is a **DynamicArray** and computed by the **_init_l1** function. Each index of the array is the same as the item identifier, and its position stores the corresponding *Manhattan Distance* of the item vector. The *inverted index* is a *Python* **defaultdict**, created by the **_init_inv_index** function, which maps a set of items for each user identifier, having an empty set by default. Finally, *item-pair* similarities are, as expected, stored inside a *SymmetricMatrix* object, for the same reasons previously explained. Similarities are calculated using the **cosine_similarity** function from the **utils** module, which receives the number of item intersections and the *squared-roots* of each one of the items' *L1-norms* of the pair. Note that, the number of intersections is the same as calculating the dot product between the two item vectors, and the *squared-root* of the items *l1-norms* is the same as calculating a **l2-norm** (**euclidean distance**) because item vectors are made up of binary values. The **_init_similarity** function computes the similarity between a pair of items, and the **_update_similarities** uses the previous function to compute the similarities between a user provided in the arguments and all the others. The **recommend** function checks the neighborhood of all the items which the user provided in the arguments has rated and then shuffles that list.

4.2.2.5 The actual neighborhood algorithm classes

Previously, we described the classes that implemented neighborhood search and similarity computation. However these classes are not supposed to be instantiated and this section is dedicated to the classes which extend the previous heuristics and actual implement the algorithms.

The explicit UserBasedNeighborhood class

This class implements the classic explicit user-based neighborhood and extends **UserBasedExplicitCF** and **UserNeighborhood**.
It only implements the **new_rating** function which calls **UserBasedExplicitCF's** *new_rating* to compute similarities and then recalculates the entire neighborhood.

The explicit UserBasedClustering class

This class implements a clustering version of the explicit user-based neighborhood and extends **UserBasedExplicitCF** and **UserClustering**. It only implements the **new_rating** function which calls **UserBasedExplicitCF's** *new_rating* to compute similarities and then increments the clusters, calculating the neighborhood according to the cluster where the user in the rating is present.

The implicit UserBasedNeighborhood class

This class implements the classic implicit user-based neighborhood and extends **UserBasedImplicitCF** and **UserNeighborhood**.
It only implements the **new_rating** function which calls **UserBasedImplicitCF's** *new_rating* to compute similarities and then recalculates the entire neighborhood.

The implicit UserBasedClustering class

This class implements a clustering version of the implicit user-based neighborhood and extends **UserBasedImplicitCF** and **UserClustering**. It only implements the **new_rating** function which calls **UserBasedImplicitCF's** *new_rating* to compute similarities and then increments the clusters, calculating the neighborhood according to the cluster where the user in the rating is present.

The UserLSHMinHash class

This class implements the user-based locality-sensitive min hashing algorithm.
It extends the **LSHMinHash** class. It implements a **new_rating** function which calls *LSHMinHash's new_rating* and updates the signature matrix and the buckets according the user identifier. The **get_vector** function returns a row from the given matrix (since by default users are stored on rows), and the **_elements** function returns the user set. The **recommend** function groups the given user by bands, and for each band we check the users in that respective bucket which are sorted according to the number of times they come up. Then we list the items the top *n* (where n comes in the arguments) users have rated, shuffle that list, and return the first *n* elements.

The ItemBasedNeighborhood class

This class implements implements the classic implicit item-based neighborhood and extends **ItemBasedImplicitCF** and **ItemNeighborhood**.
It only implements the **new_rating** function which calls **ItemBasedImplicitCF's** *new_rating* to compute similarities and then recalculates the entire neighborhood.

The implicit ItemBasedClustering class

This class implements a clustering version of the implicit item-based neighborhood and extends **ItemBasedImplicitCF** and **ItemClustering**. It only implements the **new_rating** function which calls **ItemBasedImplicitCF's** *new_rating* to compute similarities and then increments the clusters, calculating the neighborhood according to the cluster where the item in the rating is present.

The ItemLSHMinHash class

This class implements the item-based locality-sensitive min hashing algorithm.
It extends the **LSHMinHash** class. It implements a **new_rating** function which calls *LSHMinHash's new_rating* and updates the signature matrix and the buckets according the item identifier.

The **get_vector** function returns a column from the given matrix (since by default items are stored on columns), and the **_elements** function returns the item set. The **recommend** function checks the rated items by the user provided in the arguments, and retrieves their signatures from the signature matrix. Then it groups every signature by the number of bands and for each signature and each band we retrieve the items in the **band-bucket** sorting the items for most appearances. After this we return the top n items.

4.3 The data structures module

This section is dedicated to the description of the implemented data structures in *increc*, challenges they were meant to solve and how they solved those same challenges.

4.3.1 Dynamic Array

The **DynamicArray** class aimed to create a **dynamic** data structure which would help simplify code for incoming new ratings. This is related to the nature of the user and item identifiers. In live recommendation systems, data is only timely ordered (normally through) a timestamp. This means that there is no guarantee that a rating for a $user_{10}$ will come before $user_{20}$ just because $10 < 20$. The same applies for items. So we have a problem of dynamically allocating memory to hold data for unknown users or items. Plus, since algorithm classes store a lot of information for each user or item, it would not be feasible to check if the new rating's user and item identifiers were greater than the current highest ones for every manipulation that needed to be done. That would be chaotic to implement. To solve this issue, a dynamic data structure was created so that the algorithm classes could just access the indexes they pleased, without having to worry about identifiers or index errors. Before getting further into more details, we will take a look at a specific example. Let us imagine we are storing a *user-item* matrix, with a current state of 3 users for 3 items, as in figure 4.6.

$$\begin{bmatrix} & item_1 & item_2 & item_3 \\ user_1 & 1 & ? & ? \\ user_2 & 3 & ? & 3 \\ user_3 & ? & ? & ? \end{bmatrix}$$

Figure 4.6: A user/item matrix example.

Let us now consider we receive a new rating: $(u_6, i_4, 5)$. In *increc*, this matrix is represented through the usage of **DynamicArray** objects, where each row is an object of that class. In this specific example two more *DynamicArray* objects are created with empty ratings for the items since they had not appear in the system beforehand. A final user is created, and this *DynamicArray* object holds 5 items, on which only the rating for the 5th item is known. This process is represented in figure 4.7. As we can see, not all the rows of the matrix have the same length, which is not problematic.

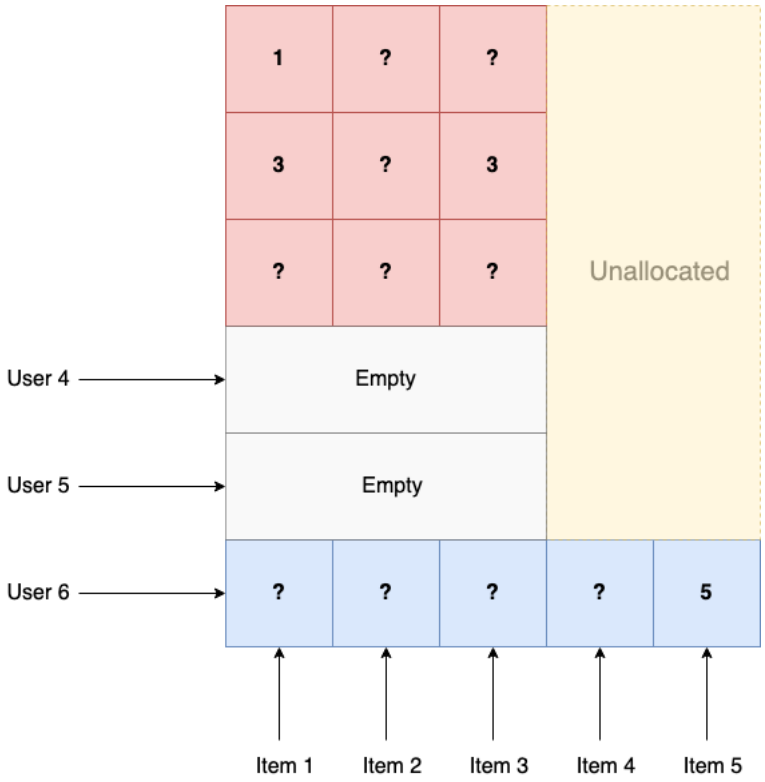

Figure 4.7: Matrix Update.

This means that, when we want to access an index, the *DynamicArray* class first checks whether it is greater than the current size, and if it is it adds new elements until it matches the specified index. This means that lookup operations will nots have necessarily **O(1)** complexity. This class is implemented on top of *Python's* lists. The option for lists instead of *numpy arrays* was made due to the fact that *numpy* is optimized for homogeneous arrays, which does not match our use case [26]. *DynamicArray* overrides *Python's* **__iter__**, **__len__**, **__getitem__** and **__setitem__** to simulate a normal list behavior and implements an **extend** method for dynamicist purposes. Due to column operations needed in section 4.2.2.3, a **set_col** and **col** methods were implemented for setting a column and retrieving a column respectively.

4.3.2 Symmetric Matrix

The **SymmetricMatrix** class is responsible for storing parts of the algorithms' models which can be fit inside a matrix with symmetric characteristics. This class was needed to make a more efficient use of memory, taking advantage of the fact that there was no need to store information for the same tuple of elements twice.

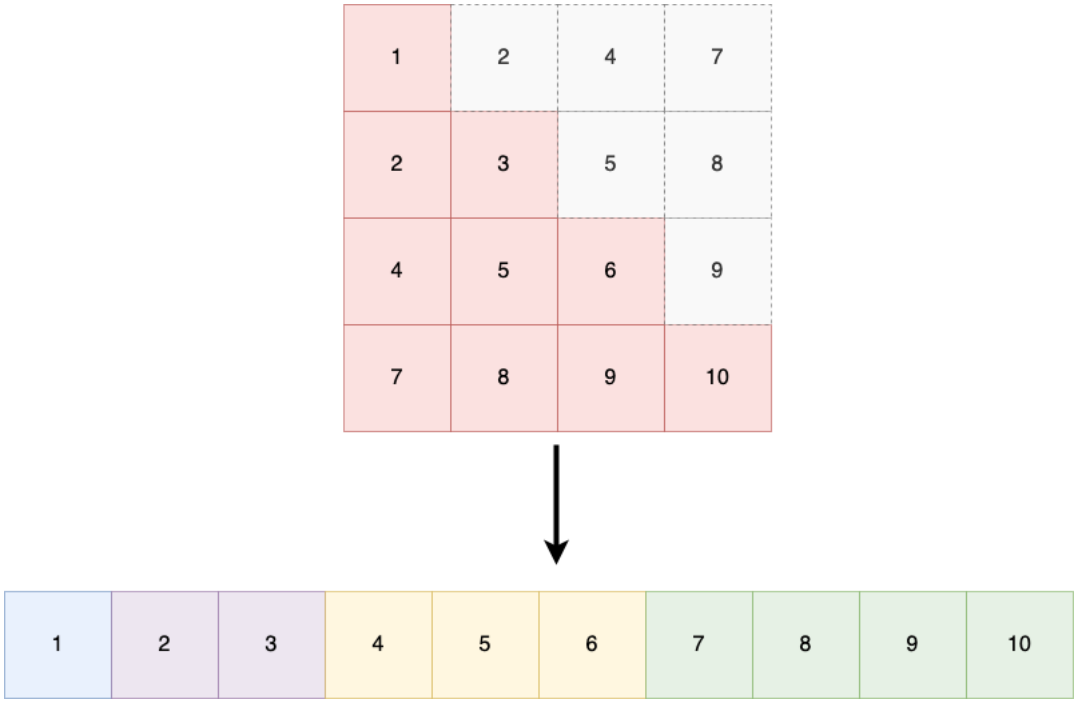

Figure 4.8: Symmetric Matrix.

This class is built on top of *DynamicArray* due to the need of dynamically scaling models when updates need to be done. The **SymmetricMatrix** is represented by a *DynamicArray* object not following the traditional implementation of a matrix. Its implementation follows the approach present in figure 4.8. It is a flat, 1 dimensional object, which follows the following computation to get *row-column* pairs:

$$index = row \times \frac{row+1}{2} + column \tag{4.1}$$

This means that, the position *(3,2)* will be stored on the $3 \times \frac{3+1}{2} + 2 = 8th$ index. This class accomplishes this by overriding the **__iter__**, **__setitem__** and **__getitem__** methods. When accesses to an index are done, the row and column provided in the arguments are switched if the column is greater than the row, since we are looking at the left side of the matrix.

4.4 The prequential evaluator module

As discussed in 3.3, and similarly to what is present in [11], *increc* supports the existence of **Prequential Evaluators** and makes a distinction between *implicit* and *explicit* feedback. Besides measuring error rates, it also measures how much time it takes for a model to process ratings and recommendations as data grows with time. The **PrequentialEvaluator** class holds logic common to both subclasses which deal with their type of data. This class' constructor expects to receive an **algorithm class**, a **window size**, and the **number of recommendations** to be done. Every time a new rating is about to be processed, the prequential evaluator executes code which is subclass dependent, which in turn generates an error. This superclass adds the error to a window array and when the window limit is reached, it calculates the **average error** of the window. If no window is specified, the error is calculated at each iteration, making it a global average error. This process can be viewed in 4.9.

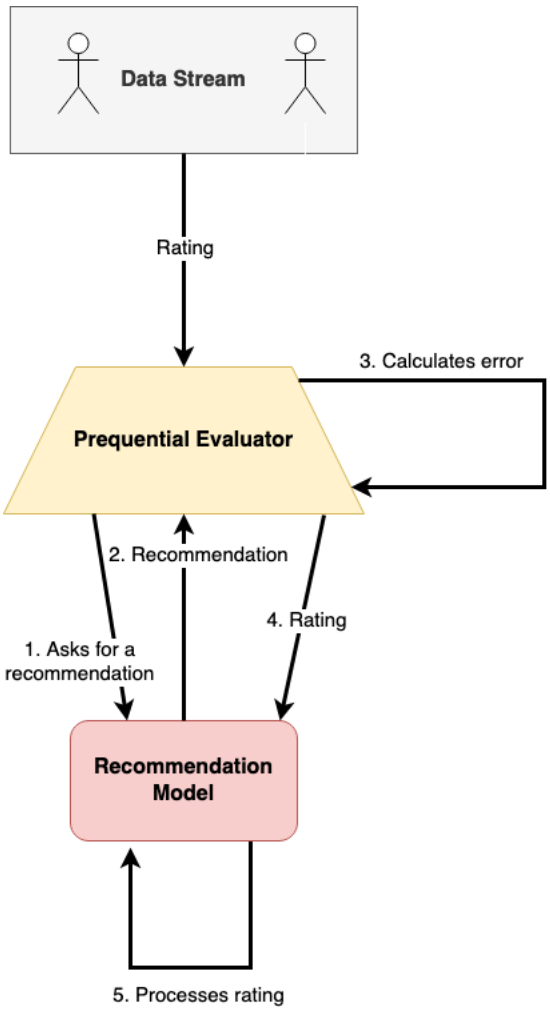

Figure 4.9: Prequential Evaluator.

4.4.1 Implicit Prequential Evaluator

This evaluator deals with *implicit/nominal* type of feedback. Its implementation is located inside the **PrequentialEvaluatorImplicit** class. For every incoming rating, of the type (u_a, i_b) where a and b represent the user and item identifiers respectively, the evaluator first asks the recommendation model for a recommendation of n items for user a. If item b is in the recommended items, it returns a **0**, else it returns a **1** which is added to the evaluators window data. This is similar to an **hit-ratio** approach where we calculate the number of times an item was predicted to be recommended before it appeared as a user preference. This measure was implemented instead of the *hit-ratio* one so that for both types of feedback, *increc* could compute an **error** measure for both types of feedback, instead of having an *error* measure for one and **success** measure for another, which in turn makes latter visualization more comprehensible. Having said this, in *increc* we calculate the error at iteration k, with a tuple $(user, item)_k$ as:

ALGORITHM 11
Implicit prequential evaluation - Error calculation

$recommendations \leftarrow recommendation(user_k)$
if $item \in recommendations$ **then**
  $error \leftarrow 0$
else
  $error \leftarrow 1$
end if
$window\_counter \leftarrow window\_counter + 1$
$window\_error.append(error_k)$
if $window\_counter > window\_size$ **then**
  $window\_average\_error = \frac{sum(window\_error)}{window\_size}$
end if

4.4.2 Explicit Prequential Evaluator

It does not make sense to calculate *hit ratios* for explicit type of data feedback. Since, with this type of data, represented as $(u_a, i_b, value)$, we have access to the absolute preference of user a to item b in the form of *value* we can compute the difference between a predicted value user a to item b, and the real one: *value*. This way, since every explicit recommendation algorithm needs to have a *predict* method, at iteration k with a tuple $(user, item, value)_k$ we calculate the error as:

ALGORITHM 12
Implicit prequential evaluation - Error calculation

$prediction \leftarrow predict(user, item)$
$error \leftarrow prediction - value$
$window\_error.append(error)$
$window\_counter \leftarrow window\_counter + 1$
if $window\_counter > window\_size$ **then**
  $window\_average\_error = \frac{sum(window\_error)}{window\_size}$
end if

4.5 The file stream module

This module is located inside the **stream** module and is aimed at building a ratings data stream out of a dataset file. This stream is later used for recommendation algorithms.

The FileStream class

This class implements generic logic for both implicit and explicit feedback datasets. It expects to receive the path to the file and a column-wise separator which defaults to the *TAB* character. A file is supposed to have a rating per line, and each line is supposed to have a user, an item, a rating value (for explicit feedback) and a timestamp, each separated by a separator character. The constructor creates a stream with the **_parse_file** function which receives a path and a separator character and iterates over each line forming a rating tuple with the **_parse_rating** (defined in each subclass) and returning a list of rating tuples, as portrayed in 4.10.

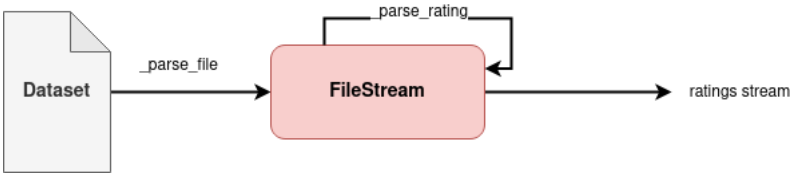

Figure 4.10: File stream.

4.5.1 Implicit file stream

The FileStreamImplicit class

This class extends **FileStream** and expects to receive the same arguments. It implements the **_parse_rating** returning the first two elements of a line in the file which are supposed to be the user and item, in the form of an *int*.

4.5.2 Explicit file stream

The FileStreamExplicit class

This class extends **FileStream** and expects to receive the same arguments. It implements the **_parse_rating** returning the first three elements of a line in the file which are supposed to be the user, item and rating value in the form of an *int* for the user/item identifiers and float for the latest.

4.6 The graphic module

Increc supports a feature of graphic visualization through the implementation of the **EvaluationStatic** class. Before digging further, it is worth mentioning that part of the library uses **matplotlib** as a dependency. *Matplotlib* is a software library written in *Python* which offers support for creating static, animated and interactive data visualizations. *EvaluationStatic's* goal is to provide a visualization of the variation of the error rate and time execution of a recommendation model as data grows. Its constructor accepts a **data stream** and a **PrequentialEvaluator** object, and it constructs 3 *matplotlib* plots: **error ratio**, **time of rating processing** and **time it takes to recommend items**. *EvaluationStatic* uses the **process** function to process a data stream. As we can see from 4.11, it starts by evaluating the whole data stream with the **evaluate** function, which sets up a progress bar from **Pypi progress** package. At the end, it plots the graph with the **plot** function and then renders it with the **show** function. This class is useful to compare algorithms' performances between each other and with different datasets.

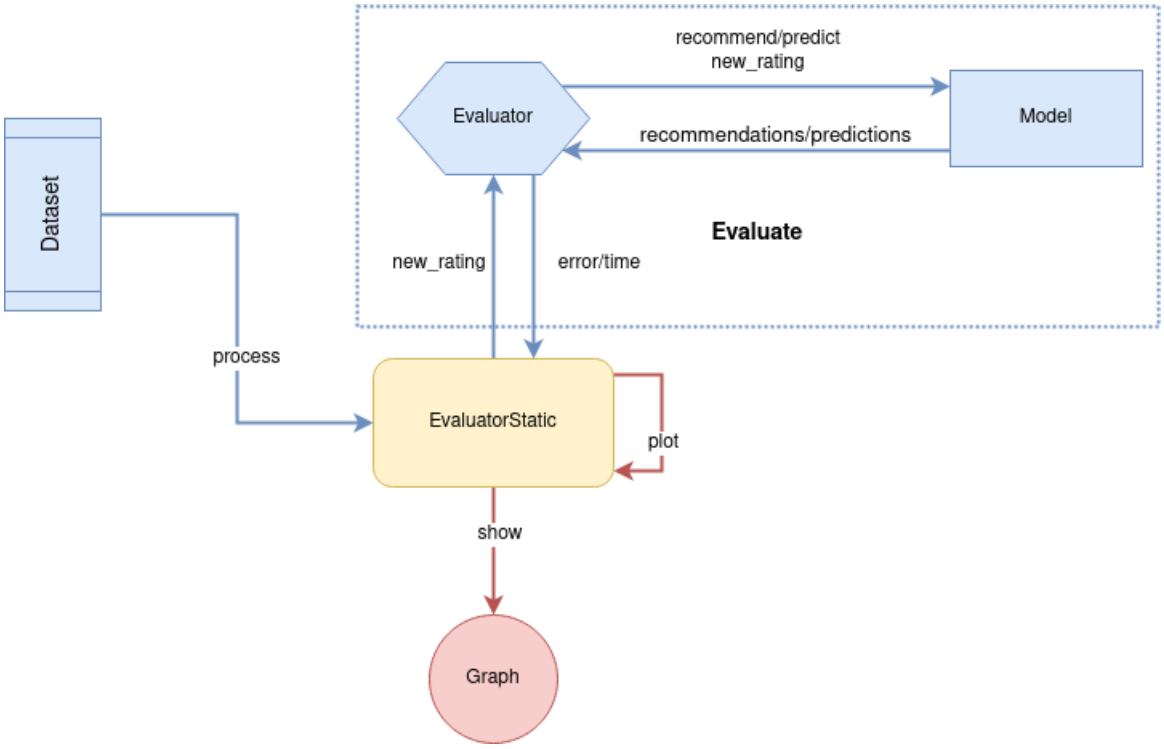

Figure 4.11: Graphical evaluation diagram.

4.7 Continuous integration and deployment

Increc is **open-sourced** at https://github.com/Marko50/FEUP-DISS, under the **MIT** license which provides the ability to be commercially used, distributed, modified and privately used without any kind of liability or warranty. It's packaged using *Python's setuptools*, under **Travis** continuous integration platform which deploys the library automatically to the *Python's* package index: **PyPi**. *TravisCI* also runs all the linting and unit test verifications. It also uses **Azure Pipelines** to keep track of code coverage using **codecov.io** which is integrated with **Github**. Documentation is done using the **Sphinx** package and is present at http://www.andrefernandes.me/increc-documentation/. Figure 4.12, located below, sums up the whole deployment process.

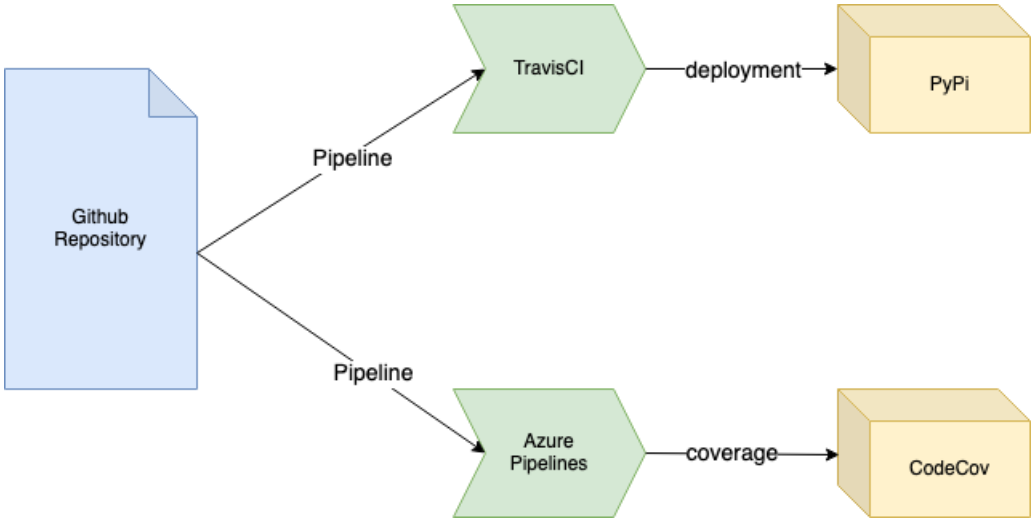

Figure 4.12: Increc's deployment diagram.

Chapter 5

Results

This chapter aims to provide information about *increc's* results with useful chart visualizations and discussion about them.

5.1 Algorithm Tests

This section is dedicated to the output of *increc's* algorithms. It is meant to act as a confirmation of the accuracy and efficiency of the developed code and that the algorithms are functional. Before showing the graphics it is worth mentioning how the testing was done and why. First of all, all the graphs presented in this section are produced by *increc's* **graphic** module using the **Evaluation-Static** class explained in section 4.6 combined with the **PrequentialEvaluator** class explained in section 4.4. There are 2 tested metrics: **Time efficiency**, how long do algorithms take to run and **accuracy**, how good are they at recommending or predicting ratings. They are spread across 3 different charts:

1. **Average error** - Average recommendation error using a sliding window.

2. **Rating process time** - The amount of time it takes to process a new rating, in **seconds**.

3. **Recommendation time** - The amount of time it takes to recommend products to a user, in **seconds**.

The error was calculated for implicit and explicit feedback as explained in sections 4.4.1 and 4.4.2 respectively. With this being said we can refer to the error calculation as the **average error** of the window applied to the prequential evaluator. This means that for implicit feedback we calculate the error for a window as :

$$window\_average\_error = \frac{number\_of\_unsuccessful\_recommendations}{window\_size} \tag{5.1}$$

Similarly, for explicit feedback, we sum the error inside the window:

$$window\_average\_error = \frac{sum\_of\_errors\_in\_window}{window\_size} \tag{5.2}$$

Regarding test data, 3 different datasets were used:

1. **MovieLens (100k)** - 100.000 ratings on a 1-5 rating scale from 943 users on 1682 movies. Used in its entirety.

2. **MovieLens (1M)** - 1.000.209 ratings on a 1-5 of 3.900 movies made by 6.040 users. Used in its entirety for implicit matrix factorization, implicit clustering and locality-sensitive hashing approaches, and and implicit neighborhood methods the first 50000 implicit ratings were used. The first half was used for explicit matrix factorization approaches, the first 250.000 ratings were used for explicit user-based clustering and the first 100.000 were used for the equivalent user-based neighborhood algorithm.

3. **MovieLens (10M)** - 10.000.054 ratings on 10.681 movies by 71.567 users of the online movie recommender service MovieLens. This dataset was only used for implicit matrix factorization with the first 1.000.000 implicit ratings.

4. **INESC (Playlisted tracks)** - 111.942 implicit ratings. Used in its entirety only for implicit matrix factorization. This dataset is taken from the portuguese music social network *"Palco Principal"*, that gathers non-mainstream musicians with fans. The website allows free music streaming and users can organize their favorite music tracks in personal playlists. The dataset consists of a music track playlisting log (users adding music tracks to their personal playlist).

The *MovieLens* datasets are broadly used in education and research and are the result of user interactions with the *MovieLens* online recommender system [18]. All these datasets have been timely ordered before being fed to the algorithms. For *implicit feedback* algorithms, the datasets were parsed only to include favorable ratings, that is only ratings whose value equals **5**, in order to simulate implicit *preference* in the form of ratings. The graphs with sliding windows are present for the *Ml-100k* and are meant to act just as a confirmation of that feature and to help visualize the difference between sliding windows. Some datasets were not used in its entirety due to time related issues in order to speed up the test process. The tests were done using a remote **Google Compute Engine VM**, with **8 vCPU's**, **2.2 GHz**, with **30 GB** of **RAM** working on an **Ubuntu 18.04** operating system. The graphs' x axis describes the ratings and the y axis, the related test metric.

5.1.1 Model-based algorithms tests

5.1.1.1 Implicit Feedback

Implicit Matrix Factorization

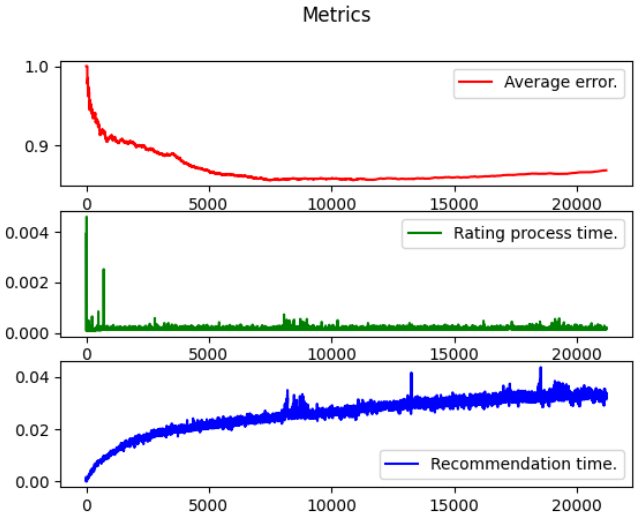

Figure 5.1: IMF; LF = 20; LR = 0.01; REG= 0.1; ML-100k.

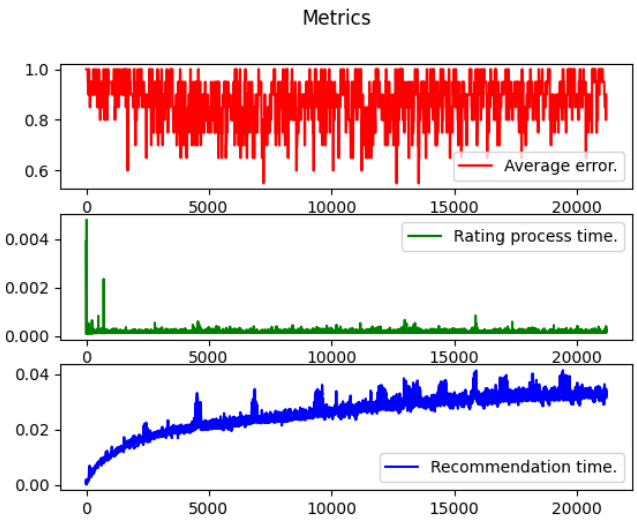

Figure 5.2: IMF; LF = 20; LR = 0.01; REG= 0.1; SW = 20 ; ML-100k.

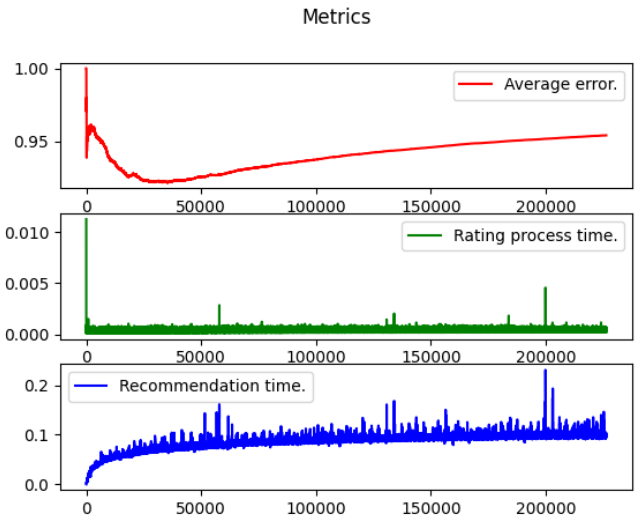

Figure 5.3: IMF; LF = 20; LR = 0.01; REG= 0.1; ML-1m.

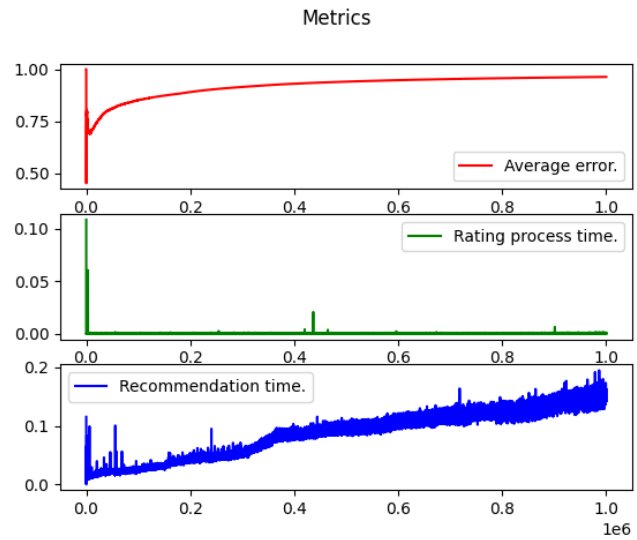

Figure 5.4: IMF; LF = 20; LR = 0.01; REG= 0.1; ML-10m.

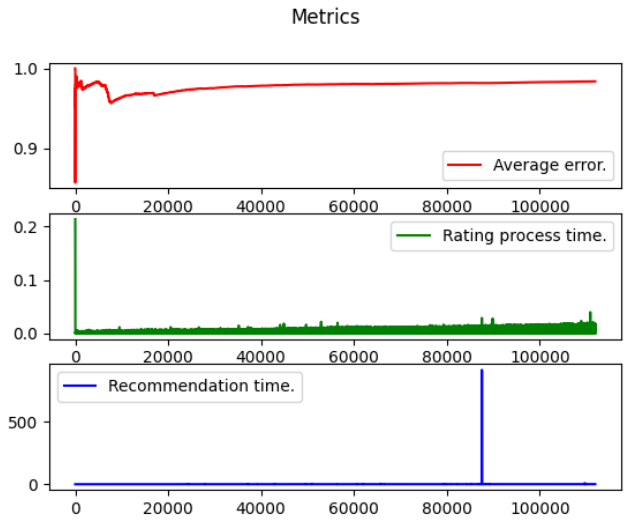

Figure 5.5: IMF; LF = 20; LR = 0.01; REG= 0.1; INESC.

As we can see from 5.1, 5.2, 5.3, 5.4 and 5.5, in terms of accuracy, there is an initial high error rate followed by a decrease at a steady pace until it reaches a point where it starts rising slowly again. The recommendation time has a small increase in time in contrast to the rating process time and we can spot some sudden increases in recommendation time along the chart.

5.1.1.2 Explicit Feedback

Explicit Matrix Factorization

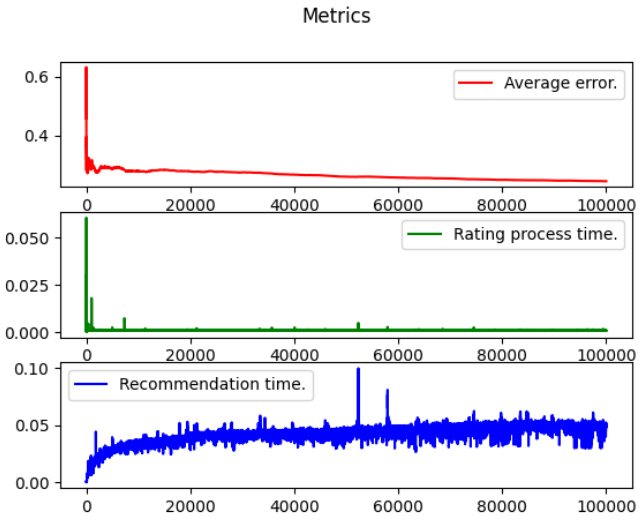

Figure 5.6: EMF; LF = 20; LR = 0.01; REG= 0.1; ML-100k.

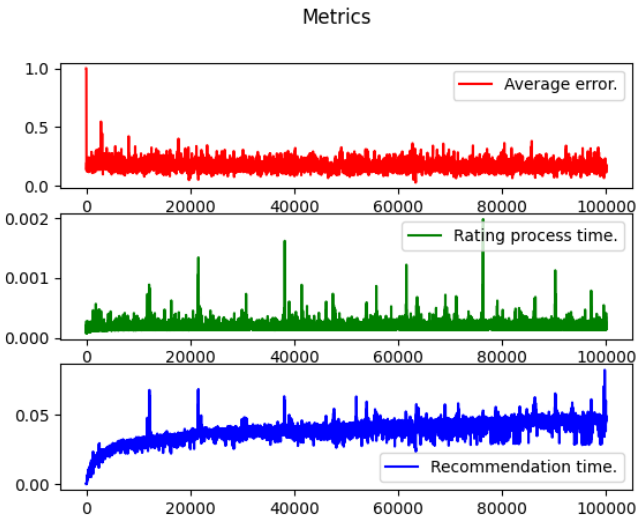

Figure 5.7: EMF; LF = 20; LR = 0.01; REG= 0.1; SW = 20; ML-100k.

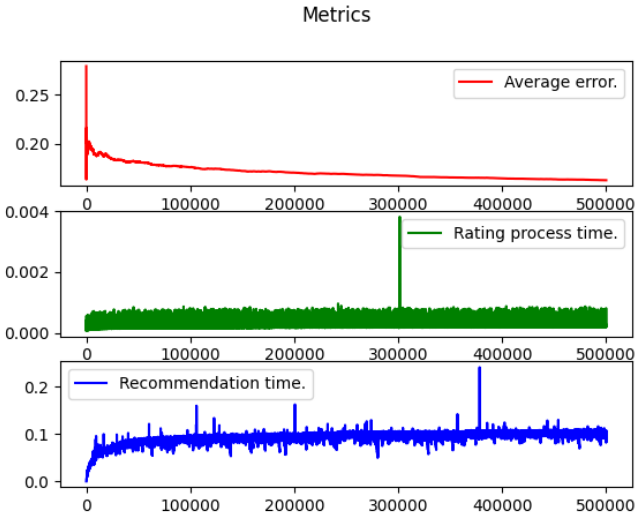

Figure 5.8: EMF; LF = 20; LR = 0.01; REG= 0.1; ML-1m.

Similarly, in figure 5.6, 5.7 and in figure 5.8 we can see the error decreasing steadily along the x axis. Only the recommendation time is worth mentioning since we can see its increase, even though limited, at each iteration.

Explicit Matrix Factorization with matrix preprocessing

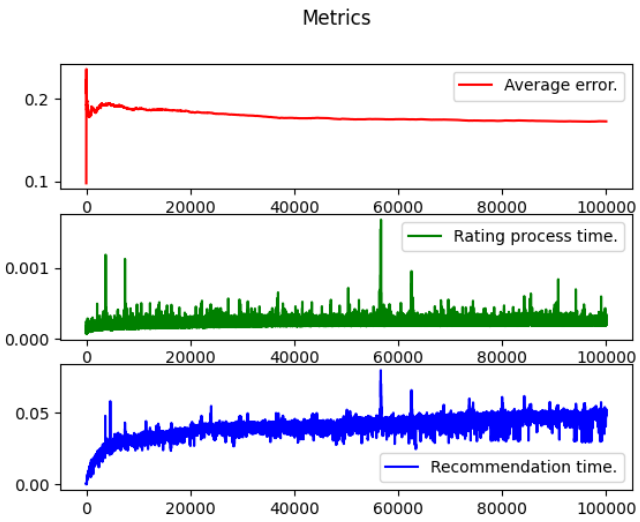

Figure 5.9: EMFP; LF = 20; LR = 0.01; REG= 0.1; ML-100k.

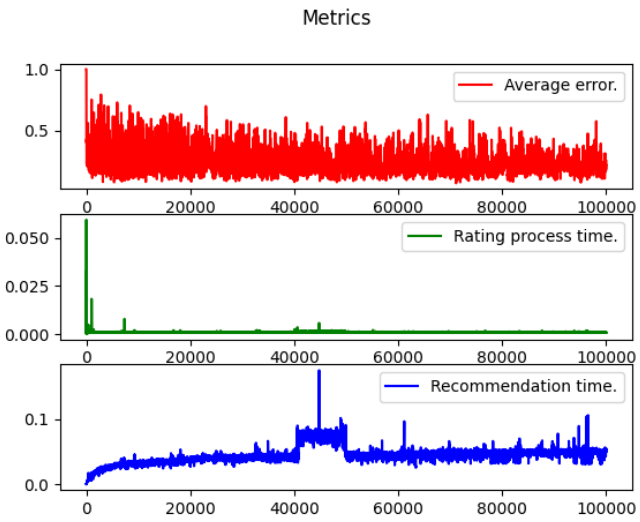

Figure 5.10: EMFP; LF = 20; LR = 0.01; REG= 0.1; SW = 20; ML-100k.

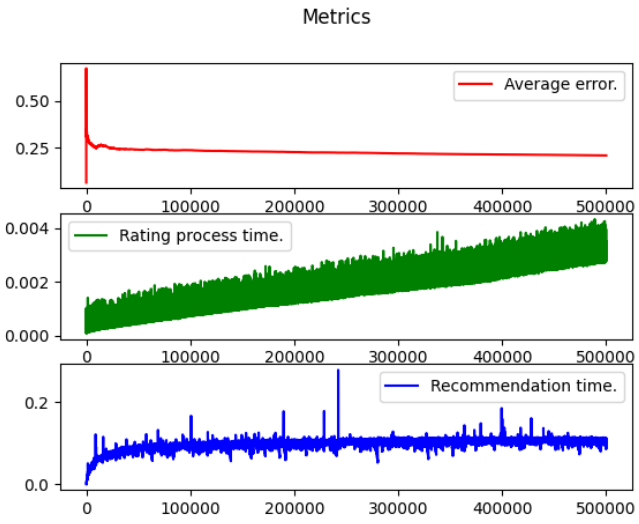

Figure 5.11: EMFP; LF = 20; LR = 0.01; REG= 0.1; ML-1m.

For explicit matrix factorization with matrix preprocessing, in image 5.9, 5.10, and in image 5.11 we can see a initial much more sudden decrease in the error, followed by a flat line with no substantial changes. Regarding the time measure the rating process time has low values in the order of milliseconds, which is much lower than the recommendation time.

5.1.2 Neighborhood-based algorithms tests

5.1.2.1 Implicit Feedback

Item-based neighborhood

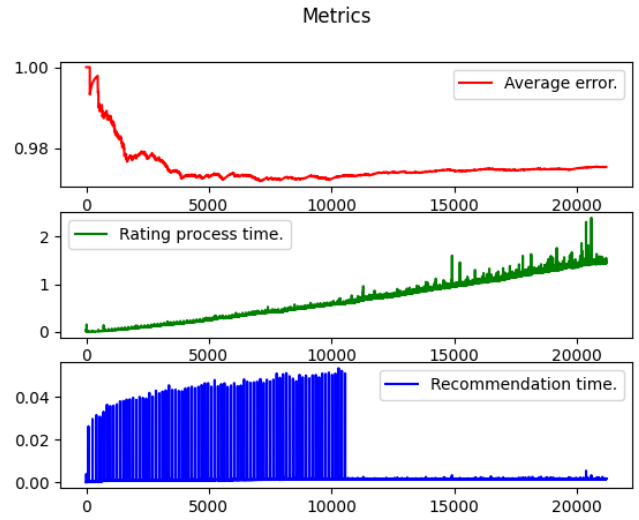

Figure 5.12: IIBN; NEIGHBORS = 5; ML-100k.

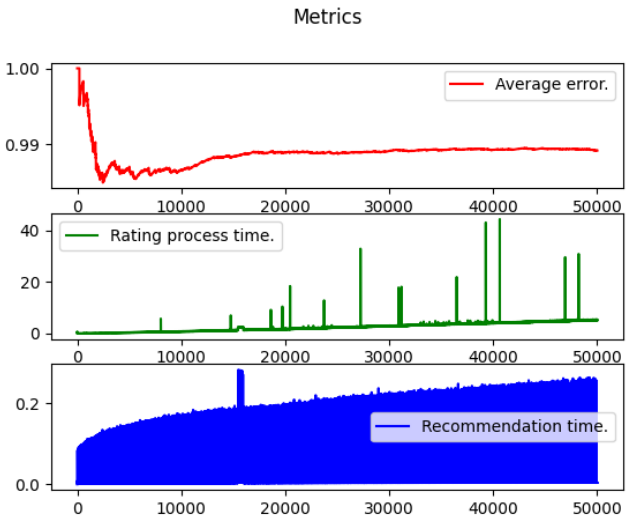

Figure 5.13: IIBN; NEIGHBORS = 5; ML-1m.

In figure 5.12 and in figure 5.13, we can see the error decreasing but the magnitude being much higher in comparison with matrix factorization methods. We can also that the increase in time processing is much higher causing the algorithm not to scale well.

Item-based clustering

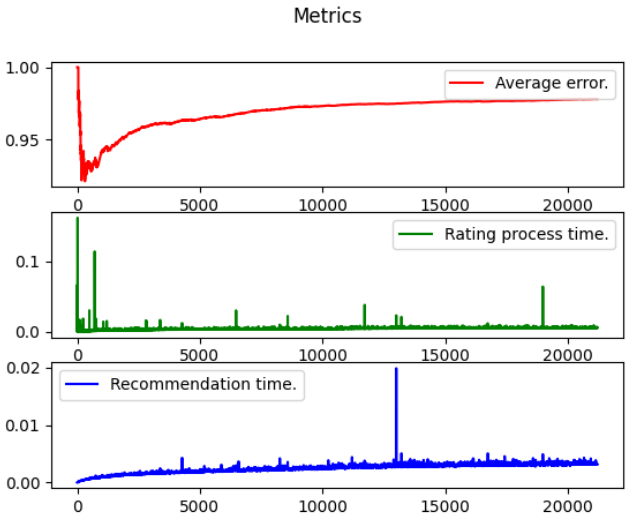

Figure 5.14: IIBC; NEIGHBORS = 5; ML-100k.

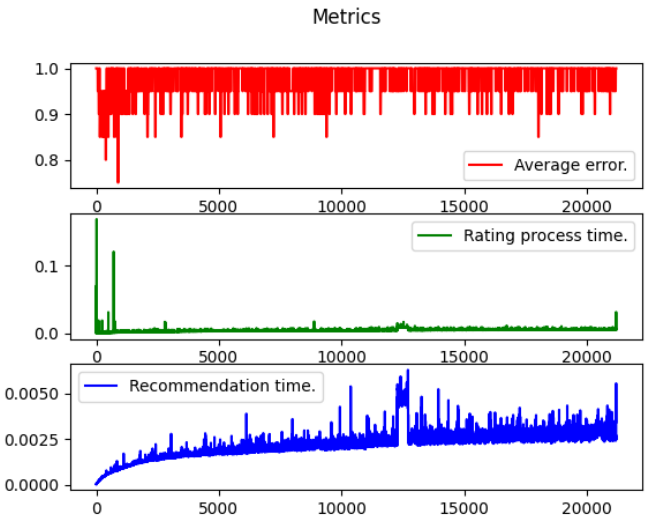

Figure 5.15: IIBC; NEIGHBORS = 5; SW = 20; ML-100k.

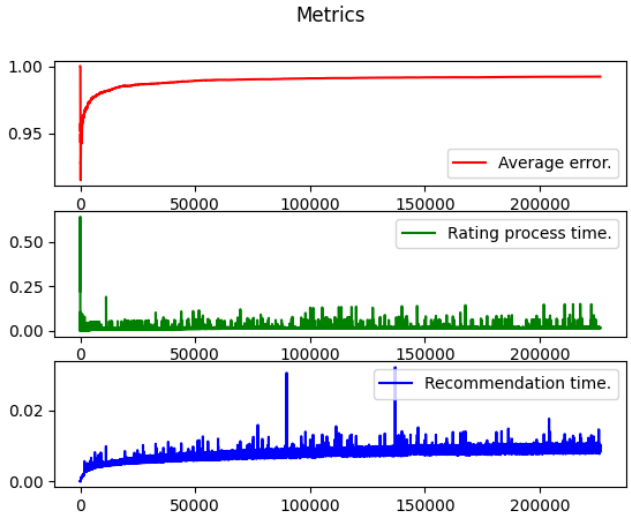

Figure 5.16: IIBC; NEIGHBORS = 5; ML-1m.

From figures 5.14, 5.15 and 5.16 we can see that in terms of time preprocessing the algorithm scales much better with this solution, since time processing values have a much lower magnitude. We can also see that after this initial decrease in the error rate it reaches a stable point without much noticeable differences.

Item-based locality-sensitive hashing

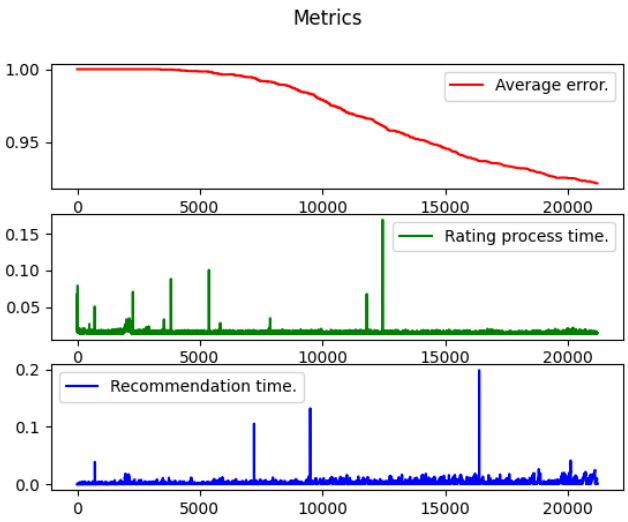

Figure 5.17: IILSMH; PERMS = 120; BANDS = 4; ML-100k.

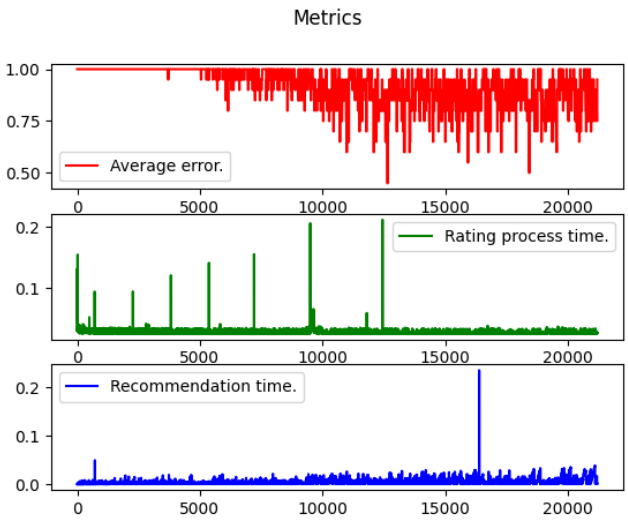

Figure 5.18: IILSMH; PERMS = 120; BANDS = 4; SW = 2; ML-100k.

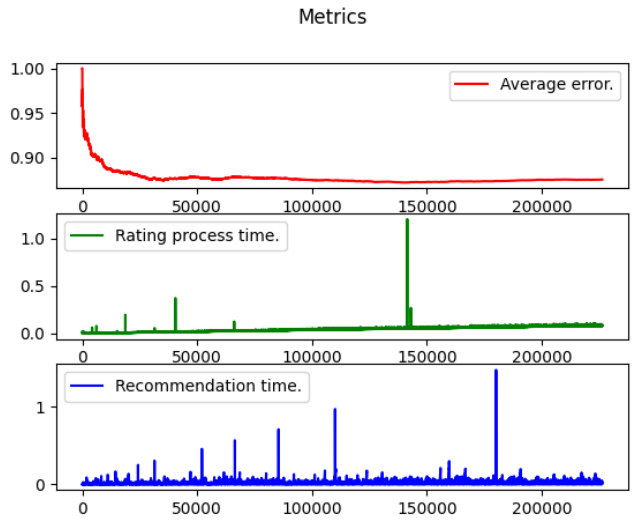

Figure 5.19: IILSMH; PERMS = 120; BANDS = 4; ML-1m.

In terms of item-based locality-sensitive min-hashing, by looking at images 5.17, 5.18 and 5.19, we can observe the decrease in the average error with the number of iterations, with surprisingly better recommendation accuracy than the other neighborhood-based approaches. The processing time and recommendation time do not vary significantly along time and have much lower values than the standard neighborhood approach.

User-based neighborhood

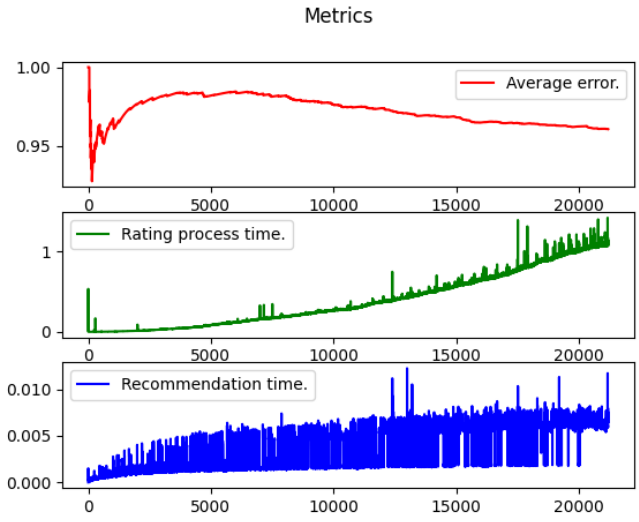

Figure 5.20: IUBN; NEIGHBORS = 5; ML-100k.

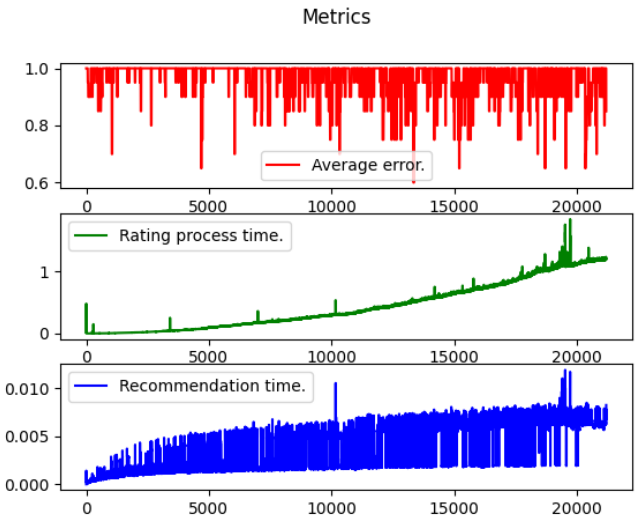

Figure 5.21: IUBN; NEIGHBORS = 5; SW = 20; ML-100k.

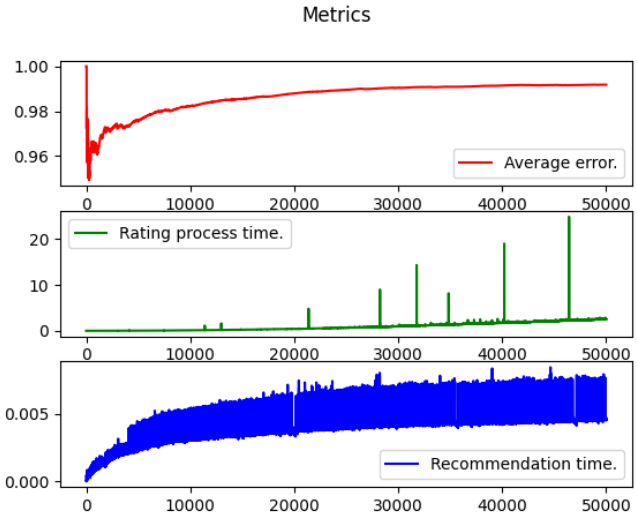

Figure 5.22: IUBN; NEIGHBORS = 5; ML-1m.

By looking at figures 5.20, 5.21 and 5.22, we can see an initial error instability with a decrease followed by a rise which then leads to a a steady decrease in the error with values similar to what happens in the item-based version. Similarly, by looking at the rating processing time, we can see its incapability to scale, as we see it reaching rating processing times of 1 or more seconds.

User-based clustering

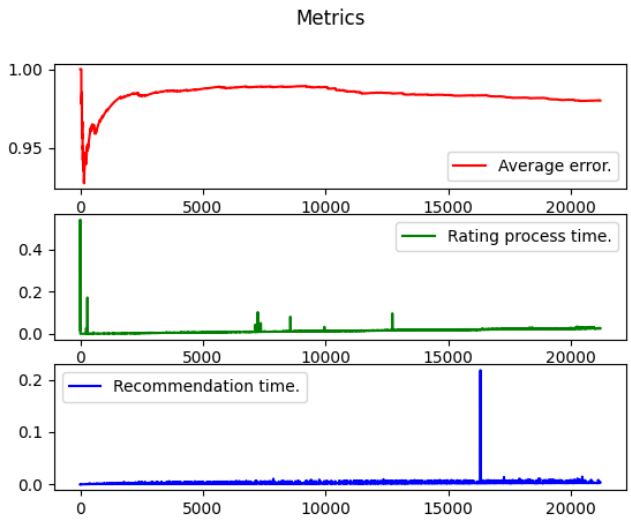

Figure 5.23: IUBC; NEIGHBORS = 5; ML-100k.

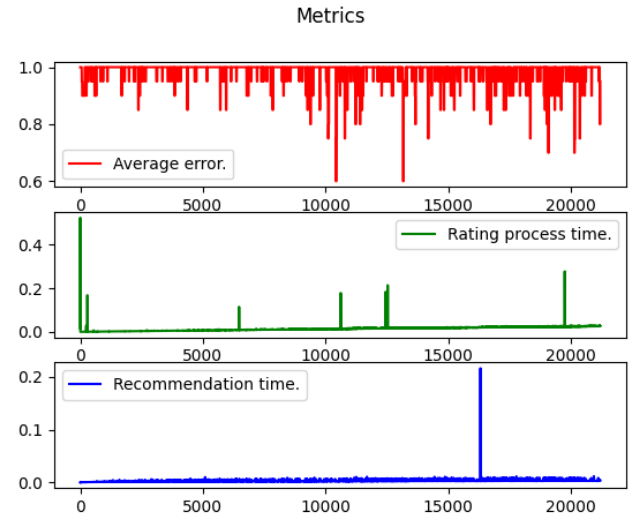

Figure 5.24: IUBC; NEIGHBORS = 5; SW = 20; ML-100k.

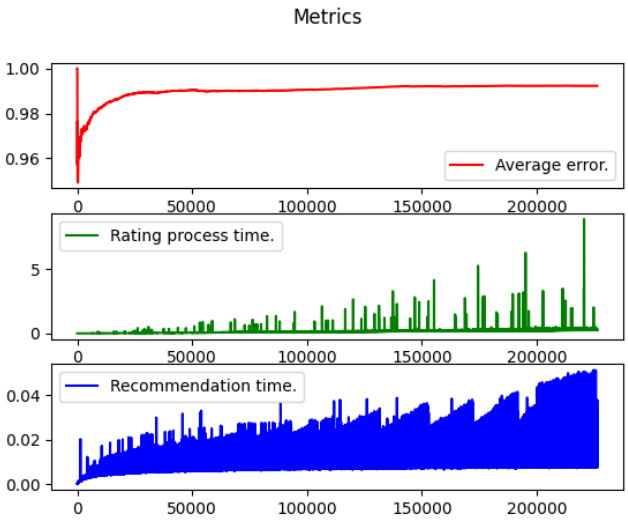

Figure 5.25: IUBC; NEIGHBORS = 5; ML-1m.

Similarly to what happened in the item-based version, if we look at figures 5.23, 5.24 and in 5.25, after an initial error decrease there is an error increase followed by a stable flat line. Recommendation and processing times do not vary significantly in relation with time even though they have higher values than the item-based version.

User-based locality-sensitive hashing

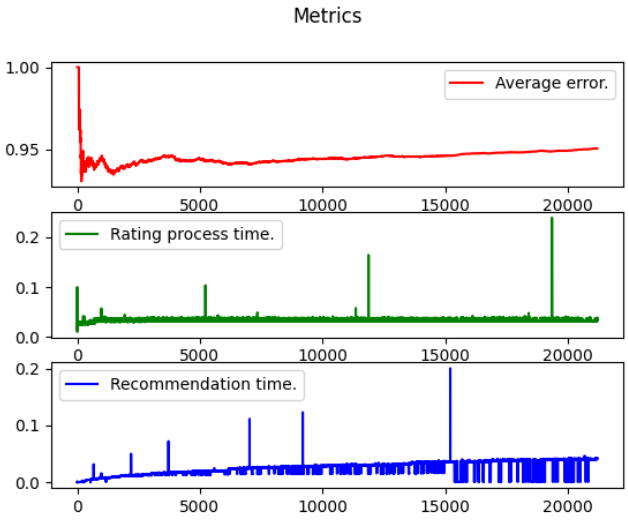

Figure 5.26: IULSMH; PERMS = 120; BANDS = 4; ML-100k.

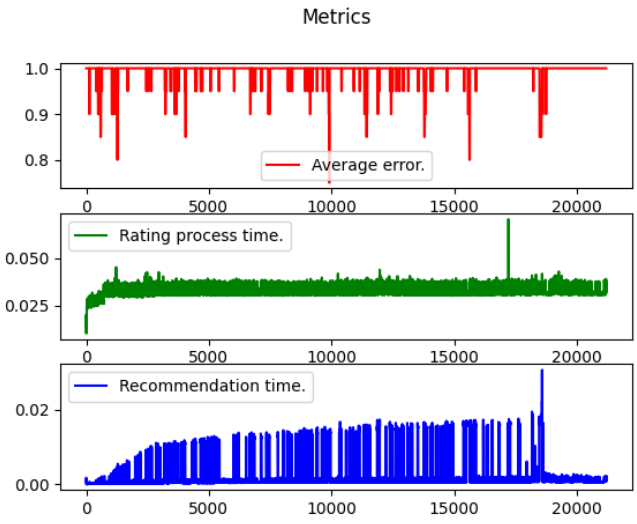

Figure 5.27: IULSMH; PERMS = 120; BANDS = 4; SW = 20; ML-100k.

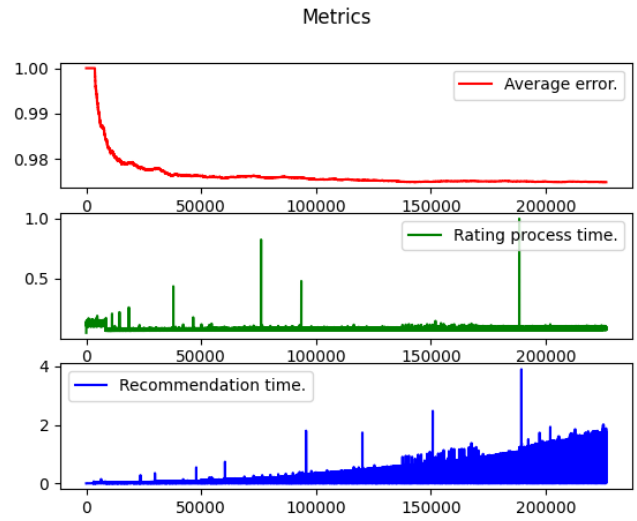

Figure 5.28: IULSMH; PERMS = 120; BANDS = 4; ML-1m.

In the user version of locality-sensitive min-hashing, by looking at images 5.26, 5.27 and 5.28, the average error line steadily decreases from the initial value 1, but according to the shown time metrics the algorithm does not scale well in terms of recommendation time.

5.1.2.2 Explicit Feedback

User-based neighborhood

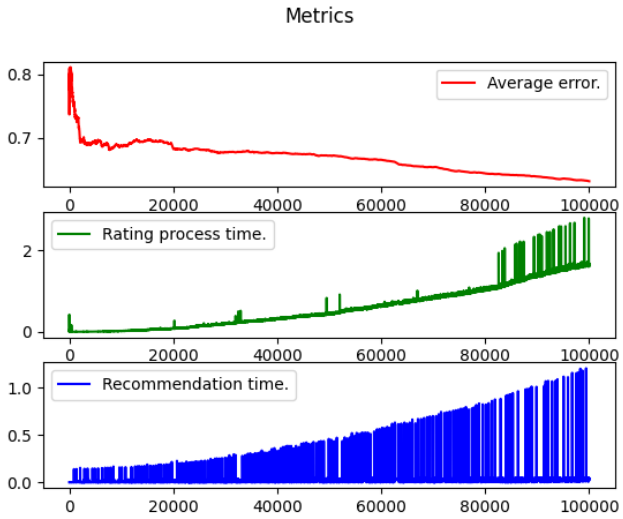

Figure 5.29: EUBN; NEIGHBORS = 5; ML-100k.

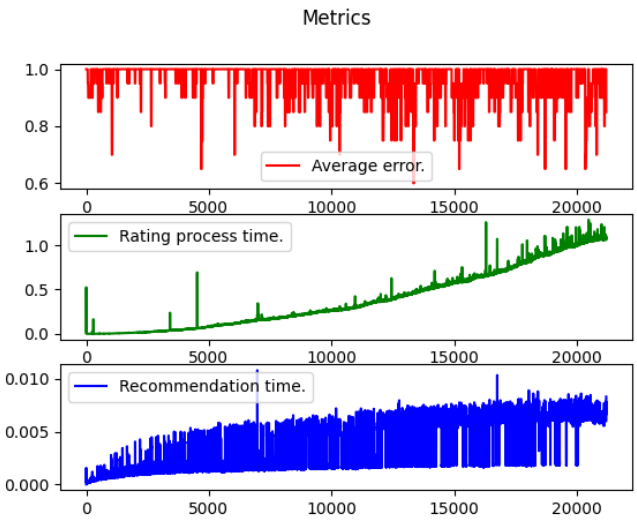

Figure 5.30: EUBN; NEIGHBORS = 5; SW = 20; ML-100k.

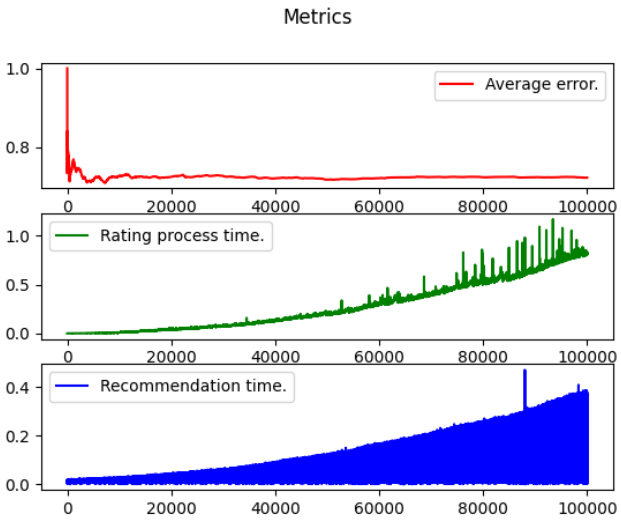

Figure 5.31: EUBN; NEIGHBORS = 5; ML-1m.

Regarding the explicit user-based neighborhood, from figures 5.29, 5.30 and 5.31, we can see the algorithm decreasing its average error before reaching a stable line with worse performance than explicit matrix factorization methods. In terms of processing time we can see it cannot scale well, as we see the rating process time reach values of 1 second at the end.

User-based clustering

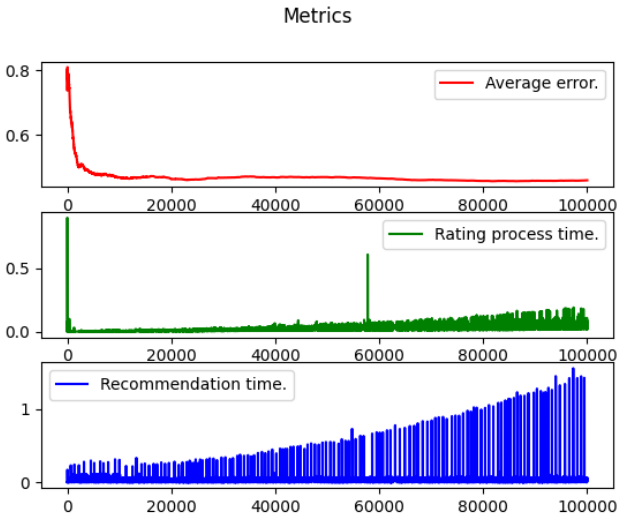

Figure 5.32: EUBC; NEIGHBORS = 5; ML-100k.

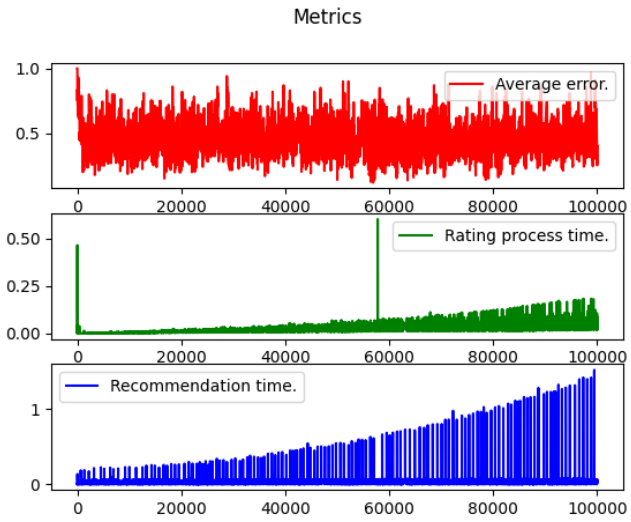

Figure 5.33: EUBC; NEIGHBORS = 5; SW = 20; ML-100k.

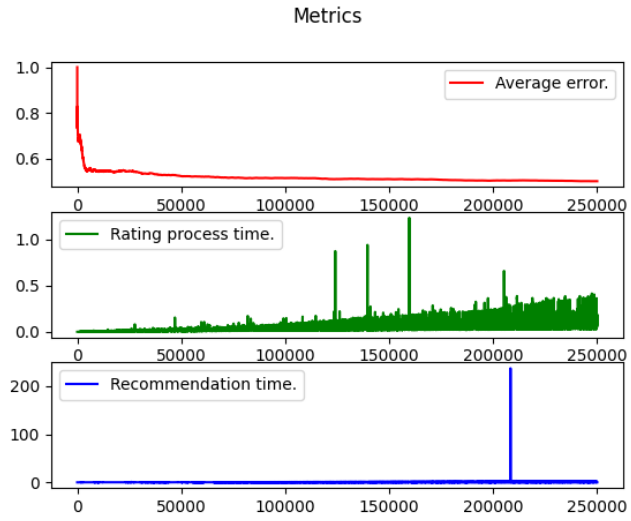

Figure 5.34: EUBC; NEIGHBORS = 5; ML-1m.

By observing 5.32, 5.33 and 5.34, in the clustering version we see the average error line behaving in the same way as in the standard neighborhood approach, and with values in a similar range. The rating process time scales much better, however it is increasing alongside the iterations. The recommendation also time increases with time to values even higher than the neighborhood approach.

5.1.2.3 Result discussion

First we can see that using different sliding windows portrays different accuracy results. This is related to the fact that not using a sliding window, calculates the average error at every iteration

which produces a more *"fluid"* line of evolution, contrasting with using a sliding window of size 20 which only calculates the error for the past 20 iterations, resulting in more abrupt *"spikes"* and sudden changes in the average error. There is another topic that needs to be discussed, which is related to the usage of dynamic data structures. Due to the dynamic environment on which data streams are involved in, the used datasets were ordered by timestamp, to replicate what happens in live systems. This meant that some of the processing time was spent on memory operations and allocation. In section 5.1, there are numerous evidences some *"spikes"* due to this characteristic. These sudden increases in time processing happen when a new user or item identifier has a much higher value than the size of the **DynamicArray** at that time, which causes some time to allocate memory until the index which equals that user or item identifier is reached. With datasets that have an higher number of users and items, this effect is much more noticeable because the identifiers will also be much higher. However, for most of the iterations, every identifier has already had its respective memory index allocated, so we only experience this phenomenon sporadically.

Implicit feedback

If we start with model-based algorithms, in section5.1.1.1 we can see the average error going down with time although at some point the algorithm loses generalization power and the error rate starts to ascend. This algorithm running time is the best of all the implemented ones for implicit feedback. For standard neighborhood methods in section 5.1.2.1 and in section 5.1.2.1, we can see the rating process time growing much faster in relation with the number of iterations when compared with other implementations which make these algorithms not suitable for data stream mining. This effect has to do with the time complexity of the algorithm since it calculates the entire neighborhood model at each iteration. The error rate acts as expected, decreasing at a very slow pace. Comparing them to other neighborhood approaches such as in section 5.1.2.1, we can initially see the recommendation power is almost none, but as soon as buckets start to form, the error steadily starts to go down. If we increase the number of bands the algorithm will take more time to start forming buckets and grouping elements, however the bucket quality will be much higher. Regarding time efficiency it is notoriously more efficient than standard neighborhood methods, even though it takes more time than matrix factorization implementations. We cannot really see the same effect with section 5.1.2.1, due to the differences we are performing recommendations on both algorithms. On the item-based version we can see the items which are in the same buckets than those items which the user has rated and directly sort them based on the number of appearances. The same cannot be done for the user-based version since buckets are made of users, and so we need to check the users that appear in the same buckets and return a shuffled list of the items these users rated. The recommendation quality is much higher in the item-based version of the locality-sensitive min hashing algorithm. For clustering algorithms, in sections 5.1.2.1 and 5.1.2.1, the recommendation quality was not affected when comparing them with standard neighborhood methods, but we can see that after an initial decrease, the error curve ascends, which has to do with elements inside clusters not being able to be candidates to recommendations of elements outside their cluster.

Explicit feedback

Similarly to what we did before, let us first discuss the model-based algorithm results. A similar behavior to 5.1.1.1 happens in 5.1.1.2, with a more initial abrupt error descent, due to the fact we are looking at explicit feedback, however we can see the error decreasing. By looking at section 5.1.1.2 we see an initial error instability but with a lower absolute value than the non preprocessing version, followed by a much slower curve descent. The rating normalization affects the initial error lowering it down. As for the explicit user-based neighborhood implementations, in section 5.1.2.2, we can also see the absolute average error decreasing but with a much higher absolute value in comparison with matrix factorization approaches. The clustering approach provides an initial error decrease followed by a flat line which is not a very good indicator of generalization.

Chapter 6

Conclusions and future work

This chapter is dedicated to this work's conclusions and plans for future improvements on *increc*. We'll discuss how *increc's* design and decisions influenced the shown results, what *increc's* implementation has lead to and mention limitations we currently have which could be implemented in the future.

6.1 Conclusions

Increc overall

Increc was implemented to be a state-of-the-art data stream mining for recommendation systems. It implements a series of baseline incremental algorithms for recommendation purposes on the model-based and neighborhood-based categories as well as mechanisms to evaluate and visualize their output. This library is meant to be used to aid the development and investigation of recommender systems.

Dynamic data structures

The usage of **Dynamic data structures** proved to be an interesting matter of discussion in two different ways. In one way, it helped to reduce the number of lines of code of each algorithm and helped separate the responsibility of memory allocation for unseen elements, either users or items in a specific place. On the other hand this implementation with these memory operations impacted the performance of the algorithms. Even for the **SymmetricMatrix** objects, the internal data is stored inside a **DynamicArray** object, so when algorithms process a new rating, there are many accesses to their data structures which use a *DynamicArray* has its internal storage which might cause some delay in processing, used for memory operations.

Object oriented programming

Even though Python is not the best language to be used for **OOP** and some critics might find it a bad design, this paradigm combined with the usage of **Composite**, was a great aid in grouping

common functionalities together and in giving a good, extensible and cohesive structure to the library. This was very important due to the amount of divisions that needed to be done, whether the algorithms were model-based or neighborhood-based, whether were focused on implicit or explicit feedback, whether they were item or user-based. They allowed the code to be divided in a understandable way.

Algorithmic comparison

The current implementations and their results coincide with the expected results. In terms of time execution and accuracy, matrix factorization methods overcome the rest of the implementations. Standard neighborhood-based proved to be unreasonable to be used in live systems due to their time complexity. Clustering solutions have better time performance, but with a generalizations power loss. Locality-sensitive min-hashing came as a good surprise to work around the neighborhood-search problem with better results and time efficiency.

From evaluators to a graphic visualization

This work implements a **prequential evaluator** mechanism for *implicit* or *explicit* feedback which computes test metrics for dynamic environments. These metrics include the average error rate, the time it takes to process a new rating, and the time it takes to make a recommendation for a user. The graphic module makes use of this evaluator to output a graphical evaluation of the output produced into 3 different charts for visualization.

6.2 Limitations and future work

One of *increc's* limitations is the fact that it does not support model persistance for future reuse. This means that, if we run an algorithm we have to manually inspect the objects fields and manually store them somehow, since increc does not automatically provide a way of doing this. This is one of the improvements which could be done in the future. The second limitation is the fact that there is no reference or mentioning to the use of deep learning in increc which could be one of the actions to develop increc for the future. Besides this, regarding locality-sensitive hashing, the only implementation concerns min-hashing. There are some other hashing techniques which aim to simulate similarity measures other than the jaccard distance. It would be nice to improve on the number of implemented hashing solutions. Another important limitation is related to the algorithm implementation. All of them were implemented according to their baseline definition as explained in literature. This results in some accuracy loss and there is a need for some tweaking in order to improve performance. The last limitation regards error measures implemented in the prequential evaluators. It would be much better if those measures could be changed to different ones in order to see how the algorithms perform under certain error measures.

Appendix A

Appendix

Proof of the inequality regarding [27] rating updates. Let us assume 3 users: $u_3 = \{1, None, 4, None, None\}$, $u_4 = \{3, None, 4, None, 5\}$, $u_5 = \{2, None, None, None, 3\}$, on a [0,5] rating scale. The average ratings of the rated items become: $\overline{u_3} = \frac{1+4}{2} = 2.5$, $\overline{u_4} = \frac{3+4+5}{3} = 4$ and $\overline{u_5} = \frac{2+3}{2} = 2.5$. If there is a rating update of the form $(5,4,5)$, then we represent u_5 as $\{2, None, None, None, 5\}$. The new average rating is $\overline{u_5} = 3.5$.

When the other user had already rated the item

The user's co-rated item items are the items with the indexes 0 and 4. The users' sample standard deviation using only co-rated items is calculated as
$\sigma_{u_4} = \sqrt{(3-4)^2 + (5-4)^2} \approx 1.4142$ and $\sigma_{u_5} = \sqrt{(2-2.5)^2 + (3-2.5)^2} \approx 0.7071$. The covariance between the users is $covariance(u_4, u_5) = ((3-4) \times (2-2.5) + (5-4) \times (3-2.5)) = 1.0$. With the new rating we calculate the user factors as $e = (5-3) \times (5-4) - (3.5-2.5) \times ((3-4) + (5-4)) = 2.0$, $f = (5-3)^2 \times 2 \times (5-3) \times (3-3.5) + 2 \times (3.5-2.5)^2 - 2 \times (3.5-2.5) \times ((2-2.5) + (3-2.5)) = -6.0$ and $g = 0$. This way the new similarity computation would be $\frac{1.0+2.0}{1.4142 \times 0.7071 - 6} \approx -0.4008$. However, if we recompute the terms after the update, the covariance is now equal to $(3-4) \times (2-3.5) + (5-4) \times (5-3.5) = 3.0$, and the variances are $\sigma_{u_4} = \sqrt{(3-4)^2 + (5-4)^2} \approx 1.4142$ and $\sigma_{u_5} = \sqrt{(2-2.5)^2 + (5-2.5)^2} \approx 2.5495$. This means that the e and g terms are correct but the f term is incorrect.

When the other user had not previously rated the item

The user's co-rated item item is the item with the index 0. The users' sample standard deviation using only co-rated items is calculated as
$\sigma_{u_3} = \sqrt{(1-2.5)^2} = 1.5$ and $\sigma_{u_5} = \sqrt{(2-2.5)^2} = 0.5$. The covariance between the users is $covariance(u_3, u_5) = ((1-2.5) \times (2-2.5)) = 0.75$. With the new rating we calculate the user factors as $e = -((3.5-2.5) \times (1-2.5)) = 1.5$, $f = 1 \times (3.5-2.5)^2 - 2 \times (3.5-2.5) \times (1-2.5 = 4.0$ and $g = 0$. This way the new similarity computation would be $\frac{0.75+1.5}{1.5 \times 0.5+4} \approx 0.3333$. However, if we recompute the terms after the update, the covariance is now equal to $(1-2.5) \times (2-3.5) =$

2.25, and the variances are $\sigma_{u_3} = \sqrt{(1-2.5)^2} = 1.5$ and $\sigma_{u_5} = \sqrt{(2-3.5)^2} = 1.5$. This means that the e and g terms are correct but the f term is incorrect.

References

[1] Announcing NVIDIA Merlin: An Application Framework for Deep Recommender Systems. https://devblogs.nvidia.com/announcing-nvidia-merlin-application-framework-for-deep-recommender-systems/. Accessed 25/05/2020.

[2] MOA - Machine Learning for Streams. https://moa.cms.waikato.ac.nz/. Accessed 22/11/2019.

[3] Principles and Techniques of Data Science. https://www.textbook.ds100.org/ch/16/reg_ridge.html. Accessed 30/05/2020.

[4] scikit-multiflow. https://scikit-multiflow.github.io/. Accessed 22/11/2019.

[5] The Legal Side of Open Source | Open Source Guides. https://opensource.guide/legal/. Accessed: 2020-05-10.

[6] Charu C Aggarwal. *Recommender Systems*. Springer International Publishing, 2016.

[7] Nicolas Auger, Cyril Nicaud, and Carine Pivoteau. Merge Strategies: from Merge Sort to TimSort. working paper or preprint, December 2015.

[8] Ahmet Maruf Aytekin and Tevfik Aytekin. Real-time recommendation with locality sensitive hashing. *Journal of Intelligent Information Systems*, pages 1–26, 2019.

[9] James Bennett, Stan Lanning, and Netflix Netflix. The netflix prize. In *KDD Cup and Workshop 2007*, 01 2009.

[10] Minh-Phung Do, Dung Nguyen, and Academic Network of Loc Nguyen. Model-based approach for collaborative filtering. In *The 6th International Conference on Information Technology for Education (IT@EDU2010)*, 08 2010.

[11] João Gama, Raquel Sebastião, and Pedro Pereira Rodrigues. On evaluating stream learning algorithms. *Machine Learning*, 90(3):317–346, Mar 2013.

[12] Zeno Gantner, Steffen Rendle, Christoph Freudenthaler, and Lars Schmidt-Thieme. MyMediaLite: A free recommender system library. In *5th ACM International Conference on Recommender Systems (RecSys 2011)*, 2011.

[13] Rainer Gemulla, Erik Nijkamp, Peter J. Haas, and Yannis Sismanis. Large-scale matrix factorization with distributed stochastic gradient descent. In *Proceedings of the 17th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '11, page 69–77, New York, NY, USA, 2011. Association for Computing Machinery.

[14] Miha Grčar, Dunja Mladenić, Blaž Fortuna, and Marko Grobelnik. Data sparsity issues in the collaborative filtering framework. pages 58–76, 10 2006.

[15] Guibing Guo, Jie Zhang, Zhu Sun, and Neil Yorke-Smith. Librec: A java library for recommender systems. In Alexandra I. Cristea, Judith Masthoff, Alan Said, and Nava Tintarev, editors, *Posters, Demos, Late-breaking Results and Workshop Proceedings of the 23rd Conference on User Modeling, Adaptation, and Personalization (UMAP 2015), Dublin, Ireland, June 29 - July 3, 2015*, volume 1388 of *CEUR Workshop Proceedings*. CEUR-WS.org, 2015.

[16] Nidhi Gupta and R.L. Ujjwal. An efficient incremental clustering algorithm. *World of Computer Science and Information Technology Journal (WCSIT)*, 3(4):97–99, 2013.

[17] Michael Hahsler. recommenderlab: A framework for developing and testing recommendation algorithms, 02 2015.

[18] F. Maxwell Harper and Joseph A. Konstan. The movielens datasets: History and context. *ACM Trans. Interact. Intell. Syst.*, 5(4), December 2015.

[19] Jure Leskovec Jeffrey D. Ullman and Anand Rajaraman. Recommendation systems. In *Mining Massive Datasets*, pages 325–360. Cambridge University Press, Cambridge, 2013.

[20] Olivier Jeunen, Koen Verstrepen, and Bart Goethals. Efficient similarity computation for collaborative filtering in dynamic environments. In *Proceedings of the 13th ACM Conference on Recommender Systems*, RecSys '19, page 251–259, New York, NY, USA, 2019. Association for Computing Machinery.

[21] Takuya Kitazawa. Incremental factorization machines for persistently cold-starting online item recommendation. *ArXiv*, abs/1607.02858, 07 2016.

[22] Y. Koren, R. Bell, and C. Volinsky. Matrix factorization techniques for recommender systems. *Computer*, 42(8):30–37, 2009.

[23] Soojung Lee. Improving jaccard index for measuring similarity in collaborative filtering. In Kuinam Kim and Nikolai Joukov, editors, *Information Science and Applications 2017*, pages 799–806, Singapore, 2017. Springer Singapore.

[24] Maizan Mat Amin, Jannifer Lan, Mokhairi Makhtar, and Abd Mamat. A decision tree based recommender system for backpackers accommodations. *International Journal of Engineering and Technology(UAE)*, 7:45–48, 04 2018.

[25] Catarina Miranda and Alípio M. Jorge. Incremental collaborative filtering for binary ratings. In *Proceedings of the 2008 IEEE/WIC/ACM International Conference on Web Intelligence and Intelligent Agent Technology - Volume 01*, WI-IAT '08, page 389–392, USA, 2008. IEEE Computer Society.

[26] Travis E Oliphant. *A guide to NumPy*, volume 1. Trelgol Publishing USA, 2006.

[27] Manos Papagelis, Ioannis Rousidis, Dimitris Plexousakis, and Elias Theoharopoulos. Incremental collaborative filtering for highly-scalable recommendation algorithms. In Mohand-Said Hacid, Neil V. Murray, Zbigniew W. Raś, and Shusaku Tsumoto, editors, *Foundations of Intelligent Systems*, pages 553–561, Berlin, Heidelberg, 2005. Springer Berlin Heidelberg.

[28] Dimitris Paraschakis and Bengt J. Nilsson. Flowrec: Prototyping session-based recommender systems in streaming mode. In Hady W. Lauw, Raymond Chi-Wing Wong, Alexandros Ntoulas, Ee-Peng Lim, See-Kiong Ng, and Sinno Jialin Pan, editors, *Advances in Knowledge Discovery and Data Mining*, pages 65–77, Cham, 2020. Springer International Publishing.

[29] Miguel Sozinho Ramalho, João Vinagre, Alípio Mário Jorge, and Rafaela Bastos. Incremental multi-dimensional recommender systems: Co-factorization vs tensors. In João Vinagre, Alípio Mário Jorge, Albert Bifet, and Marie Al-Ghossein, editors, *Proceedings of the 2nd Workshop on Online Recommder Systems and User Modeling*, volume 109 of *Proceedings of Machine Learning Research*, pages 21–35, ACM RecSys 2019, Copenhagen, Denmark, 19 Sep 2019. PMLR.

[30] Francesco Ricci, Lior Rokach, and Bracha Shapira. *Introduction to Recommender Systems Handbook*, pages 1–35. Springer US, Boston, MA, 2011.

[31] Christophe Salperwyck, Vincent Lemaire, and Carine Hue. Incremental weighted naive bayes classifiers for data stream. *Springer in the Springer Series "Studies in Classification, Data Analysis, and Knowledge Organization*, 06 2014.

[32] Jesus Bobadilla Sancho, Fernando Ortega Requena, Antonio Hernando Esteban, and Jesús Bernal Bermúdez. A collaborative filtering approach to mitigate the new user cold start problem. *Knowledge-Based Systems*, 26:225–238, February 2012.

[33] Badrul Sarwar, George Karypis, Joseph Konstan, and John Riedl. Item-based collaborative filtering recommendation algorithms. In *Proceedings of the 10th International Conference on World Wide Web*, WWW '01, page 285–295, New York, NY, USA, 2001. Association for Computing Machinery.

[34] Jeffrey C. Schlimmer and Douglas Fisher. A case study of incremental concept induction. In *Proceedings of the Fifth AAAI National Conference on Artificial Intelligence*, AAAI'86, page 496–501. AAAI Press, 1986.

[35] Leily Sheugh and Sasan H. Alizadeh. A note on pearson correlation coefficient as a metric of similarity in recommender system. *2015 AI & Robotics (IRANOPEN)*, pages 1–6, 2015.

[36] Takuya Kitazawa. Flurs: A python library for online item recommendation. https://takuti.me/note/flurs/. Accessed 22/11/2019.

[37] Loren Terveen and Will C. B. Hill. Beyond recommender systems: Helping people help each other. In *In HCI In The New Millennium, Jack Carroll, ed., Addison-Wesley, 2001*, 02 2001.

[38] Paul E Utgoff. Id5: An incremental id3. In *Fifth International Conference on Machine Learning*, pages 107–120. Morgan Kaufmann Publishers, 1988.

[39] Paul E. Utgoff. Incremental induction of decision trees. *Mach. Learn.*, 4(2):161–186, November 1989.

[40] Stefan Van Der Walt, S Chris Colbert, and Gael Varoquaux. The numpy array: a structure for efficient numerical computation. *Computing in Science & Engineering*, 13(2):22, 2011.

[41] Benjamin Van Durme and Ashwin Lall. Online generation of locality sensitive hash signatures. In *Proceedings of the ACL 2010 Conference Short Papers*, pages 231–235, Uppsala, Sweden, July 2010. Association for Computational Linguistics.

[42] Guido Van Rossum and Fred L. Drake. *Python 3 Reference Manual*. CreateSpace, Scotts Valley, CA, 2009.

[43] John Verzani. *Using R for Introductory Statistics*. CHAPMAN & HALL/CRC, 2005.

[44] João Vinagre, Alípio Mário Jorge, and João Gama. Fast incremental matrix factorization for recommendation with positive-only feedback. In Vania Dimitrova, Tsvi Kuflik, David Chin, Francesco Ricci, Peter Dolog, and Geert-Jan Houben, editors, *User Modeling, Adaptation, and Personalization*, pages 459–470, Cham, 2014. Springer International Publishing.

[45] Wei Wu, Bin Li, Ling Chen, Junbin Gao, and Chengqi Zhang. A review for weighted minhash algorithms. *ArXiv*, abs/1811.04633, 2018.

[46] Xiao Yang, Zhaoxin Zhang, and Ke Wang. Scalable collaborative filtering using incremental update and local link prediction. In *Proceedings of the 21st ACM International Conference on Information and Knowledge Management*, CIKM '12, page 2371–2374, New York, NY, USA, 2012. Association for Computing Machinery.