



UNIVERSITAT  
POLITÈCNICA  
DE VALÈNCIA



Escola Tècnica  
Superior d'Enginyeria  
Informàtica

Escola Tècnica Superior d'Enginyeria Informàtica  
Universitat Politècnica de València

# **Machine Learning based Models for Matrix Factorization**

**DEGREE FINAL WORK**

Degree in Computer Engineering

*Author:* Salvador Carrión Ponz

*Tutor:* Jon Ander Gómez Adrián

Course 2016-2017



# Resum

El filtratge col·laboratiu (FC) és el procés de filtrat d'informació o patrons utilitzant tècniques que impliquen la col·laboració entre múltiples agents o fonts de dades[1].

Aquest treball mostra la senzillesa i eficiència dels mètodes d'aprenentatge automàtic per a resolver problemes de gran escala en temps lineal, així com els desafiaments de les optimitzacions no-convexes en l'espai de problemes no-lineals.

En primer lloc, descriu el camp del FC, els seus desafiaments i l'estat de l'art actual, centrant el treball en els models basats en l'aprenentatge automàtic per a la factorització de matrius. Després, explica els problemes i desafiaments de les optimitzacions no-convexes, a més de introduir diverses tècniques d'optimització i els seus avantatges sobre els mètodes d'optimització estocàstica clàssica. Més endavant, presenta breument l'arquitectura d'una implementació eficient per a *Orange3*, a fi d'incloure suport per al FC. Posteriorment, mostra diverses formes de visualitzar i explotar els factors latents resultants produïts pels models, així com un *add-on* per a visualitzar el comportament de diferents optimitzadors estocàstics.

Finalment, presenta un model que té en compte les dinàmiques temporals, la informació addicional i de confiança, tant explícita com implícita.

**Paraules clau:** aprenentatge automàtic, mineria de dades, factorització de matrius, optimització, filtrat col·laboratiu, visualització, orange3

---

# Resumen

El filtrado colaborativo (FC) es el proceso de filtrado de información o patrones utilizando técnicas que implican la colaboración entre múltiples agentes o fuentes de datos[1].

Este trabajo muestra la sencillez y eficiencia de los métodos de aprendizaje máquina para resolver problemas de gran escala en tiempo lineal, así como los desafíos de las optimizaciones no convexas en el espacio de problemas no lineales.

En primer lugar, describo el campo del FC, sus desafíos y el estado del arte actual, centrandome el trabajo en los modelos basados en el aprendizaje automático para la factorización de matrices. Después, explico los problemas y desafíos de las optimizaciones no convexas, además de introducir varias técnicas de optimización y sus ventajas sobre los métodos de optimización estocástica clásica. Más adelante, presento brevemente la arquitectura de una implementación eficiente para *Orange3* a fin de incluir soporte para el FC. Posteriormente, muestro varias formas de visualizar y explotar los factores latentes resultantes producidos por los modelos, así como un *add-on* para visualizar el comportamiento de diferentes optimizadores estocásticos.

Finalmente, presento un modelo que tiene en cuenta las dinámicas temporales, la información adicional y de confianza, tanto explícita como implícita.

**Palabras clave:** aprendizaje automático, mineria de datos, factorización de matrices, optimización, filtrado colaborativo, visualización, orange3

---

# Abstract

Collaborative filtering (CF) is the process of filtering for information or patterns using techniques involving collaboration among multiple agents or data sources[1].

This dissertation presents the simplicity and efficiency of machine learning approaches to solve CF large-scale problems in a linear time, along with the challenges of non-convex optimizations in the space of non-linear problems.

First, I describe the field of CF, its challenges, and the current state-of-the-art, focusing the work on ML<sup>1</sup> based models for matrix factorization. After explaining the problems and challenges of non-convex optimizations, along with several optimization techniques and their performance advantage over the classical stochastic optimization. Later, I briefly outline the architecture of an efficient implementation for *Orange3* to include support for CF. Subsequently, I show several ways to visualize and exploit the resulting latent factors produced by the models, as well as an add-on for visualizing the behavior of different stochastic optimizers.

Finally, I present a model that takes into account temporal dynamics for side and trust information, both explicit and implicit.

**Key words:** machine learning, data mining, matrix factorization, optimization, collaborative filtering, visualization, orange3

---

---

<sup>1</sup>ML: Machine Learning

# Contents

---

<b>Contents</b>	<b>v</b>
<b>List of Figures</b>	<b>vii</b>
<b>List of Tables</b>	<b>viii</b>

---

<b>1 Introduction</b>	<b>1</b>
1.1 Motivation	1
1.2 Goals	1
1.3 Work Structure	2
<b>2 State of the art</b>	<b>3</b>
2.1 Overview	3
2.2 Matrix Factorization Approaches	3
<b>3 Collaborative Filtering</b>	<b>5</b>
3.1 Types of CF	5
3.1.1 Memory-based	5
3.1.1.1 Average-based	5
3.1.1.2 User-based	6
3.1.1.3 Item-based	7
3.1.2 Model-based	7
3.1.2.1 Clustering	7
3.1.2.2 Association Rules	8
3.1.2.3 Matrix Factorization	8
3.1.2.4 Restricted Boltzmann Machine (RBMs)	9
3.1.2.5 Recurrent Neural Networks	10
3.2 Challenges of CF	10
3.2.1 Cold Start	10
3.2.2 Data sparsity	11
3.2.3 Scalability	11
3.2.4 Popularity Bias	12
3.2.5 Diversity	12
3.3 Models for Matrix Factorization	12
3.3.1 Rating	12
3.3.1.1 BRISMF	12
3.3.1.2 SVD++	15
3.3.1.3 TimeSVD	16
3.3.1.4 TrustSVD	16
3.3.2 Ranking	18
3.3.2.1 BPR	18
3.3.2.2 CLiMF	19
3.4 Evaluation and testing	20
3.4.1 Rating	22
3.4.2 Ranking	24
3.5 Optimization	24
3.5.1 Gradient descent	24

3.5.2	Optimizers for gradient descent	26
3.5.2.1	Stochastic Gradient Descent (SGD)	26
3.5.2.2	Momentum	27
3.5.2.3	Nesterov's Accelerated Gradient (NAG)	28
3.5.2.4	AdaGrad	29
3.5.2.5	RMSProp	30
3.5.2.6	AdaDelta	30
3.5.2.7	Adam	31
3.5.2.8	Adamax	31
3.5.3	Non-convexity of MF problems	32
<b>4</b>	<b>Software developed</b>	<b>35</b>
4.1	Orange3	35
4.2	Orange3-Recommendation	37
4.2.1	Architecture	37
4.2.2	Technologies	39
4.2.3	Workflow	40
4.2.4	Documentation	40
4.2.5	Blog	41
4.2.6	Tutorials	41
4.2.7	Data Representation	42
4.2.7.1	Results from the <i>TrustSVD</i> experiment	44
4.2.7.2	Issue with the <i>Prediction</i> widget	45
4.2.8	Scripting	46
4.2.9	Widgets	48
4.2.10	Getting started	48
4.2.10.1	Training a model	48
4.2.10.2	Cross-Validation	50
4.2.10.3	Making recommendations	50
4.2.10.4	Analyzing low-rank matrices	51
4.3	Orange3-Educational	53
4.3.1	Extension	53
<b>5</b>	<b>Experiments and Results</b>	<b>55</b>
5.1	Datasets	55
5.2	Performance Comparison	55
5.2.0.1	Rating	55
5.2.0.1.1	FilmTrust	55
5.2.0.1.2	MovieLens100K	56
5.2.0.1.3	MovieLens1M	56
5.2.0.1.4	MovieLens10M	57
5.2.0.2	Ranking	57
5.2.0.2.1	Epinions	57
5.3	Visualization techniques	58
5.3.1	SGD optimizers	58
5.3.2	Latent factors	59
<b>6</b>	<b>Conclusions and future work</b>	<b>63</b>
	<b>Bibliography</b>	<b>65</b>

# List of Figures

---

3.1	Categorization of comedy domains [29]	8
3.2	Matrix factorization	8
3.3	Architecture of a RBM	9
3.4	Architecture of a RNN for CF	10
3.5	Cross-validation process [33]	21
3.6	Evaluation metrics (RMSE) [7]	23
3.7	Evaluation metrics (top-k rec.) [7]	23
3.8	Local and global maxima and minima [34]	25
3.9	A saddle point on the graph of $z = x^2y^2$ (in red) [35]	26
3.10	Fluctuation during a SGD optimization	27
3.11	Momentum update <b>Source:</b> <i>CS231n Convolutional Neural Networks for Visual Recognition</i>	28
3.12	Momentum update Vs. Nesterov update <b>Source:</b> <i>CS231n Convolutional Neural Networks for Visual Recognition</i>	29
4.1	Orange3 Data Mining	35
4.2	Terminal execution	36
4.3	Python implementation	36
4.4	Model-View-Controller (MVC)	37
4.5	The class structure of <i>Orange3-Recommendation</i>	38
4.6	Orange3-Recommendation tutorial	41
4.7	Prediction times	46
4.8	Widgets in the <i>Recommendation</i> category	48
4.9	Feeding a model with ratings	48
4.10	Side/Trust information	49
4.11	TrustSVD settings	49
4.12	Cross-Validation flow	50
4.13	TrustSVD settings	50
4.14	TrustSVD settings	51
4.15	Low-rank matrices	51
4.16	Visualizing items	52
4.17	Recommendation workflow	52
4.18	Orange3-Educational	53
4.19	Gradient Descent Widget	54
4.20	Marching squares algorithm[36]	54
5.1	Gradient Descent; $\alpha = 1.0$	58
5.2	Vanilla SGD; $\alpha = 1.0$	58
5.3	Nesterov Momentum; $\alpha = 1.0$	58
5.4	RSMProp; $\alpha = 1.0$	58
5.5	AdaGrad; $\alpha = 5.0$	59
5.6	Adamax; $\alpha = 5.0$	59
5.7	Different loss map	59
5.8	Small divergence	59

5.9	Scatter plot (Movies)	60
5.10	Linear projection (Fantasy)	60
5.11	Linear projection (Sci-fi)	60
5.12	Informative projections	61
5.13	Ranked clusters (Crime)	61
5.14	Ranked clusters (Romance)	61
5.15	Scatter map	62

## List of Tables

---

3.1	Toy dataset	21
4.1	Comparison of dense and sparse data representation.	42
5.1	Datasets overview	55
5.2	Filmtrust benchmark results	56
5.3	MovieLens100K benchmark results	56
5.4	MovieLens1M benchmark results	57
5.5	MovieLens10M benchmark results	57
5.6	Epinions benchmark results	57



---

---

# CHAPTER 1

## Introduction

---

Over the past decade, the volume of data generated by users and companies has grown exponentially and it's likely to continue growing at an exponential rate.

Due to the new economies, companies such as retailers and content providers have been forced to increase the amount of products and services available. The reason behind this is to stay competitive and meet the variety of special needs and tastes of a worldwide market. This has had important consequences for consumers, both positives and negatives. Among the negatives, one of the most notable is that consumers are inundated with choices. Therefore, although a product or service might exist, sometimes can it be really hard to find it. Consequently, companies have tried to predict, segment and personalize the available offer by exploiting massive datasets. Thus, matching consumers with the most appropriate products is a key factor to enhance the user loyalty and satisfaction, namely, the numbers of sales. Because of the importance of good recommendations, companies have invested great efforts in research for better and scalable collaborative filtering algorithms. But it wasn't until 2006 when Netflix opened a competition for the best collaborative filtering algorithm, when scientists began to see for the very first time an important progress in the field.<sup>1</sup>

The Netflix Prized competition demonstrated that matrix factorization models are superior to classical techniques, at least for contests. This, along with the recent advances in machine learning has led to a new and interesting path for research with a very high economic potential.

### 1.1 Motivation

---

This work is focused on the study of machine learning models for matrix factorization. I chose this topic due to its relevance in the field of collaborative filtering. Although other approaches for getting good recommendations exist, ML<sup>2</sup> approaches have demonstrated higher accuracy and more flexibility than traditional ones. Furthermore, these models usually have to deal with enormously large volumes of data so we have to come up with smart ideas to solve all those extra challenges if we want to end with a decent model.

### 1.2 Goals

---

The goals for this project are the following:

---

<sup>1</sup>Prizes were based on improvements over Netflix's own algorithm (*Cinematch*), or the previous year's score if a team has made improvement beyond a certain threshold. Furthermore, Netflix offered a grand prize of \$1,000,000 to the team who improves the score of their algorithms by at least 10%[\[31\]](#).

<sup>2</sup>ML: Machine Learning

1. Research the field of collaborative filtering (CF)
2. Understand how ML can be applied to CF
3. Implement ML models able to deal with large volumes of data (TBs<sup>3</sup>)
4. Research optimization techniques for non-convex problems
5. Develop methods and techniques to visualize and exploit the outcome of the models

### 1.3 Work Structure

---

First, I introduce the state-of-the-art of collaborative filtering. Then, I describe the most relevant techniques in the field, along with their problems and limitations. After that, factorization models are studied in detail as well as optimization techniques for non-convex problems and their effects on the presented models. Later, I analyze an add-on for *Orange3 Data mining* which I have developed in order to simplify the recommendation pipeline so you can train, test, score and visualize matrix factorization models and its outputs using a unified and efficient interface. In addition, I have also extended an educational add-on so students and researchers can easily understand the behavior of different optimizers under different problems and settings. Finally, I cover a series of methods to visualize and analyze the latent factors obtained by the models.

---

<sup>3</sup>TB: Terabytes

---

---

## CHAPTER 2

# State of the art

---

In 2006, Netflix opened a competition for the best collaborative filtering algorithm to predict user ratings for films based on previous ratings[31]. They published a dataset with 100M ratings and set a prize of \$1,000,000 to the team who improve their own algorithm by at least a 10%. From this point, thousands of contestants fight for the prize accelerating the research in the field of collaborative filtering. In September of 2009, the *BellKor's Pragmatic Chaos* won the contest with a revolutionary idea inspired by *Simon Funk* which consisted in using ML to factorize a given sparse matrix.

This was the breakthrough the industry needed to keep developing this field.

### 2.1 Overview

---

In the topic of collaborative filtering we can find many approaches, usually divided into memory-based, model-based and hybrid.

Currently, the research of memory-based approaches for CF is pretty small so it is vastly inferior to the model-based ones. Typically, these memory-based models are based on averages, similarities and correlations. Consequently, they perform very poorly and too slow. However, they are pretty useful as baselines for other models.

With regard to model-based approaches, matrix factorization and statistical linear models are generally the *by default* choice for most of the CF problems. Nevertheless, there are other completely different approaches based on clustering, association rules, Restricted Boltzmann Machine and Recurrent Neural Networks, that are either less flexible or perform worse on this kind of CF problems.

### 2.2 Matrix Factorization Approaches

---

Nowadays, matrix factorization models deliver the highest accuracies but they come in all shapes and sizes.

For instance, we can find two options: rating and ranking. The former is based on the continuous error between the target and the hypothesized rating, and the latter is focused on predicting the ranking at which an item is supposed to be found.

In the case of rating with MF, we have *Funk's SVD* (a.k.a *(BR)ISMF*) which is a sort of iterative version to compute the Single Value Decomposition of a matrix. Further modifications have been made to model using side information (or implicit feedback) as *SVD++*, which was developed by Yehuda Koren in 2008. Yehuda also developed a similar version named *Asymmetric SVD++*, which has some interesting properties like: fewer parameters, handling of new users, explainability

and efficient integration of implicit feedback, etc. But its accuracy is slightly lower than *SVD++*. On the other hand, there are other models based on neighborhood methods and even integrated models which combine both strategies and deliver better results.

In 2009, Yehuda Koren and his team won the Netflix using a similar version of *TimeSVD*. This algorithm takes advantage of the temporal dynamics of the ratings, resulting in a significant improvement in terms of accuracy.

Past the Netflix Prize, new approaches have shown up based on the modeling of social information like: *SocialMF* (2010), *SoRec* (2011), *TrustSVD* (2015).

In contrast, ranking modeling has gained popularity in the last years. For instance, *BPR* (2009) is a ranking model that ranks using a Bayesian analysis of the problem and side information. Then, *CLiMF* (2012) is a model for scenarios with binary relevance data, based on directly maximizing the Mean Reciprocal Rank (MRR). Finally, we can find new models that use social information such as *SBPR* (2014)

---

---

## CHAPTER 3

# Collaborative Filtering

---

Collaborative filtering (CF) is the process of filtering for information or patterns using techniques involving collaboration among multiple agents or data sources[1].

### 3.1 Types of CF

---

#### 3.1.1. Memory-based

This approach uses the entire dataset for making recommendations.

##### 3.1.1.1. Average-based

This is the most basic type CF possible so it only works with averages.

In its basic form, the model takes the mean value of all ratings in the dataset to make predictions. Although it's obvious that this model is useless for making recommendations, it can be used as a baseline to test others.

$$\hat{r}_{ui} = \mu \tag{3.1}$$

where:

$r_{ui}$  = User  $u$ 's rating of item  $i$   
 $\mu$  = Overall average rating

In order to improve this model, we can also take into account the user or item deviation. For instance, if the mean of ratings is 3.5, and the mean of the item *Titanic* is 4.0, the deviation of this item is +0.5.

$$\hat{r}_{ui} = \mu + b_i \tag{3.2}$$

where:

$r_{ui}$  = User  $u$ 's rating of item  $i$   
 $\mu$  = Overall average rating  
 $b_i$  = Deviation of item  $i$

Using the principles described above, now we can design a model that takes into account both the global average (as a reference) and the standard deviation of users and items to predict more accurately the target rating. This model is named *User-Item Baseline*.

$$\hat{r}_{ui} = \mu + b_u + b_i \quad (3.3)$$

where:

- $r_{ui}$  = User  $u$ 's rating of item  $i$
- $\mu$  = Overall average rating
- $b_u$  = Deviation of user  $u$
- $b_i$  = Deviation of item  $i$

### 3.1.1.2. User-based

Suppose each user has expressed an opinion about certain elements, therefore, if we find the similarity between the different users we can create a network of similarity on which to build our predictions.

---



---

#### Algorithm 3.1 Pseudocode for User-based CF

---



---

Identify items rated by user  $u$

Identify users that have at least one item rated in common with user  $u$  (neighborhood formation)

Compute similarity of neighbors with regard user  $u$

Select  $k$  most similar neighbors

Predict missing ratings in user  $u$  using the similarity of neighbors and their known ratings.

**return**  $\mathbf{R}$  matrix

---

To compute the similarity between users we can use whatever function we please, but it is recommended to use the *Pearson Correlation* due to its well-known good results.

$$\text{sim}(u, v) = \frac{\sum_{i \in I_{u,v}} (y_{u,i} - \hat{y}_u)(y_{v,i} - \hat{y}_v)}{\sqrt{\sum_{i \in I_{u,v}} (y_{u,i} - \hat{y}_u)^2 \sum_{i \in I_{u,v}} (y_{v,i} - \hat{y}_v)^2}} \quad (3.4)$$

where:

- $I_{u,v}$  = Item rated by user  $u$  and  $v$
- $y_{u,i}$  = Rating by user  $u$  for item  $i$

Then, we can predict ratings with:

$$y^*(u, i) = \hat{y}_u + \frac{\sum_{j \in y^*, j \neq 0} \text{sim}(v_j, u)(y_{v_j, i} - \hat{y}_{v_j})}{\sum_{j \in y^*, j \neq 0} |\text{sim}(v_j, u)|} \quad (3.5)$$

In this model, the main bottleneck is the similarity computation which is very time-consuming. Due to this, we have a two-step process for making recommendations:

1. **Offline:** Precompute and store the similarity
2. **Online:** Prediction process

### 3.1.1.3. Item-based

This model is practically the same as the *User-based* but taking the items as the reference axis.

### 3.1.2. Model-based

These models are developed using ML algorithms for data mining to find patterns based on training data. Later, these patterns are used to make predictions[5].

#### 3.1.2.1. Clustering

In simple words, it tries to group elements in such a way that all the elements inside a group are more similar between them than the elements in another group.

Once the clusters are defined, we make the recommendations at the cluster level. In addition, this is a pretty interesting technique when the number of users or items is so big that we cannot deal with them. Therefore, we can use clustering techniques to reduce the recommendation space.

However, one of its main drawbacks is that it leads to worse recommendations so we are making predictions for a set of elements instead of an individual in particular.

For example, let's suppose that we want to recommend funny things to a specific user. What do we consider as funny? Maybe we can list all the items in a database that contain the word *funny* either in its description or title, but if we do so, we are going to find that this will result in: comedians, actors, TV presenters, shows, TV series, movies, etc. One way to approach this problem is to use the above groups as clusters and then make  $k$  recommendations for each cluster. But by doing so, we lose most of the intra-cluster relations and we won't be able to find simple relations such as an actor that appears both in movie  $A$  and show  $B$  that we may like.

To solve this problem, we can use clustering algorithms to discover hidden relations between the items in the dataset and then make the recommendations upon these clusters.

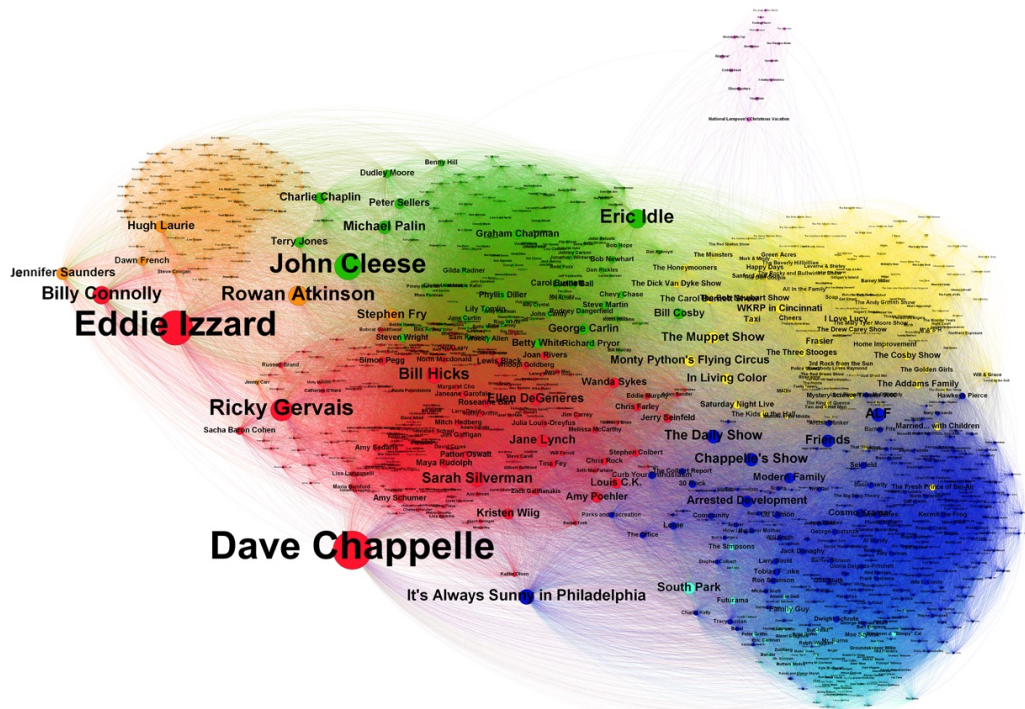


Figure 3.1: Categorization of comedy domains [29]

### 3.1.2.2. Association Rules

With this technique, we try to discover relations between variables in a dataset.

To illustrate this, let's suppose we are a supermarket trying to find a better distribution of the products in the shelves. If we know association rules, we can analyze the tickets of our customers and find relations such as  $\{\text{onions, potatoes}\} \Rightarrow \{\text{burger}\}$ . Then, we can place related items close to each other in order to ease the purchase and increase our sales.

### 3.1.2.3. Matrix Factorization

The basic idea behind matrix factorization is to find two matrices  $P$  and  $Q$  that multiplied together return a matrix  $R$ , which is the original matrix containing all the ratings.

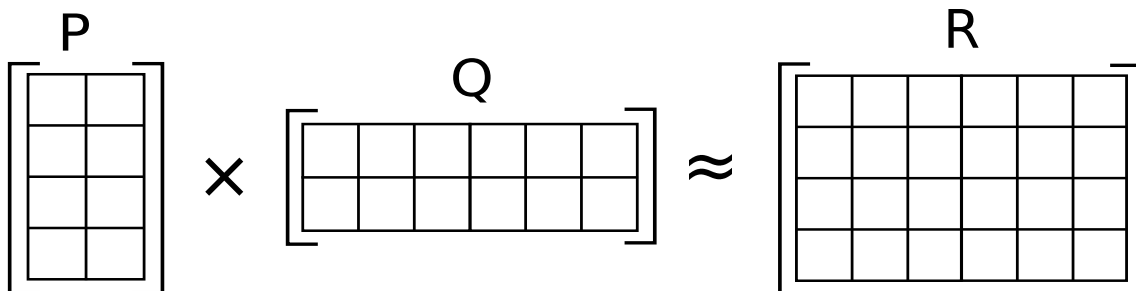


Figure 3.2: Matrix factorization



As most users have only rated a very small set of items, the sparsity of  $R_{u \times i}$  is very high. Due to this sparsity, the best approach we have is to fill  $P_{u \times k}$  and  $Q_{i \times k}$  with random numbers, and then minimize the error between the known ratings in  $R_{u \times i}$  and  $\hat{R}_{u \times i} = PQ$ .

The pseudocode for this kind of factorization models usually is something like this:

---



---

**Algorithm 3.2** Pseudocode for MF algorithms

---



---

Initialize  $\mathbf{P}^*$  and  $\mathbf{Q}^*$  with small numbers

**loop** until final condition is met

**for** every rating  $r_{ui}$  **do**

    Compute prediction  $\hat{r}_{ui} = q_i^T p_u$

    Compute error  $e_{ui}$

    Update rows  $p_u$  and  $q_i$

**end for**

**end loop**

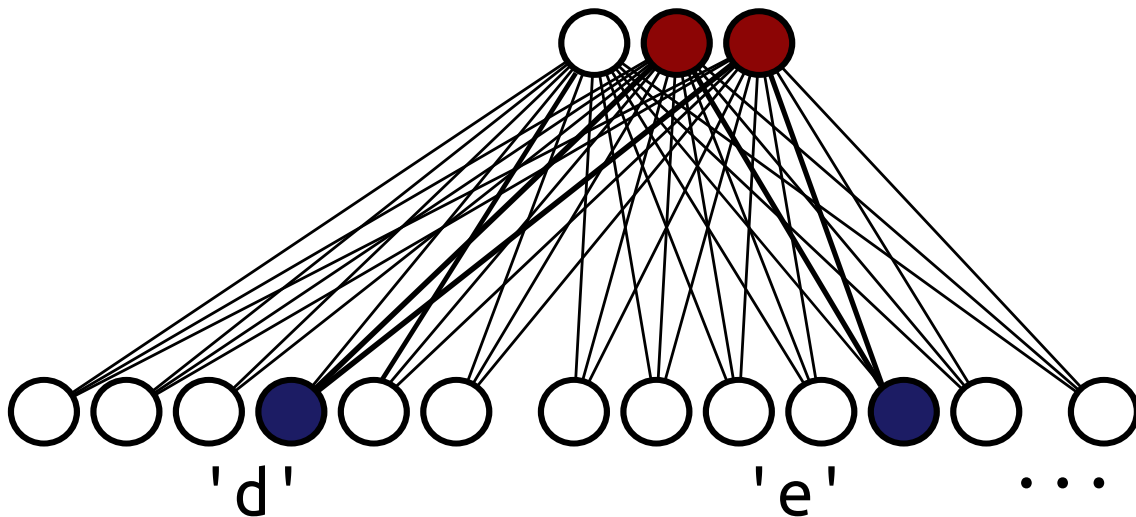
**return**  $\mathbf{P}^*$  and  $\mathbf{Q}^*$

---

Furthermore, we can model more complex behaviors by increasing the number of latent factors  $k$  and adding more matrix-features ( $Y, W, \dots$ ) for modeling aspects such as implicit information, social information, temporal dynamics,...

### 3.1.2.4. Restricted Boltzmann Machine (RBMs)

RBMs<sup>1</sup> are stochastic neural networks consisting of an input and a hidden layer where each visible unit is connected to all the hidden units in an undirected way.



**Figure 3.3:** Architecture of a RBM

They can be seen as a generative model able to find out how two set of variables (visible and hidden) are connected to each other by learning the joint probability distribution of hidden and input variables.

---

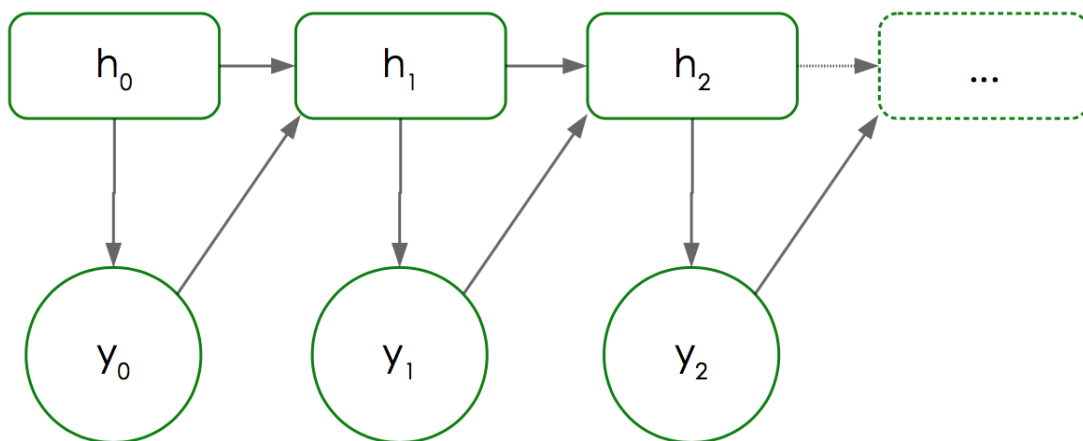
<sup>1</sup>RBMs: Restricted Boltzmann Machines

Similarly, we can see hidden units as the number of latent factors and visible units as the users' preferences.

### 3.1.2.5. Recurrent Neural Networks

Although RNNs<sup>2</sup> are also a kind of neural network as RBMs, the way they work is completely different. The connections of a RNN form a directed cycle creating a sort of internal state that can be used for modeling temporal dynamics, making RNNs especially good at modeling arbitrary sequences of inputs.

In order to exploit its benefits, we can try to predict what could be the next interesting item that a user may like, given all the previous ones he (or others) liked.



**Figure 3.4:** Architecture of a RNN for CF

As the detailed explanation of how RNNs work and are trained is out of the scope of this work, I will briefly summarize the process of recommendation. Let's imagine that we already have our network trained, now we can compute the probability distribution  $P(y_i|h_i)$  of the item  $y_i$  at the hidden state  $h_i$  and then we can simply recommend the item with the highest probability. A good idea is to use the *Softmax* distribution to output the conditional probability of each item at that given moment  $h_i$ .

## 3.2 Challenges of CF

### 3.2.1. Cold Start

Cold Start is the first and most important problem that we encounter in collaborative filtering and it is found when a system cannot infer any useful information from its users or items because it does not yet have enough information.

Although it is an intrinsic problem of information filtering approaches, it's commonly found in recommender systems.

Either if we are working with content-based or collaborative filtering approaches, a typical system has to find at least one common feature upon which construct its recommendations. If these common features don't exist, the system won't be able to provide any relevant recommendation.

<sup>2</sup>RNNs: Recurrent Neural Networks

For content-based approaches, item features are matched against user features so the model can be constructed. This feature can be obtained either explicitly (by querying the user at any given time) or implicitly (by monitoring the user's behavior). Among the drawbacks of this approach is the choice of either wasting the user's time in doing *test* or having the system in a dumb state (recommending *top-popular* or trending items) for a while until the user has provided us with enough activity to start making smart recommendations based on that.

For collaborative filtering approaches, a system has to identify rating patterns amongst users and then make recommendations based on the similarity with those users. This approach tries to profile a user by finding the tastes of like-minded ones. But similar to the content-based approaches, what do we do when no one has rated an item previously? If no further information is provided, this item will be always excluded from being recommended.

In order to solve these problems, we have lots of proposed solutions depending on the specific type of problem that we have. For instance, in content-based approaches is more common to use *active learning*<sup>3</sup> so the performance of the recommender system increases. On the other hand, in collaborative filtering approaches is more common to add new sets of data or more complex modeling, although they can still use active learning techniques. In practice, the cold start problem is often tackled by adopting hybrid approaches, this means combining the best of content-based matching and collaborative filtering.

New items can be seeded with ratings of similar items. To determine the similarity of the items, content-based information may be used and obtained through its tags or descriptions as romance, action, year,... Meanwhile, user information can be obtained querying the user with: profiling queries or personality test<sup>4</sup> or analyzing their implicit information such as social groups, browsing history, region, age,... Moreover, side information from other services can be included as a seed.

### 3.2.2. Data sparsity

Collaborative filtering is mostly used in recommender systems. As the number of users and items increase, the user-item matrix grows by its product  $O(R) = n_{users} \times n_{items}$  resulting in extremely large matrices with just a few thousand users and items.

Typically, a user is not able to rate all the items in the dataset when this is large enough. Consequently, these matrices tend to be really sparse, with levels of sparsity ranging around 95-99%. As a result of this sparsity, two things happen: [1], the cold start problem gains presence and [2], we can have a relatively small spatial cost. With regard the former, I have already described a couple solutions for it in the above section, and for the latter, we can simply use special data structures to take advantage of this property (e.g. *sparse-matrices*).

### 3.2.3. Scalability

As commented above, CF algorithms usually suffer serious scalability problems (if not designed properly) due to their n-dimensional structures.

For example, for a matrix of integers<sup>5</sup>, with a size of 1M users and 50,000 items may need 400GB of memory storage if not handled properly. Now, let's add just a single new dimension for modeling 1,000 possible context. This cube will now need 4 petabytes of memory, or what is the same, a 4,000,000GB.

<sup>3</sup>Active learning is a technique of semi-supervised learning, in which an algorithm interactively queries a user to obtain more data[2].

<sup>4</sup>A *Five Factor Model* can identify the personality characteristics for creating an initial profile.

<sup>5</sup>Normally (C/C++, gcc), an integer of x64 architecture is 8 bytes.

No matter how powerful computers are or they can be, these problems can not be solved like this. In order to do so, we have to come up with smart solutions such as tensor factorization and sparse matrices[3].

### 3.2.4. Popularity Bias

In most cases, biases are the core of the ratings, and latent factors only add (or subtract) small values to those biases in order to predict a specific rating. This doesn't mean that model-based approaches are useless (actually is the opposite), the implication of this is the huge impact that it has on the rating equation. As a result, the outcome of a recommender system for a specific item can be altered relatively easily by a small amount of strongly biased ratings, items with too many ratings, advertising campaigns, attacks,...

In order to make more robust a CF system, it's advisable to clean and review the input data giving a special treatment to cases such as:

1. **Too popular items:** Popular items tend to be recommended to users regardless whether they are related to them or not.
2. **Very few ratings:** An item with a mean rating of 5/5 that has only been rated once, cannot have the same relevance that item with 100,000 ratings and a score of 4/5.
3. **Gray sheeps:** Users whose opinions are not consistent with any group and thus do not benefit from the CF.
4. **Shilling attacks:** A user or system that consistently gives lots of positive ratings that benefit their items and negatives ones for the competitors.
5. **Advertising campaigns:** Massive advertising campaigns may have a high influence on certain items that do not apply in a long-term view.

Moreover, if possible, it is valuable to attach confidence scores with the estimated preferences so that the factorization is more robust[15].

### 3.2.5. Diversity

Sometimes it is not enough for a recommender system to be just accurate. It has to favor diversity so that new products can be discovered. Paradoxically, most CF algorithms do exactly the opposite due to the fact that they are based on past items and normally cannot discover new ones, contributing to create a *rich-get-richer* effect. As a result, diversity generally needs to be forced. Fortunately, nowadays there are many studies that have proposed several solutions to this problem along with many others such as the well-known *long-tail* recommendation[4].

## 3.3 Models for Matrix Factorization

---

### 3.3.1. Rating

#### 3.3.1.1. BRISMF

Biased Regularized Incremental Simultaneous Matrix Factorization (often shortened in BRISMF) is a factorization-based algorithm for large scale recommendation systems.

*BRISMF* has many names and variations such as: *BiasedMF*, *RISMF*, *SVD*, *Funk's SVD*,... But from a historical point of view, it was created based on the idea of *Simon Funk* for decomposing a very sparse matrix into two low-rank matrices that represent user factors and item factors of SVD[6]. This factorization can be done by using an iterative approach to minimize the loss function.

The intuition behind this idea is that for a given user  $u$  there is a vector  $p_u$  which measures the interest of the user  $u$  in the available items. Similarly, for an item  $i$  there is a vector  $q_i$  that measures the relevance of those items amongst the users. As a result of this interaction, the resulting dot product  $q_i^T \cdot p_u$  captures the interaction between user  $u$  and the item  $i$ [15]. The main advantage of this method is that by capturing the overall interest of user  $u$  in the item  $i$ , we can approximate the user  $u$ 's rating over that item  $i$ .

So that future (and more complex) models are easily understood, let's present *Funk's SVD* approach, the most basic version for factorizing matrices in an iterative way.

As explained above, we want to predict ratings as follows:

$$\hat{r}_{ui} = q_i^T p_u \quad (3.6)$$

where:

$\hat{r}_{ui}$  = Predicted rating of user  $u$  on item  $i$   
 $p_u$  = User  $u$  factor vector  
 $q_i$  = Item  $i$  factor vector

But now, the main challenge is to compute the mapping between users and items to the latent factors  $q_i, p_u \in \mathbb{R}^f$ . One way to learn the factor vectors  $p_u$  and  $q_i$  is to minimize the error between the hypothesised rating and the target ratings, we can do this by minimizing an objective function:

$$\min_{p^*, q^*} \sum_{(u, i \in k)} (r_{ui} - q_i^T p_u) \quad (3.7)$$

Although this objective function may work, in practice is too simple and can lead to pretty serious problems. As this function tries to minimize the error between the real rating and the target rating, a negative error (no matter how large) would cause the model to *think* that it is improving. However, the opposite would be happening. In addition, it is a good idea to emphasize large errors to apply a more aggressive solution. Consequently, we can solve these problems by squaring the difference and if needed, we can square root its error value to return to the original units.

Besides squaring the error, we have to take care of the overfitting which in simple words is when the complexity of a model allows to *memorize* the training set instead of *learning* a solution from it. To overcome this problem we can apply regularization techniques, and for this kind of models, the *L2* regularization turns to be the best due to its differentiable properties and how it shrinks the coefficients without eliminating them as oppose to *L1*.

Therefore, we can now transform our previous (*silly*) model to a functional one: (*The constant  $\lambda$  controls the extent of regularization*)

$$\min_{p^*, q^*} \sum_{(u, i \in k)} (r_{ui} - q_i^T p_u)^2 + \lambda (\|p_u\|^2 + \|q_i\|^2) \quad (3.8)$$

Despite the fact that this model can deliver good results, we want to make even better. One problem that it has right now is that the latent factors  $P$  and  $Q$  are responsible for the whole rating

$\hat{r}_{ui}$  and because of that, some ratings can be completely out of range. To solve this, we can initialize a rating  $r_{ui}$  as: (it's the same process as the User-Item baseline model described before 3.1.1.1)

$$r_{temp_{ui}} = \mu + \sigma_u + \sigma_i \quad (3.9)$$

where:

$r_{temp_{ui}}$  = Temporal user  $u$ 's rating of item  $i$   
 $\mu$  = Overall average rating  
 $\sigma_u$  = Standard deviation of user  $u$   
 $\sigma_i$  = Standard deviation of item  $i$

With this temporal rating,  $P$  and  $Q$  will learn less aggressively leading to better results. Now we can combine the above-presented ideas to construct the *BRISMF* model as follows:

$$\begin{aligned} \hat{r}_{ui} &= \mu + \sigma_u + \sigma_i + q_i^T p_u \\ &= \mu + b_u + b_i + q_i^T p_u \end{aligned} \quad (3.10)$$

where:

$\hat{r}_{ui}$  = Predicted rating of user  $u$  on item  $i$   
 $\mu$  = Overall average rating  
 $b_u$  = Standard deviation user  
 $b_i$  = Standard deviation item  
 $q_i^T p_u$  = user-item interaction

The final model (*BRISMF*) learns, as the others, by minimizing the squared error function:

$$\min_{p^*, q^*, b^*} \sum_{(u, i \in k)} (r_{ui} - \mu - b_u - b_i - q_i^T p_u)^2 + \lambda (\|p_u\|^2 + \|q_i\|^2 + b_u^2 + b_i^2) \quad (3.11)$$

For minimizing the objective function there are two *by default* options: stochastic gradient descent (SGD) and alternating least squares (ALS). Both of them will be discussed in the following sections but for now, we only need to know that the modification of the parameters is done by moving the current solution in the opposite direction of the gradient:

In this case, the partial derivatives of  $J(\theta)$ <sup>6</sup> yields to:

- $b_u \leftarrow b_u + \gamma_1 \cdot (e_{ui} - \lambda_1 \cdot b_u)$
- $b_i \leftarrow b_i + \gamma_1 \cdot (e_{ui} - \lambda_1 \cdot b_i)$
- $p_u \leftarrow p_u + \gamma_2 \cdot (e_{ui} \cdot q_i - \lambda_2 \cdot p_u)$
- $q_i \leftarrow q_i + \gamma_2 \cdot (e_{ui} \cdot p_u - \lambda_2 \cdot q_i)$

where:

$\gamma$  = Denotes the learning rate  
 $\lambda$  = Denotes the regularization factor  
 $e_{u,i}$  = Denotes the absolute error between the hypothesis and the target value ( $e_{u,i} = r_{u,i} - q_i^T p_u$ ).

---

<sup>6</sup> $J(\theta)$ : Cost function

Finally, it is important to point out two things: First,  $P$  and  $Q$  are matrices that must be initialized with small random numbers to break the symmetry and ease the convergence. If not done, many parameters would receive the same updates and thus not reduce the error. And second, it is not mandatory to consider the bias as a learning parameter, but we do so, the accuracy of the model is going to increase because the additional parameters are learned in the context of other parameters that compose the objective function. Besides, we can set the regularization factor to zero and we will obtain the same results as before.

### 3.3.1.2. SVD++

In many scenarios, we have explicit and implicit information available in a dataset. Nevertheless, the latter is not available most of the time and also can be really hard to get. A classical example of implicit information can be a dataset with a search history per user, which can tell us the user's preferences or even the login times. What's more, sometimes we have less obvious but fantastic sources of implicit information in front of us and we don't notice. An example of this could be a dataset with only explicit ratings. The simple act of rating an item with a numeric score is giving us information about *which* items the user has reviewed (binary information; implicit) and *how* they rated these items (score value; explicit). Once we know that, we can now design better models that account for both explicit and implicit information. Hence, we can get more *juice* from our datasets allowing us to increase the prediction accuracy of the previous models.[7, 15]

In order to solve this problem, *Yehuda Koren* came in 2008 with the idea of *SVD++*, which is an enhanced version of the classical *Funk's SVD* to support the use of implicit feedback information[7].

$$\hat{r}_{ui} = \mu + b_u + b_i + \left( p_u + \frac{1}{\sqrt{|N(u)|}} \sum_{j \in N(u)} y_j \right)^T q_i \quad (3.12)$$

where:

- $\hat{r}_{ui}$  = Predicted rating of user  $u$  on item  $i$
- $\mu$  = Overall average rating
- $b_u$  = Standard deviation user
- $b_i$  = Standard deviation item
- $N(u)$  = Set of all items for which  $u$  provided an implicit preference
- $p_u$  = User-factors vector
- $q_i$  = Item-factors vector
- $y_j$  = Implicit-factors vector

Unlike previous models, in *SVD++* each item  $i$  is associated with three factor vectors  $q_i, p_u, y_j \in \mathbb{R}^f$ . Model parameters are learnt by minimizing the corresponding squared error loss function:

$$\min_{p^*, q^*, y^*, b^*} \sum_{(u, i \in k)} (r_{ui} - \mu - b_u - b_i - q_i^T \left( p_u + \frac{1}{\sqrt{|N(u)|}} \sum_{j \in N(u)} y_j \right))^2 + \lambda (b_u^2 + b_i^2 + \|p_u\|^2 + \|q_i\|^2 + \sum_{j \in N(u)} \|y_j\|^2) \quad (3.13)$$

To train the model using *SGD* we have to loop through all the ratings, modifying the parameters by moving in the opposite direction of the gradient. Hence, the partial derivatives of  $J(\theta)$  yields to:

- $b_u \leftarrow b_u + \gamma_1 \cdot (e_{ui} - \lambda_1 \cdot b_u)$

- $b_i \leftarrow b_i + \gamma_1 \cdot (e_{ui} - \lambda_1 \cdot b_i)$
- $p_u \leftarrow p_u + \gamma_2 \cdot (e_{ui} \cdot q_i - \lambda_2 \cdot p_u)$
- $q_i \leftarrow q_i + \gamma_2 \cdot (e_{ui} \cdot (p_u + |N(u)|^{-\frac{1}{2}} \sum_{j \in N(u)} y_j) - \lambda_2 \cdot q_i)$
- $\forall j \in N(u) :$   
 $y_j \leftarrow y_j + \gamma_2 \cdot (e_{uj} \cdot |N(u)|^{-\frac{1}{2}} \cdot q_i - \lambda_2 \cdot y_j)$

Looking at the cost function, the inversion of  $|N(u)|^{-\frac{1}{2}}$  is simply a mechanism for normalization. But the reason behind the square root is still unclear. One of the best explanation that I currently have is related with the radical properties, so the less frequent a user or item is, the stronger the penalization or the control over their parameters will be.

### 3.3.1.3. TimeSVD

We can improve SVD++ in order to model temporal dynamics. In order to do so, we can think of classical time-window or instance-decay approaches, but they don't usually work because they lose too much signal when discarding data instances. So that we can get better distinctions between transient effects and long-term patterns, *Yehuda Koren* came up in 2009 with *TimeSVD*. These models tracks the time changing behavior throughout the life span of the data, allowing us to exploit relevant components of all data instances, while discarding only what is modeled as being irrelevant[9].

As an intuition, it's a sort of SVD++ that uses framed information in a fixed set of temporals bins. In such a way that the predictions are compute as:

$$\hat{r}_{ui}(t) = \mu + b_u(t) + b_i(t) + \left( p_u(t) + \frac{1}{\sqrt{|N(u)|}} \sum_{j \in N(u)} y_j \right)^T q_i \quad (3.14)$$

*Note: This model will be implemented in a future version of our library due to temporal constraints.*

### 3.3.1.4. TrustSVD

So far we have introduced a very simple model, on which we have been introducing more and more modifications to model increasingly complex behaviors.

Therefore, I present *TrustSVD*, which is trust-based matrix factorization technique. What this means is that it takes into account both the explicit and implicit influence of rated items as well as the explicit and implicit influence of trusted users on the prediction of items for active user items[8]. Or in simple terms, it uses *friendship* information to improve SVD++.

This model is build on top of SVD++ so to simplify things, we can say that *TrustSVD* is the combination of the classical SVD++ plus a modified version of SVD++ to model trust information.

In order to understand how this model works, first we need to know the premises on which it is built[8]:

1. *Trust information is very sparse, yet is complementary to rating information.*
2. *A user's ratings have a **weakly positive correlation** with the average of her social neighbors under the concept of trust-alike relationships, and a **strongly positive correlation** under the concept of trust relationships.*



First, let's suppose that we want to decompose a matrix  $R_{m \times n}$  (as *Funk's SVD*), where  $m$  denotes users and  $n$  denotes items. So we have to find two low rank matrices  $R \approx P^\top Q$  where  $P \in \mathbb{R}^{d \times m}$  and  $Q \in \mathbb{R}^{d \times n}$  (being  $d$  the number of latent factors). The resulting loss function is:<sup>7</sup>

$$\mathcal{L}_r = \sum_{(u,i \in k)} (r_{u,i} - q_i^\top p_u)^2 + \lambda (\|p_u\|_F^2 + \|q_i\|_F^2) \quad (3.15)$$

Secondly, now we have to do pretty much the same but to decompose a trust-matrix  $T = [t_{u,v}]_{m \times m}$ , where  $u$  denotes users and  $t_{u,v}$  the extent to which users  $u$  trust user  $v$ [8]. Similarly to the previous rating loss function, we have to find another two low rank matrices  $T \approx P^\top W$  where  $P \in \mathbb{R}^{d \times m}$  and  $W \in \mathbb{R}^{d \times m}$  (being  $d$  the number of latent factors). The resulting loss function is:

$$\mathcal{L}_t = \sum_{(u,v \in k)} (t_{u,v} - w_v^\top p_u)^2 + \lambda (\|p_u\|_F^2 + \|w_v\|_F^2) \quad (3.16)$$

The next stage is to transform the above loss functions to account for implicit information as *Koren* shown in 2008 with his model *SVD++* and also including the implicit effect of trusted users on item ratings using the same techniques (3.3.1.2). Hence, the rating equation will result as follows:<sup>8</sup>

$$\hat{r}_{ui} = \mu + b_u + b_i + q_i^\top \left( p_u + |I_u|^{-\frac{1}{2}} \sum_{i \in I_u} y_i + |T_u|^{-\frac{1}{2}} \sum_{v \in T_u} w_v \right) \quad (3.17)$$

where:

- $\hat{r}_{ui}$  = Predicted rating of user  $u$  on item  $i$
- $\mu$  = Overall average rating
- $b_u$  = Standard deviation user
- $b_i$  = Standard deviation item
- $p_u$  = User-factors vector
- $q_i$  = Item-factors vector
- $w_v$  = Trustee-factors vector
- $y_i$  = Implicit-factors vector
- $I_u$  = Set of items rated by user  $u$
- $T_u$  = Set of users trusted by user  $u$

With regard the final loss function, it will be the sum of the two previously commented loss function:

$$\begin{aligned} \mathcal{L} &= \mathcal{L}_r + \mathcal{L}_t + [\text{reg.params}] \\ &= \sum_{(u,i \in k)} (r_{u,i} - q_i^\top p_u)^2 + \sum_{(u,v \in k)} (t_{u,v} - w_v^\top p_u)^2 + [\text{reg.params}] \end{aligned} \quad (3.18)$$

<sup>7</sup> $F$  denotes the Frobenius norm

<sup>8</sup>The product  $q_j^\top w_u$  can be seen as the influence of trustees on the rating prediction

The regularization strategy is based on high regularizations for cold-start users and niche items and small penalizations for popular users and items due to their smaller chance of being overfitted. Finally, taking these things into account and other small modifications we obtain:

$$\begin{aligned}
\mathcal{L} = & \frac{1}{2} \sum_u \sum_{j \in I_u} (\hat{r}_{u,j} - r_{u,j})^2 + \frac{\lambda_t}{2} \sum_u \sum_{v \in T_u} (\hat{t}_{u,v} - t_{u,v})^2 \\
& + \frac{\lambda}{2} \sum_u |I_u|^{-\frac{1}{2}} b_u^2 + \frac{\lambda}{2} \sum_j |U_j|^{-\frac{1}{2}} b_j^2 \\
& + \sum_u \left( \frac{\lambda}{2} |I_u|^{-\frac{1}{2}} + \frac{\lambda_t}{2} |T_u|^{-\frac{1}{2}} \right) \|p_u\|_F^2 \\
& + \frac{\lambda}{2} \sum_j |U_j|^{-\frac{1}{2}} \|q_j\|_F^2 + \frac{\lambda}{2} \sum_i |U_i|^{-\frac{1}{2}} \|y_i\|_F^2 \\
& + \frac{\lambda}{2} |T_v^+|^{-\frac{1}{2}} \|w_v\|_F^2
\end{aligned} \tag{3.19}$$

where:

- $U_j, U_i$  = Set of users who rated items  $j$  and  $i$
- $t_{u,v}$  = User  $u$  that trust  $v$ .
- $I_u$  = Set of items rated by user  $u$
- $T_v^+$  = Set of users who trust user  $v$
- $T_u$  = Set of users trusted by user  $u$
- $\|\cdot\|_F$  = Denotes the Frobenius norm

To optimize this model, its partial derivatives are:

- $\frac{\partial \mathcal{L}}{\partial b_u} = \sum_{j \in I_u} e_{u,j} + \lambda |I_u|^{-\frac{1}{2}} b_u$
- $\frac{\partial \mathcal{L}}{\partial b_j} = \sum_{u \in U_j} e_{u,j} + \lambda |U_j|^{-\frac{1}{2}} b_j$
- $\frac{\partial \mathcal{L}}{\partial p_u} = \sum_{j \in I_u} e_{u,j} q_j + \lambda_t \sum_{v \in T_u} e_{u,v} w_v + (\lambda |I_u|^{-\frac{1}{2}} + \lambda_t |T_u|^{-\frac{1}{2}}) p_u$
- $\frac{\partial \mathcal{L}}{\partial q_j} = \sum_{u \in U_j} e_{u,j} (p_u + |I_u|^{-\frac{1}{2}} \sum_{i \in I_u} y_i + |T_u|^{-\frac{1}{2}} \sum_{v \in T_u} w_v) + \lambda |U_j|^{-\frac{1}{2}} q_j$
- $\forall i \in I_u, \frac{\partial \mathcal{L}}{\partial y_i} = \sum_{j \in I_u} e_{u,j} |I_u|^{-\frac{1}{2}} q_j + \lambda |U_i|^{-\frac{1}{2}} y_i$
- $\forall i \in T_u, \frac{\partial \mathcal{L}}{\partial w_v} = \sum_{j \in I_u} e_{u,j} |T_u|^{-\frac{1}{2}} q_j + \lambda_t e_{u,v} p_u \lambda |T_v^+|^{-\frac{1}{2}} w_v$

### 3.3.2. Ranking

#### 3.3.2.1. BPR

*BPR* is a model that uses a bayesian personalized ranking from implicit feedback in order to make recommendations.

This model uses a generic optimization criterion *BPR-Opt* for personalized ranking that is the maximum posterior estimator derived from a Bayesian analysis of the problem[10].

*Note: As TimeSVD, this model will be implemented in a future version of our library due to temporal constraints.*

### 3.3.2.2. CLiMF

Collaborative Less-is-More Filtering (often shortened as CLiMF) is a matrix factorization model used in scenarios with binary relevance data.

In simple terms, it's been designed for scenarios where a user has had (or not) a binary action over a specific item such as un/watched, un/liked, un/read,...

Apart from the binary relevance, there is another thing to take into account... It's a ranking model! What this means is that it doesn't care about the predicted score itself but the position at which the item was recommended. To illustrate the point, a good example is the *Amazon's* recommender system where showing you the right item is more important than the accuracy of which they did. Although the goal of all recommender systems is to recommend you the best items, in many services the most important measure in the sorting position.

For instance, let's suppose that we have two items that you may like: item *A* with predicted score of 3.89 and item *B* with 3.90. Clearly, both items have practically the same score and the scoring difference between them could be due to small nuances produced by the input data, the tuning of the model, the initial seed,... But when these two items have to be sorted, this negligible difference is very important because one of them is going to be at the *i*-th position and the other at the (*i*-th + 1) position.

As a result of this, *CLiMF* tries to improve top-*k* recommendations through ranking by directly maximizing the Mean Reciprocal Rank (MRR).

$$RR_i = \sum_{j=1}^N \frac{Y_{i,j}}{R_{i,j}} \prod_{k=1}^N (1 - Y_{i,j} \mathbb{I}(R_{i,k} < R_{i,j})) \quad (3.20)$$

where:

- $RR_i$  = Reciprocal Rank (RR) of a ranked list for user *i*
- $R_{i,j}$  = Denotes the rank of item *j* in the ranked list of items for user *i*
- $Y_{i,j}$  = Denotes the binary relevance score of item *j* to user *i*
- $\mathbb{I}(x)$  = Function that indicates whether *x* is *True* or *False*

The first problem that we encounter is that ranking implies integer values and discrete optimizations are simply unfeasible. In order to overcome this problem, the authors decided to transform the non-smoothed loss function to a smoothed one. To achieve so, they use a logistic function to derive an approximation for  $\mathbb{I}(x)$ :

$$\mathbb{I}(R_{i,k} < R_{i,j}) \approx g(f_{i,k} - f_{i,j}) \quad (3.21)$$

where:

- $g(t)$  = Sigmoid function:  $g(t) = \frac{1}{1+e^{-t}}$
- $f_{i,j}$  = Predictor function that maps user *u* and item *i* to a relevance score

Despite the fact that the loss function is smoothed, there are more problems that we need to address such as the intractability of the gradient  $O(N^2)$ . The authors solved this second problem by deriving a lower bound based on the *Jensen's inequality* (3.23) and the monotonic properties of the logarithmic functions (3.22) so that the loss function is concave.

$$\ln\left(\frac{1}{n_i^+} RR_i\right) \quad (3.22)$$

$$f(tx_1 + (1-t)x_2) \leq tf(x_1) + (1-t)f(x_2) \quad (3.23)$$

After modifying the previous loss function by the techniques described above, we obtain the final loss function. A continuous function that is differentiable and convex:<sup>9</sup>

$$\begin{aligned}
 F(U, V) = & \sum_{i=1}^M \sum_{j=1}^N Y_{ij} [\ln g(U_i^T V_j) \\
 & + \sum_{k=1}^N \ln(1 - Y_{ik} g(U_i^T V_k - U_i^T V_j))] \\
 & - \frac{\lambda}{2} (\|U\|^2 + \|V\|^2)
 \end{aligned} \tag{3.24}$$

where:

$U$  = User-feature matrix  
 $V$  = Item-feature matrix

Now we can use stochastic gradient ascent to maximize the objective function  $F(U, V)$ . Its corresponding partial derivatives are:

- $\frac{\partial F}{\partial U_i} = \sum_{j=1}^N Y_{ij} [g(-f_{ij}) V_j + \sum_{k=1}^N \frac{Y_{ik} g'(f_{ik} - f_{ij})}{1 - Y_{ik} g(f_{ik} - f_{ij})} (V_j - V_k)] - \lambda U_i$
- $\frac{\partial F}{\partial V_j} = Y_{ij} [g(-f_{ij}) + \sum_{k=1}^N Y_{ik} g'(f_{ij} - f_{ik}) (\frac{1}{1 - Y_{ik} g(f_{ik} - f_{ij})} - \frac{1}{1 - Y_{ik} g(f_{ij} - f_{ik})})] U_i - \lambda V_j$

The matrices  $U$  and  $V$  should be randomly initialized with small values (close to zero with a standard deviation equal to one) in the exact same way as the models previously described.

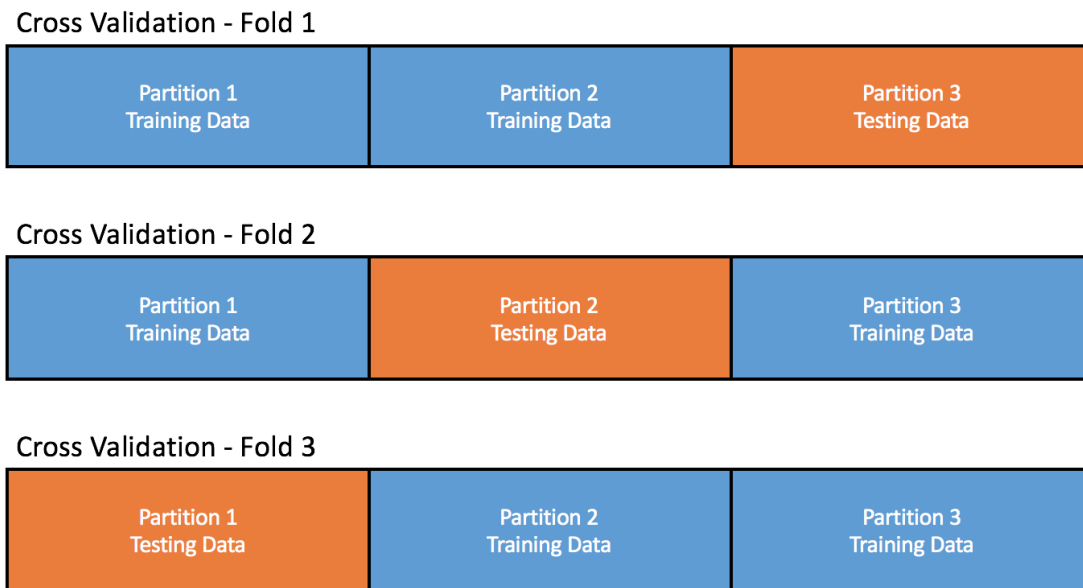
Moreover, it's worth to mention that the product of  $U_i \times V_j$  indicates the relevance score of user  $i$  in the item  $j$ . In order to get the top- $k$  best items, we have to sort in descending order the values of product  $U_i \times V^T$  but preserving their original index as the item reference. Then we simply recommend the first  $k$  values.

### 3.4 Evaluation and testing

First of all, we have to see the matrix factorization problem as a regression problem. The described matrix factorization models in this work are all statistical models. The models try to estimate relationships among variables (users, items, friends,..) so that we can make predictions of unknown inputs. Unfortunately, training a model is just the beginning of a very long process of evaluation and testing because once the estimations are obtained we need a validation model. This validation model will allow us to decide whether results quantifying hypothesized relationships obtained from regression analysis are acceptable or not.

In collaborative filtering we usually deal with lots of data so cross-validation processes are the way to go. With cross-validation we can assess how the results obtained with a model will generalize to an independent dataset. The process basically consists in splitting the dataset in  $k$  parts, using  $k-1$  folds to train the model and the remaining part to validate it. We repeat the process  $k$  times, each time using another part for the validation. Additionally, it is convenient to use a completely different dataset to test the model after the cross-validation so the resulting scoring is more robust.

<sup>9</sup>In practice these functions are non-convex but due to other properties can be considered as they were.



**Figure 3.5:** Cross-validation process [33]

Basically, we try to know how good is the design and tuning of our model. If the measured error in the training set is significantly lower than the one in the validation or testing set, our model probably needs some modifications. Once, we've got a final model, we should use all the available data to train the model.

So far everything seems okay, but the evaluation and testing in matrix factorization models are slightly different than most regression models. Why? Because when removing information from our data set, we may lose the shape of the final matrix.

For example, let's suppose that we want to train a recommender system with 10 users and 7 items, using the following dataset:

**Table 3.1:** Toy dataset

	User ID	Item ID	Rating
1	5	6	3.5
2	2	1	4.0
3	6	4	3.0
4	1	4	4.0
5	7	5	3.5
6	8	2	2.0
7	4	3	3.0
8	4	7	5.0
9	1	2	2.5
10	9	7	4.5

The matrix  $R$  has a shape  $(10, 7)$ . If we perform cross-validation on a *blank-model*<sup>10</sup> with a  $k = 3$  the resulting shapes of  $R$  will be something like this:

- **Fold 1:** trained with rows: 1-7;  $R$  shape:  $(8, 6)$
- **Fold 2:** trained with rows: 4-10;  $R$  shape:  $(9, 7)$

<sup>10</sup>Not pretrained

- **Fold 3:** trained with rows: 1-3 and 8-10;  $R$  shape: (9, 7)

It is not hard to see that the model is going to have problems during the validation part of the first-fold. As the model has been trained to make recommendations up to the user 8-th and the 6-th item, as soon as the model is asked for user 9 in the validation set, the model is going to have problems.

In order to solve these problems we have several options:

- Set the shape of  $R$  before the cross-validation
- Ignore that rating
- Return the overall average rating
- Return the user-item baseline rating ( $r_{ui} = \mu + \sigma_u + \sigma_i$ )

Depending on the case we may apply one solution or another. But a strong point in favor of the last solution is that if at least we know the user or the item we're trying to predict, the model can make use of the known information to improve the accuracy even in situations where part of the data is missing.

### 3.4.1. Rating

Models that use numeric ratings need regression metrics. To assess the model, a function measures the prediction error given a ground truth and the predicted value. There are many metrics to assess the rating models as:

- Root Mean Squared Error (RMSE)
- Relative Squared Error (RSE)
- Mean Absolute Error (MAE)
- Relative Absolute Error (RAE)
- Coefficient of Determination ( $R^2$ )

To the best of my knowledge, *RMSE* is the most common metric<sup>11</sup> for rating models. Additionally, there are popular benchmarks which also provide the *MAE* measure but it is not as common as *RMSE*.

Back in 2008, during the Netflix challenge, *Yehuda Koren* and other contestants observed that a solution with a slightly better *RMSE* can lead to completely different results and better recommendations[7].

Based on this observation, *Yehuda K.* developed a new evaluation metric based on a *top-k recommendation*. The evaluation metric works as follows: For each movie  $i$  rated 5-stars by user  $u$  we select additional 1,000 random movies and then we predict the ratings by user  $u$  for each one of the 1,001 movies. As the 1,000 movies were random, statistically we know that most of them are of no interest to the user  $u$  so we hope that movie  $i$  will precede the rest 1,000 once they are sorted by rating. In this case there are 1,001 different possible ranks for movie  $i$  so the closer to the 1-st position, the better (percentile 100% - any other movie above it).<sup>12</sup>

<sup>11</sup>In the scientific papers

<sup>12</sup>Example extracted from the *SVD++*, *Yehuda K. 2008* paper [7]

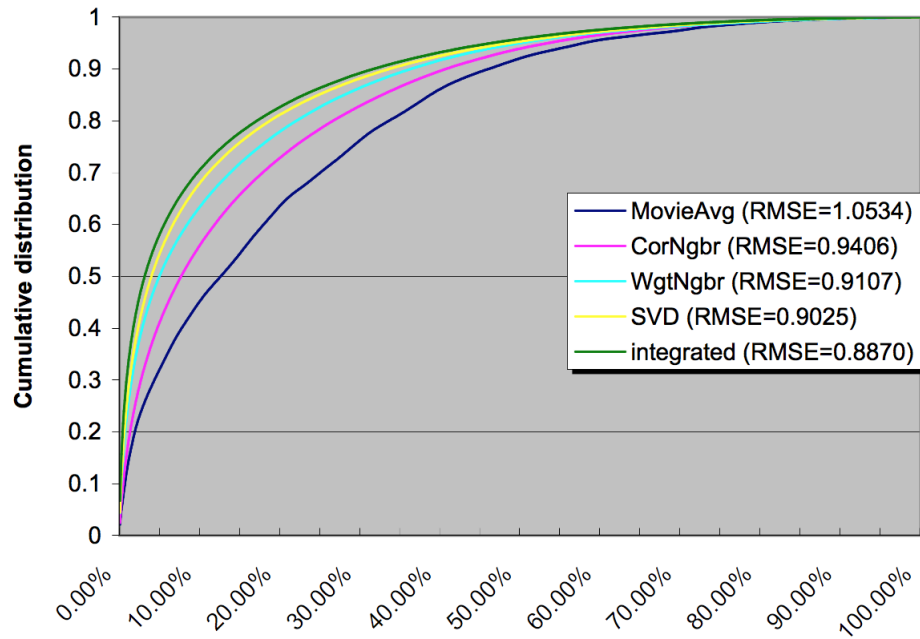


Figure 3.6: Evaluation metrics (RMSE) [7]

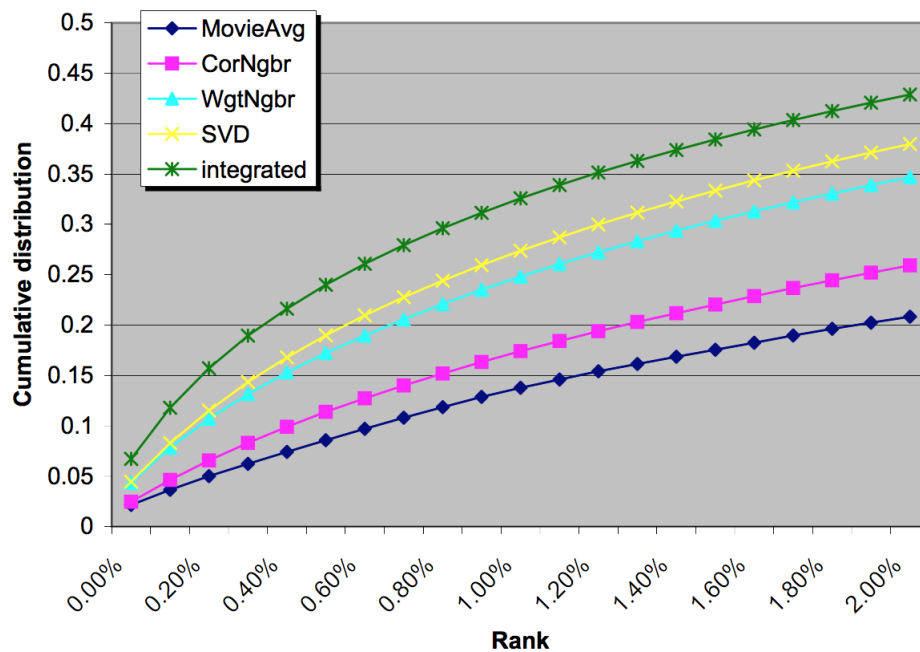


Figure 3.7: Evaluation metrics (top-k rec.) [7]

The main advantage of this method is that it evaluates through a top-K recommender so it sharpens the differences between models, giving better and more accurate picture of the performance than traditional evaluation metrics can not provide.

### 3.4.2. Ranking

Ranking is a central part of many information retrieval problems[32], such as document retrieval, collaborative filtering, sentiment analysis, online advertising, machine translation, computational biology, and many others.

Ranking metrics assess how well a model is performing with regard certain parameters and intrinsic constraints such as the natural values that an object can hold when ranked. In simple words, when a F1 overtakes another, it takes the position of the overtaken car. (It would be a non-sense to be at the position 2.76)

Hence, as opposed to rating models in which the error between the original rating and the predicted one matters, in ranking models it doesn't (at least not directly). As a result of this, ranking metrics are going to be focused on the relative position of an item within a defined set.

There are many ranking measures, but amongst the most popular we found:

- Mean Reciprocal Rank (MRR)
- Mean Average Precision (MAP)
- Discounted Cumulative Gain (DCG)
- Normalized Discounted Cumulative Gain (NDCG)

In the area of collaborative filtering, there are two non-exclusive standard measures *MRR* and *MAP*, as it happened with the evaluation of rating models. Additionally, the *top-k recommender* technique introduced by *Yehuda K.* and described in the previous section might be applied too. (See 3.4.1).

## 3.5 Optimization

---

The optimization process is a crucial component for this topic that cannot be separated from the models. Maybe we have an astonishing and super fancy model but if this model cannot be trained in a reasonable amount of time then it's useless.

In this section, the discussion will be focused on the well-known gradient based methods. Thus, I will explain how they work, why they can be used for matrix factorization, modifications to gain robustness and how to visualize their behavior under different circumstances.

### 3.5.1. Gradient descent

The objective function is the function that we want to either maximize or minimize. So that its parameters can be optimized using first-order optimization methods, it has to be differentiable and convex. Without these properties, the optimization process cannot be guaranteed.

Typically, gradient descents approaches are the *by default* option for most model-based approaches in CF since it combines an easy implementation with a relatively fast running time. Gradient descent finds the local minimum of a function by taking steps that are proportional to the opposite direction of the gradient of the function at the current point.

Let's illustrate this with a mathematical description.  $J(\theta)$  is the cost function and  $\frac{\partial J(\theta)}{\partial \theta_j}$  the partial derivative w.r.t  $\theta_j$ . Therefore,

$$J_{train}(\theta) = \sum_{i=1}^m (h_{\theta}(x^{(i)}) - y^{(i)})^2 \quad (3.25)$$



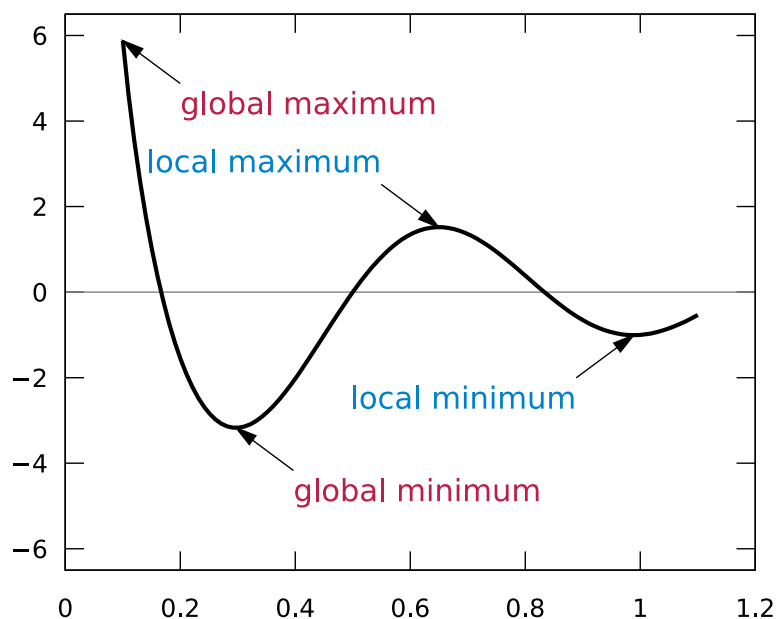
and this:

$$\theta_j := \theta_j - \alpha \sum_{i=1}^m (h_\theta(x^{(i)}) - y^{(i)}) x_j^{(i)} \quad (3.26)$$

...is the equation to move the solution in the opposite direction of the gradient so that  $J$  can be minimized. This last step must be repeated until the solution has converged or until some termination criteria is met.

Likewise, if the steps are taken in the direction of the gradient and proportional to it, the method will find a local maximum. This procedure is known as gradient ascent. Depending on the problem, sometimes is easier to just multiply the objective function by  $-1$  and apply gradient descent.

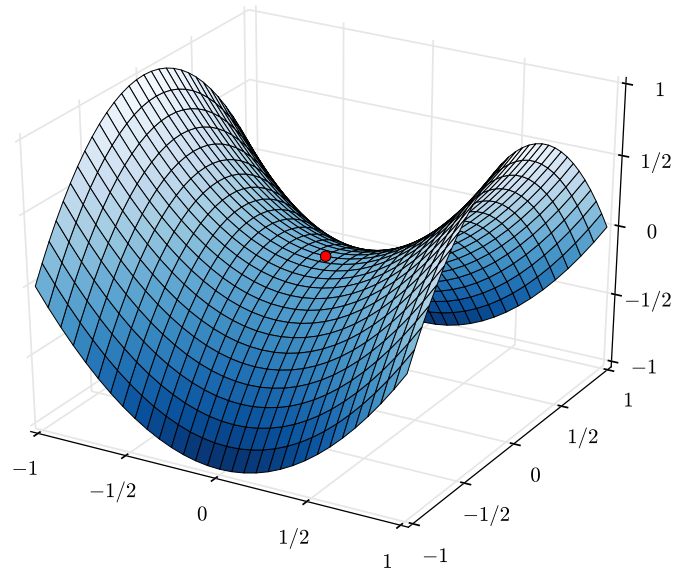
Gradient descent presents several problems related with slow convergence rates and getting stuck in suboptimal solutions. The latter is produced because gradient descent might find a local minima instead of the global minima. The nuance might seem small but it has huge implications in practice. For example, if we are trying to minimize the parameters of a function we can get stuck in a local minima (suboptimal solution) instead of finding the global minima (optimal solution):



**Figure 3.8:** Local and global maxima and minima [34]

Additionally, we find saddle points: A *saddle point* is a point in the domain of a function where the slopes (derivatives) of orthogonal function components defining the surface become zero (a stationary point) but are not a local extremum on both axes.<sup>13</sup> One of the implications of these points in gradient descent methods is that the solution might get stuck as the derivative is close to zero, being unable to escape.

<sup>13</sup>Mathematical definition [35]



**Figure 3.9:** A saddle point on the graph of  $z = x^2y^2$  (in red) [35]

To overcome this problem, we can modify the gradient descent method so it gains robustness against saddle points. Later we will discuss these modifications in detail but a simple way to make gradient descent approaches more robust to saddle points is to add a sort of linear momentum to it. Adding momentum means that the current solution has kind of *inertia* that will push it away from the saddle point.

### 3.5.2. Optimizers for gradient descent

#### 3.5.2.1. Stochastic Gradient Descent (SGD)

Stochastic gradient descent is a stochastic approximation of the gradient descent optimization method. The main advantage of this method is that it converges significantly faster than gradient descent. Although the taken step might not be optimal, it usually converges faster and making the pertinent corrections at each new step.

Mathematically is defined exactly as the deterministic version of gradient descent, but the key difference is that instead of looping through the entire dataset to make one update of  $j$ , we make one update per sample.

$$\theta_j := \theta_j - \alpha(h_{\theta}(x^{(i)}) - y^{(i)})x_j^{(i)} \quad (3.27)$$

Similarly, we can write the pseudo code as follows:

---



---

#### Algorithm 3.3 Pseudocode for SGD

---



---

Initialize the vector of parameters  $\theta$  and set learning rate  $\alpha$

Repeat until the termination criteria is met

Shuffle the set of samples (optional, but recommended)

**for**  $i = 1, 2, \dots, m$  **do**

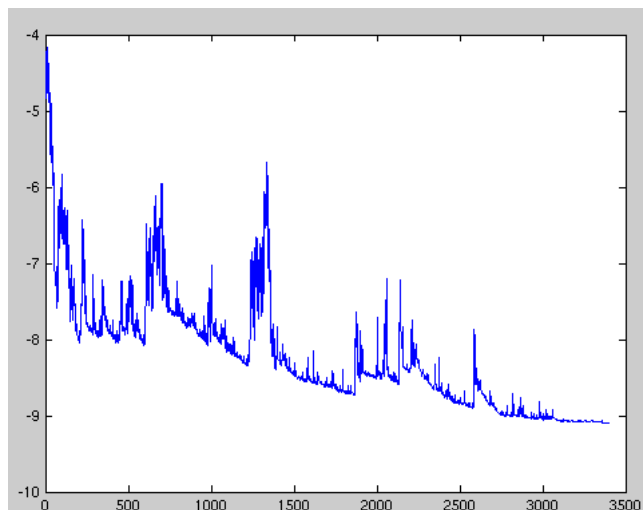
$\theta := \theta - \alpha \nabla J_i(\theta)$

**end for**

---

As it is been said before, *SGD* may not take the right step towards the minimum at given point but we know that statistically speaking, in future iterations it will go back towards to minimum converging faster than gradient descent. As a result of this stochastic behavior, the loss function might fluctuate over the time (iterations) creating the appearance that the solution is diverging at a certain point when in reality is not, namely, just *noise*. In addition, this event might even happen because the optimizers applied to SGD are pushing the current solution away in order to escape from a local minima or simply because of the momentum.

To illustrate this point, just take a look at the image below. Although there are strong fluctuations during the training, we can see that at the same time the solution is converging towards a better local minima.



**Figure 3.10:** Fluctuation during a SGD optimization

Even though the benefits of *SGD* are pretty clear, sometimes we may want to get a compromise between the true gradient of the deterministic gradient descent and the fast converge rate of the stochastic approaches at a single sample. Fortunately, we can get the best of both approaches by using mini-batches. This means to use batches of  $b$  samples (with  $b < m$ ) taken from the original dataset with  $m$  samples, to compute the gradient against more than one training example.

For  $i$  to  $m$ , step  $b$ :

$$\theta_j := \theta_j - \alpha \sum_{k=1}^{i+(b-1)} (h_{\theta}(x^{(k)}) - y^{(k)})x_j^{(k)} \quad (3.28)$$

Likewise, it's interesting to mention that in most real life problems there is no way to find a global minimum so stochastic approaches usually outperform the classical gradient descent in terms of accuracy and time.

### 3.5.2.2. Momentum

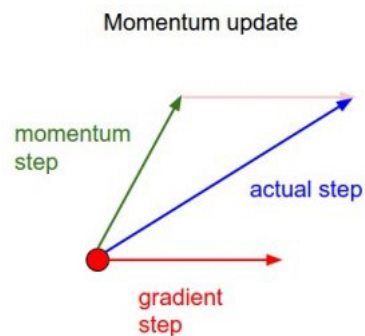
The optimization process through gradient descent methods is hard and usually very complex, so many improvements on it have been proposed to make it easier and less painful. Typical problems relate with the learning parameter  $\alpha$ , which tends to be quite problematic because if set too low, it can make the convergence rate of the solution too small, but in the contrary, if it is set too high, the solution might diverge. Or other problems previously mentioned such as the saddle points or the loss fluctuations.

This update is strongly related to physics, so let's explain it somehow. First, we have to consider the loss surface as a valley and the initial solution is a football ball located somewhere in

that valley. The initial velocity can be considered equal to zero because the random initialization of the parameters (usually) has a mean in zero. Then, the potential energy of the ball is probably not zero (can be located somewhere within the valley);  $P_{Energy} = mgh$ . Hence, the force felt by the ball is the (negative) gradient of the loss function (valley);  $F = -\nabla U$ . Therefore, as soon as the ball starts rolling down due to the gravity, it will gain kinetic energy in exchange for the potential energy at the rate of the gradient;  $P_{Energy} + K_{Energy} = k$ . Finally, we know that the ball will stop at the bottom of the valley, but maybe not at the lowest part. Besides, before losing all its kinetic energy the ball will oscillate around the bottom (local minima) until it loses all its kinetic energy.

Additionally, the value of the hyperparameter *momentum* can be seen as the coefficient of friction in a physical interpretation since its value (typically around 0.9) reduces the *kinetic energy*

Analytically, it is pushing the solution a bit further:



**Figure 3.11:** Momentum update

**Source:** CS231n Convolutional Neural Networks for Visual Recognition

Mathematically, a *SGD* with momentum can be expressed as follows:

$$\begin{aligned} v_t &= \gamma v_{t-1} + \eta \nabla_{\theta} J(\theta) \\ \theta &= \theta - v_t \end{aligned} \quad (3.29)$$

I prefer to simplify things even more and see the *Momentum* update as: *an update with inertia*, expressed by:

$$\begin{aligned} \text{velocity} &:= \text{momentum} \cdot \text{velocity} - \text{learning\_rate} \cdot \text{gradient} \\ \text{param} &:= \text{param} + \text{velocity} \end{aligned} \quad (3.30)$$

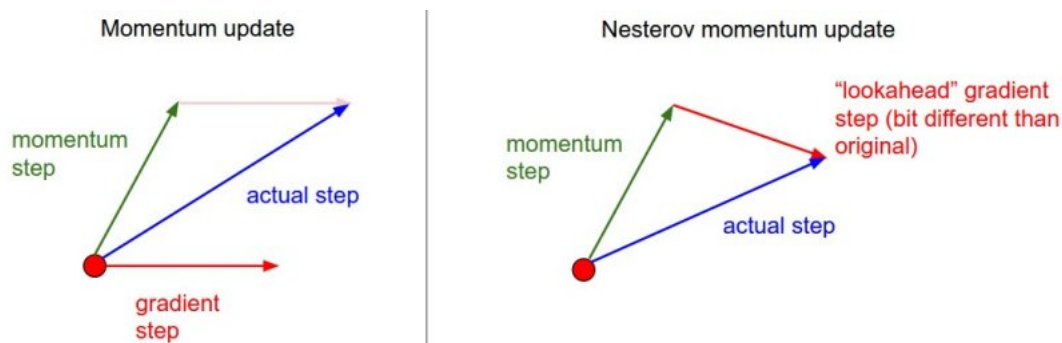
### 3.5.2.3. Nesterov's Accelerated Gradient (NAG)

*NAG*, also known as *Nesterov Momentum* has been gaining popularity lately, especially in the field of deep learning. With regard to the theory, the *Nesterov Momentum* has stronger theoretical converge guarantees for convex functions than standard momentum. Besides, in practice consistently works slightly better.

Following the previous analogy, a ball with momentum reaches faster the bottom of the valley, but it doesn't know where is going. To do so, we need a smarter ball, a ball with a notion of where it is going. This smarter ball might not follow the optimal path towards the bottom of the valley, but we're confident that its step down to the hill will be closer to the optimal one than the step that blindly ball with momentum would have taken.

So that we can translate this analogy to a mathematical expression, let's say that *NAG* performs the standard momentum update but looking ahead into the future. As we cannot look into the

future, we have to guess the future position through an approximation which makes use of the current momentum.



**Figure 3.12:** Momentum update Vs. Nesterov update

**Source:** CS231n *Convolutional Neural Networks for Visual Recognition*

Mathematically, the *Nesterov* update can be expressed as follows:

$$\begin{aligned} v_t &= \gamma v_{t-1} + \eta \nabla_{\theta} J(\theta - \gamma v_{t-1}) \\ \theta &= \theta - v_t \end{aligned} \quad (3.31)$$

Following the philosophy of simplicity, we can see the *Nesterov momentum* update as: *an update with momentum that looks ahead into the future*, expressed by: <sup>14</sup>

$$\begin{aligned} param\_ahead &:= param + momentum \cdot velocity \\ velocity &:= momentum \cdot velocity - learning\_rate \cdot gradient\_ahead \\ param &:= param + velocity \end{aligned} \quad (3.32)$$

Although the previous form is easier to understand, from a programming perspective reusing chunks of code is very valuable (here, the momentum update function). Accordingly, we can express this update in such a way that it looks as the *vanilla SGD*:

$$\begin{aligned} v\_prev &:= velocity \\ velocity &:= momentum \cdot velocity - learning\_rate \cdot gradient \\ param &:= param - momentum \cdot v\_prev + (1 + momentum) \cdot velocity \end{aligned} \quad (3.33)$$

#### 3.5.2.4. AdaGrad

*AdaGrad* is an adaptive gradient algorithm, what means that it adapts the learning rate to the parameters. There is one learning rate per parameter and the algorithm adapts them in order to deal with the sparsity of the parameters, giving larger values for sparse parameters and shorter ones for less sparse parameters. A consequence of this approach is the significant improvement in the convergence rate over other stochastic gradient-based approaches where the data is sparse. Because of that, it can be a good option in problems such as natural language processing, image recognition or basically, for training large-scale neural networks.

One important downside to *AdaGrad* is that its monotonic learning rate can be too aggressive so it stops learning too early. This is produced because it accumulates the squared gradients in the

<sup>14</sup>*param\_ahead* is equivalent to say *x\_ahead* (future “ball position”), therefore, *gradient\_ahead* is the same as saying *dx\_ahead*. In other words, *gradient\_ahead* is the gradient at *x\_ahead* instead of at *x*

denominator. Then, after each iteration the learning rate shrinks until it becomes infinitesimally small and the learning is stopped.

Mathematically, the *AdaGrad* update can be expressed as follows:

$$\begin{aligned} g_{t,i} &= \nabla_{\theta} J(\theta_i) \\ \theta_{t+1,i} &= \theta_{t,i} - \eta \cdot \frac{g_{t,i}}{\sqrt{G_{t,ii}} + \epsilon} \end{aligned} \quad (3.34)$$

To grasp the concept and thus get a better intuition of what *AdaGrad* does, it could be seen as an optimizer that adapts its learning rate during the process, expressed by:

$$\begin{aligned} \text{cache} &:= \text{cache} + \text{gradient}^2 \\ \text{param} &:= \text{param} - \text{learning\_rate} \cdot \frac{\text{gradient}}{\sqrt{\text{cache} + \text{epsilon}}} \end{aligned} \quad (3.35)$$

### 3.5.2.5. RMSProp

RMSProp (Root Mean Square Propagation) is a method that adapts the per-parameter learning rate, as *AdaGrad* does. But here, the idea is to divide the learning rate for a weight by a running average of the magnitudes of recent gradients for that weight[22].

Basically, *RMSProp* modifies *Adagrad* in order to reduce its aggressiveness by monotonically decreasing its learning rate.

Mathematically, it can be expressed as follows:

$$\begin{aligned} E[g^2]_t &= 0.9E[g^2]_{t-1} + 0.1g_t^2 \\ \theta_{t+1} &= \theta_t - \eta \cdot \frac{g_t}{\sqrt{E[g^2]_t + \epsilon}} \end{aligned} \quad (3.36)$$

Again, to get a better intuition of what *RMSProp* does, it could be seen as a *leaky AdaGrad*, expressed by:

$$\begin{aligned} \text{cache} &:= \text{decay\_rate} \cdot \text{cache} + (1 - \text{decay\_rate}) \cdot \text{gradient}^2 \\ \text{param} &:= \text{param} - \text{learning\_rate} \cdot \frac{\text{gradient}}{\sqrt{\text{cache} + \text{epsilon}}} \end{aligned} \quad (3.37)$$

### 3.5.2.6. AdaDelta

Similar to *RMSProp*, *AdaDelta* is a method that adapts the per-parameter learning rate and seeks to reduce the aggressiveness of *AdaGrad* by monotonically decreasing learning rate.

The main difference with regards to *RMSProp* is that instead of accumulating all past squared gradients, *AdaDelta* restricts the window of accumulated past gradients to some fixed size  $w$ [17].

Mathematically, it can be expressed as follows:

$$\begin{aligned} E[\Delta\theta^2]_t &= \gamma E[\Delta\theta^2]_{t-1} + (1 - \gamma)\Delta\theta_t^2 \\ \text{RMS}[\Delta\theta]_t &= \sqrt{E[\Delta\theta^2]_t + \epsilon} \\ \Delta\theta_t &= -\frac{\text{RMS}[\Delta\theta]_{t-1}}{\text{RMS}[g]_t} g_t \\ \theta_{t+1} &= \theta_t + \Delta\theta_t \end{aligned} \quad (3.38)$$

To grasp the concept of *AdaDelta*, it could be seen as *an extension of Adagrad that seeks to reduce its aggressiveness*, expressed by:

$$\begin{aligned}
 \text{cache} &:= \text{decay\_rate} \cdot \text{cache} + (1 - \text{decay\_rate}) \cdot \text{gradient}^2 \\
 \text{update} &:= \text{gradient} \cdot \sqrt{\frac{\text{delta\_cache}}{\text{cache} + \text{epsilon}}} \\
 \text{param} &:= \text{param} - \text{learning\_rate} \cdot \text{update} \\
 \text{delta\_cache} &:= \text{decay\_rate} \cdot \text{delta\_cache} + (1 - \text{decay\_rate}) \cdot \text{update}^2
 \end{aligned} \tag{3.39}$$

### 3.5.2.7. Adam

*Adam* (for Adaptive Moment Estimation) is a method that adapts the learning rates for each parameter. Basically, it is an update to *RMSProp* optimizer. Hence, it is practically the same as *RMSProp* except that it uses a *smooth* version of the gradient. A possible explanation for this smoothed version of the gradient may lie in trying to remove part of the maybe noisy original gradient.

Mathematically, it can be expressed as follows:

$$\begin{aligned}
 m_t &= \beta_1 \cdot m_{t-1} + (1 - \beta_1) \cdot g_t \\
 v_t &= \beta_2 \cdot v_{t-1} + (1 - \beta_2) \cdot g_t^2 \\
 \theta_{t+1} &= \theta_t - \eta \cdot \frac{m_t}{\sqrt{v_t} + \epsilon}
 \end{aligned} \tag{3.40}$$

Usually,  $m_t$  and  $v_t$  are initialized as a vectors of zeros. The problem of this initialization is that  $m_t$  and  $v_t$  will be biased towards zero. In order to counteract these biases, the authors presented a corrected version[24]:

$$\begin{aligned}
 \hat{m}_t &= \frac{m_t}{1 - \beta_1^t} \\
 \hat{v}_t &= \frac{v_t}{1 - \beta_2^t}
 \end{aligned} \tag{3.41}$$

In simple words, *Adam* is a method similar to *AdaGrad* and *RMSProp* but with an exponentially decaying average of past gradients. Another way to see it is like *a RMSProp with momentum*, expressed by:

$$\begin{aligned}
 m &:= \text{beta}_1 \cdot m + (1 - \text{beta}_1) \cdot \text{gradient} \\
 v &:= \text{beta}_2 \cdot v + (1 - \text{beta}_2) \cdot \text{gradient}^2 \\
 \text{param} &:= \text{param} - \text{learning\_rate} \cdot \frac{m}{\sqrt{v} + \text{epsilon}}
 \end{aligned} \tag{3.42}$$

### 3.5.2.8. Adamax

*Adamax* is practically the same method as *Adam*, but based on the infinity norm. The only difference is that instead of always using the smoothed version of the gradient, it takes the maximum between this version and the original gradient ( $\max(v_t, g_t)$ ). For practical purposes, the single most important feature w.r.t *Adam* is that the infinity norm makes the algorithm surprisingly stable[24].

Hence, the mathematical definition will be expressed as follows:

$$\begin{aligned} m_t &= \beta_1 \cdot m_{t-1} + (1 - \beta_1) \cdot g_t \\ v_t &= \beta_2 \cdot v_{t-1} + (1 - \beta_2) \cdot g_t^2 \\ \theta_{t+1} &= \theta_t - \eta \cdot \frac{m_t}{\sqrt{\max(v_t, |g_t|) + \epsilon}} \end{aligned} \quad (3.43)$$

The intuition remains as the *Adam*'s one. At best we could add a comment as *Adamax is Adam based on the infinity norm..* Similarly, it would be expressed by:

$$\begin{aligned} m &:= \text{beta}_1 \cdot m + (1 - \text{beta}_1) \cdot \text{gradient} \\ v &:= \text{beta}_2 \cdot v + (1 - \text{beta}_2) \cdot \text{gradient}^2 \\ \text{param} &:= \text{param} - \text{learning\_rate} \cdot \frac{m}{\sqrt{\max(v, |\text{gradient}|) + \text{epsilon}}} \end{aligned} \quad (3.44)$$

### 3.5.3. Non-convexity of MF problems

A linear model is defined as *a model which is linear in the parameters which have to be estimated but not necessarily in the independent variables.*

Therefore, this is a linear model:

$$h_\theta(x) = \theta_0 + \theta_1 x_1 + \theta_2 x_2 + \theta_3 x_3 + \dots + \theta_n x_n \quad (3.45)$$

...and this too:

$$h_\theta(x) = \theta_0 + \theta_1 x_1 + \theta_2 x_2^2 \quad (3.46)$$

...but not this:

$$h_\theta(x) = \theta_0 + \frac{x_1}{\theta_1} \quad (3.47)$$

It is straightforward to see the partial derivatives of the first model do not depend on unknowns ( $x$  is a known value; *Input data*). Hence it's linear. The second model is also linear, technically *curvilinear*. Although  $x^2$  is not a linear function, the model is linear in the parameters so we don't care about the  $x$  power. (Actually, this can be a trick to train linear models with nonlinear data). Finally, the third model is not linear so the parameter  $\theta_1$  is powered to  $-1$ , what means that the derivative of the second term depends on this parameter.

$$\frac{\partial h_\theta(x)}{\partial \theta_1} = -\frac{x_1}{\theta_1^2} \quad (3.48)$$

It's important to highlight that linearity is also a point of view. Depending on the field and the problem, the linearity of a model can be defined in different ways or under certain constraints. To sum up, we can say that a model is nonlinear if it is not linear in at least one parameter. Or even easier: *a nonlinear model is a model that is not linear.*

The linearity of a model has huge implications for the optimization of its parameters. In a nonlinear space, convexity is the exception. Hence, the convexity of a problem is something that has to be proven, not assumed.

To prove the non-convexity of a simple matrix factorization model, let's use *Funk's SVD* model:

$$\min_{p^*, q^*} \sum_{(u, i \in k)} (r_{ui} - q_i^T p_u)^2 \quad (3.49)$$



As the rating matrix  $R$  is defined as  $R^{m \times n}$ , let's considering the simplest case where  $m = 1$  and  $n = 1$ ;  $R^{1 \times 1}$ :

$$\min_{p^*, q^*} \sum_{(u, i \in k)} (r_{ui} - q_i^T p_u)^2 \Rightarrow \min_{p^*, q^*} (r - q^T p)^2 = \min_{p^*, q^*} (r^2 - 2rqp + q^2 p^2) \quad (3.50)$$

One of the properties of the *Hessian matrix* is that it describes the local curvature of a function of many variables. Thus, we can use it to study the convexity of our model which it will be redefined as:

$$h_r(p, q) = r^2 - 2rqp + q^2 p^2 \quad (3.51)$$

...where  $h$  stands for hypothesis.

Then, the gradient and the Hessian matrix is:

$$\nabla h_r(p, q) = \begin{bmatrix} 2pq^2 - 2rq \\ 2p^2q - 2rp \end{bmatrix} \quad (3.52)$$

$$\nabla^2 h_r(p, q) = \begin{bmatrix} 2q^2 & 4pq - 2r \\ 4pq - 2r & 2p^2 \end{bmatrix} \quad (3.53)$$

Consequently, if the Hessian is positive semidefinite in the entire domain, then the function will be convex. Since it is not, it's non-convex.

$$\nabla^2 h_1(2, 1) = \begin{bmatrix} 2 & 6 \\ 6 & 8 \end{bmatrix} \quad (3.54)$$

Moreover, the sign of its eigenvalues is different, so the concavity of the surface is inconsistent at  $(p, q) = (2, 1)$ ;  $\lambda_1 > 0$  and  $\lambda_2 < 0$  [26]:

$$\begin{aligned} \lambda_1 &= 11.7082 \\ \lambda_2 &= -1.7082 \end{aligned} \quad (3.55)$$

To sum up, in the space of nonlinear problems, convexity is the exception, not the rule. Therefore, convexity is something to be proven, not assumed.

Now that we know that MF problems are usually non-convex, it is interesting to understand why gradient-descent methods work here. Gradient descent is a generic method for continuous optimization so in non-convex problems the solution will eventually converge to a stationary point (either a local minimum or a saddle point). The reason why gradient-descent methods tend to work so well with this kind of problems in many situations is related to the dimensionality of the problem and the properties of quasiconvex functions. Additionally, small tricks and optimization on the gradients allow the algorithms to escape from certain stationary and saddle points.

Another approach we can take to optimizing the loss function is using Alternating Least Squares (ALS), which is a two-step iterative optimization process. With ALS we can turn a non-convex optimization problem into a quadratic problem that can be solved optimally. The algorithm works by fixing one of the unknowns and solving for the other, alternating these in each iteration.

In general, SGD approaches are easier and faster than ALS, but ALS is usually preferred for systems centered on implicit data or when the system can use parallelization.



---

# CHAPTER 4

## Software developed

---

### 4.1 Orange3

---

Orange is an open-source, cross-platform, component-based data mining and machine learning software suite which features friendly yet powerful and flexible visual programming front-end for exploratory data analysis, visualization, model construction, evaluation, and forecast[30].

Orange allows you to do interactive data analysis workflows with a large toolbox:

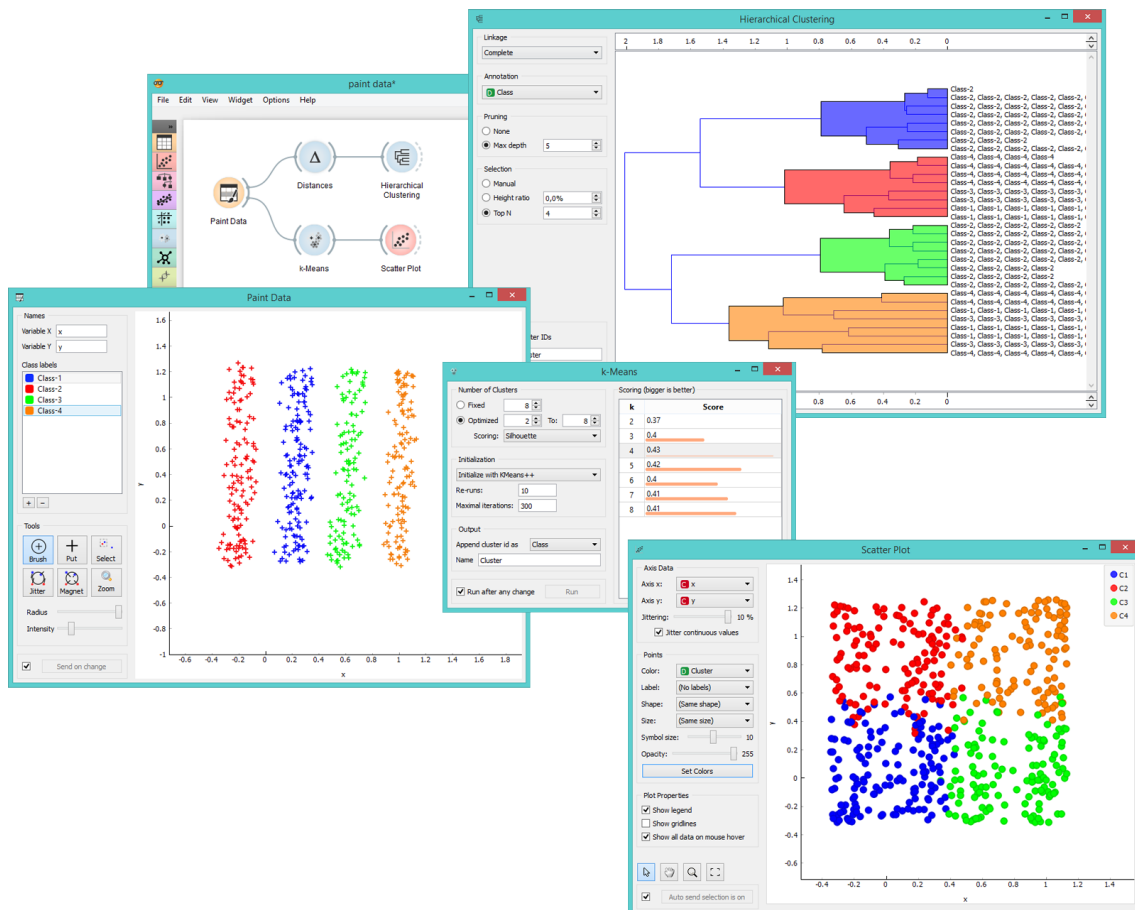
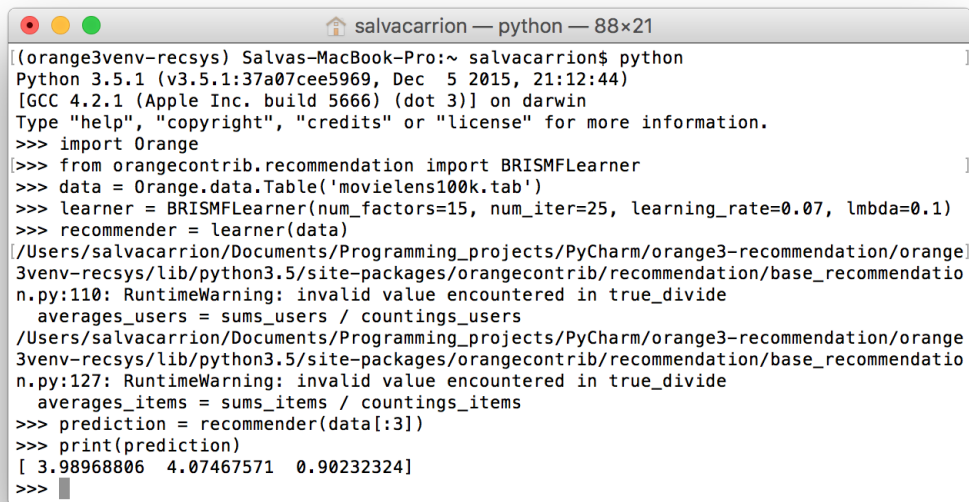


Figure 4.1: Orange3 Data Mining

Additionally, it also can be used from an interactive notebook like *iPython* or simply from the terminal:



```

salvacarrion — python — 88x21
(orange3venv-recsys) Salvas-MacBook-Pro:~ salvacarrion$ python
Python 3.5.1 (v3.5.1:37a07cee5969, Dec 5 2015, 21:12:44)
[GCC 4.2.1 (Apple Inc. build 5666) (dot 3)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>> import Orange
[>>> from orangecontrib.recommendation import BRISMFLearner
>>> data = Orange.data.Table('movielens100k.tab')
>>> learner = BRISMFLearner(num_factors=15, num_iter=25, learning_rate=0.07, lmbda=0.1)
>>> recommender = learner(data)
/Users/salvacarrion/Documents/Programming_projects/PyCharm/orange3-recommendation/orange3venv-recsys/lib/python3.5/site-packages/orangecontrib/recommendation/base_recommendation.py:110: RuntimeWarning: invalid value encountered in true_divide
  averages_users = sums_users / countings_users
/Users/salvacarrion/Documents/Programming_projects/PyCharm/orange3-recommendation/orange3venv-recsys/lib/python3.5/site-packages/orangecontrib/recommendation/base_recommendation.py:127: RuntimeWarning: invalid value encountered in true_divide
  averages_items = sums_items / countings_items
>>> prediction = recommender(data[:3])
>>> print(prediction)
[ 3.98968806  4.07467571  0.90232324]
>>>

```

**Figure 4.2:** Terminal execution

...and as a library, which can be installed from *pip install orange3*. (If you want to install this add-on directly, you can use *pip install orange3-recommendation*):

```

1  import Orange
2  from orangecontrib.recommendation import BRISMFLearner
3
4  # Load data and train the model
5  data = Orange.data.Table('movielens100k.tab')
6  learner = BRISMFLearner(num_factors=15, num_iter=25, learning_rate=0.07, lmbda=0.1)
7  recommender = learner(data)
8
9  # Make predictions
10 prediction = recommender(data[:3])
11 print(prediction)
12 >>>
13 [ 3.79505151  3.75096513  1.293013 ]

```

**Figure 4.3:** Python implementation

## 4.2 Orange3-Recommendation

*Orange3-Recommendation* is an add-on for *Orange3* to include support for recommender systems. For this purpose, we've built an efficient and unified interface to tackle the problem of collaborative filtering (CF). The scripting library is the core of the add-on, which includes a number of published factorization algorithms as well as methods to optimize them and analyze their outcomes.

### 4.2.1. Architecture

*Orange3-Recommendation* uses a model-view-controller (MVC)<sup>1</sup> design pattern as all official *Orange3* add-ons.

Basically, MVC is a software design pattern for implementing user interfaces, in which the software application is divided into several interconnected parts, so to separate the internal representation of information from the user's final presentation.

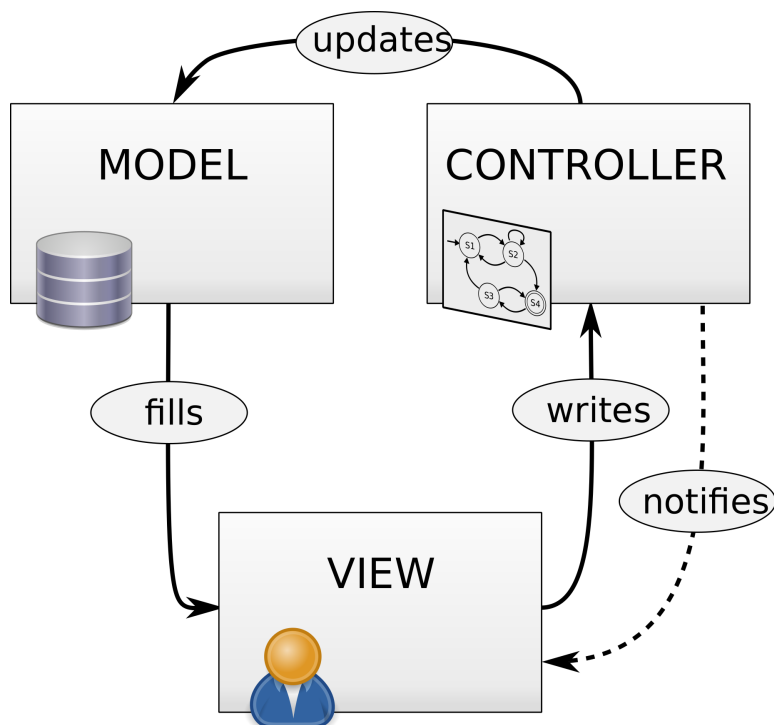
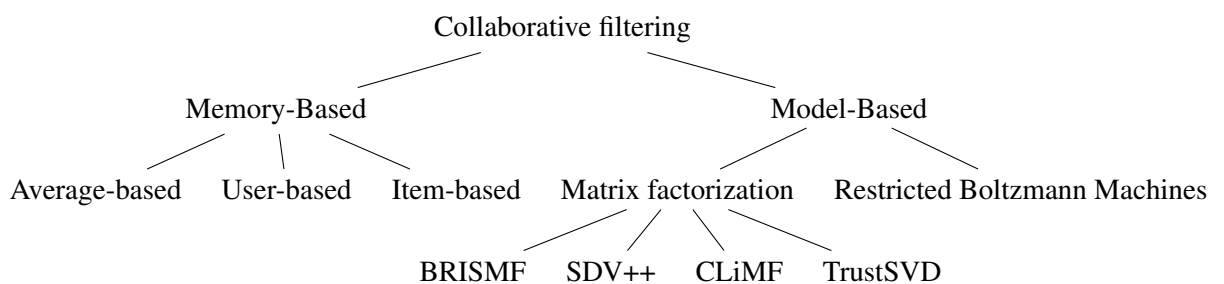


Figure 4.4: Model-View-Controller (MVC)

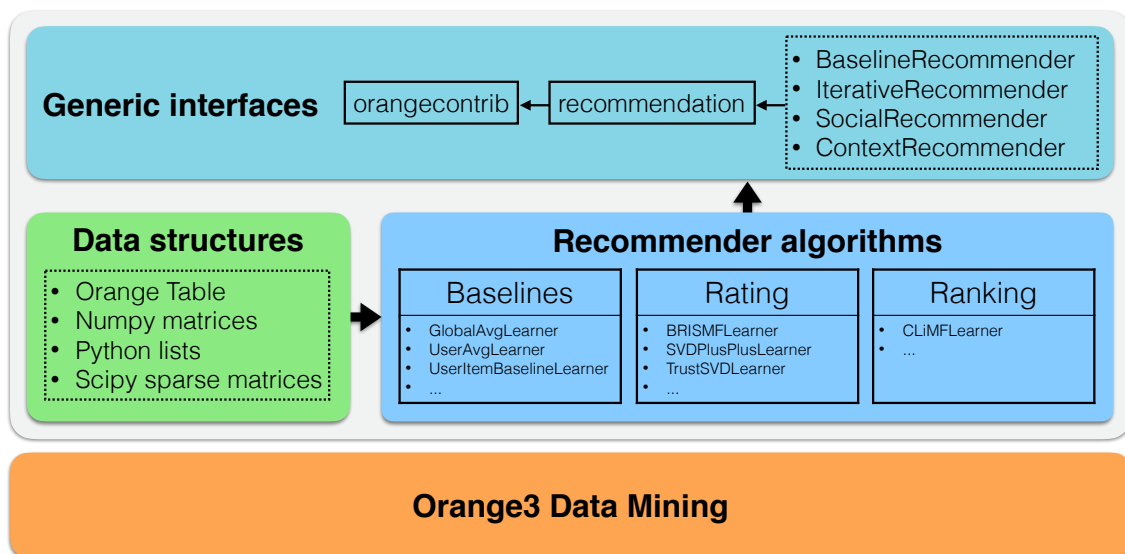
At a conceptual level, this add-on follows a representation in which every model belongs to a parent node (group):



<sup>1</sup>Image source: Wikimedia, MVC diagram

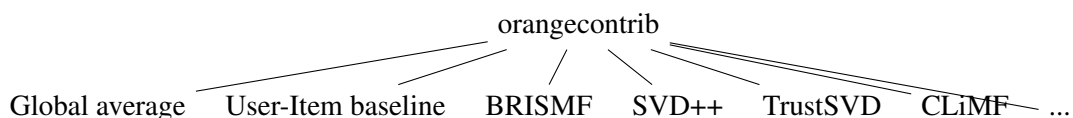
In practice, it's pretty complicated to strictly follow this previous architecture due to problems such as Orange architecture constraints, add-on homogenization problems, code re-usability,... As a result of these problems, we needed to use a different architecture.

The architecture presented here is a simplification of the real one, which will be introduced later. *Orange3-Recommendation* is built on top of *Orange3* but preserving a relative independence with regard to the middle-high level model abstractions and the data structures. For example, *recommender algorithms* are independent of *Orange*, but to do so, they need more abstraction (provided by the generic interfaces). Similarly, the models can work with many types of different data structures such as Orange Tables, Numpy matrices, python lists, scipy sparse matrices,... but as Orange mainly works with its own data structures (Orange Tables), we need a new layer of abstraction.

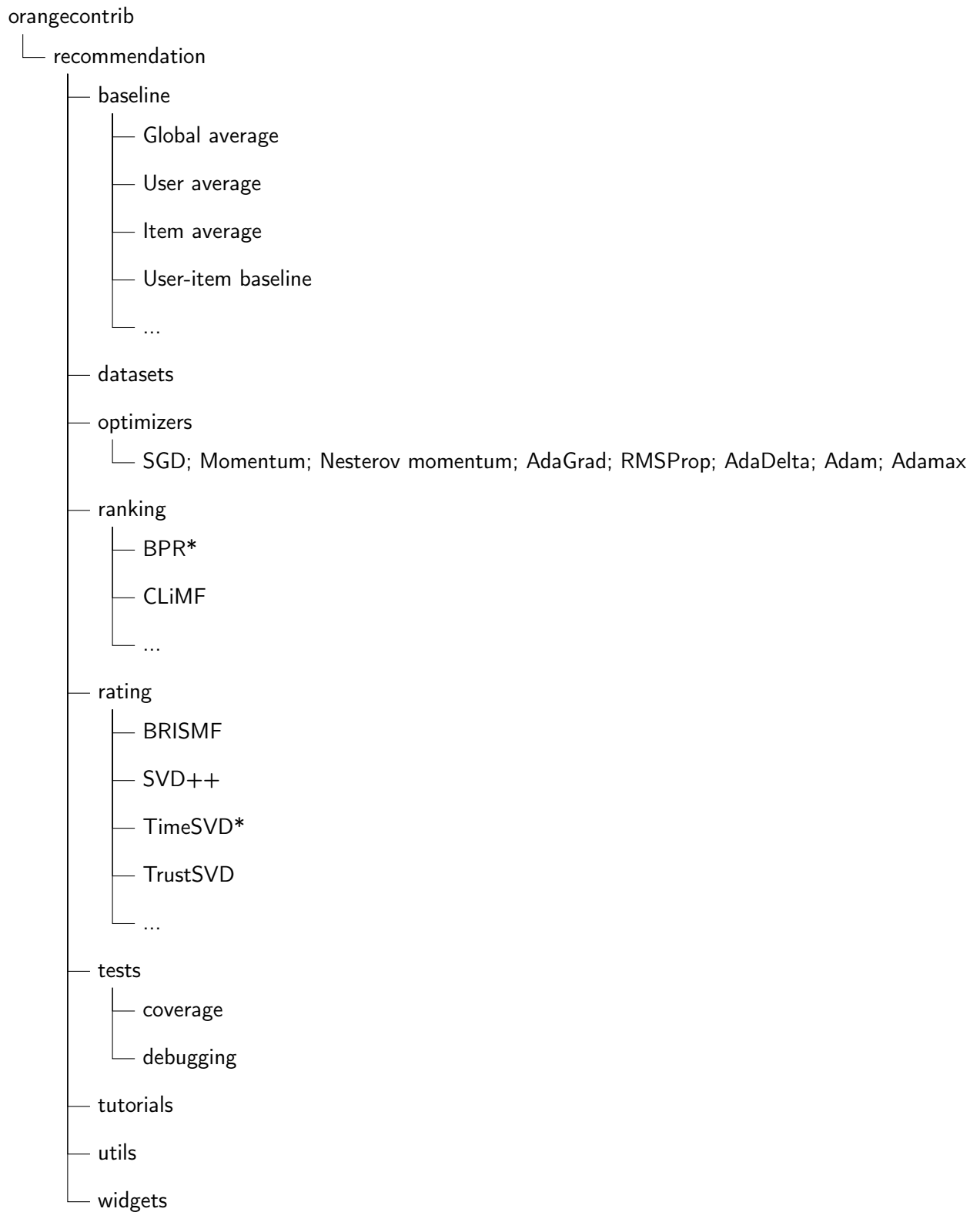


**Figure 4.5:** The class structure of *Orange3-Recommendation*

All *Orange3* add-ons must be included inside a folder named *orangecontrib*, which tells *Orange* where is located the core of the add-on so it can be added as a valid extension. Besides this folder, there are many others that need a special treatment. It's not hard to see, that one of the most important problems that we encounter with this kind of structure is that it might produce a couple headaches in novel users and developers. To solve it, all users (and developers) can have access to all the models directly from the root, with no need to import or write the whole path. In this way, they can get focus on the things that matter such as solving the problem, instead of wasting their time in writing long paths or in trying to remember the name of a certain parent node.



Currently, the folder structure of the package is defined by representative folders in such a way that it gives a rough idea of how it is organized internally:



#### 4.2.2. Technologies

Following the *Orange3* requirements and back-compatibility, this will be the libraries used:

- *Python* == 3.4
- *Scikit-Learn*  $\geq 0.16$
- *Scipy*  $\geq 0.11.0$
- *Numpy*  $\geq 1.9.0$
- *Theano*  $\geq 0.8$
- *Keras*  $\geq 0.3.2$  or *Lasagne*  $\geq 0.1$
- *Cython*  $\geq 0.23.4$
- *Bottlechest*  $\geq 0.7.1$
- *Nose* == 1.2
- *Unittest*  $\geq 2.1$

#### 4.2.3. Workflow

The development workflow was based on the SCRUM methodology, where every Friday there was a meeting to discuss new aspects of the software, designs, implementations and problems.

All the code is available on Github<sup>2</sup> and as many other open source organizations, *Orange* has some development guidelines to ensure the quality and correctness of our code.

For example, we use *TravisCI* for the continuous integration, which builds and tests projects hosted at Github. By doing so, we can partially ensure that new commits haven't broken anything. As a rule of thumb, every official add-on in *Orange* must ensure a coverage of at least a 90%, although the more, the better. A fantastic tool to check this coverage automatically is *Codecov*, so with every new commit in our repo, it checks automatically the new changes. Regarding the quality, this was optional, but I used *Codacy* with a custom arrange of settings for my purposes.

Furthermore, we had to follow some extra guidelines such as a maximum of 79 chars per line, PEP8 rules, Napoleon-compatible docstrings (Google Style Python Docstrings), Google Style Guide, OS X Human Interface Guidelines, *bug reporting guidelines*<sup>3</sup>, Git Commit-Guidelines, Orange3 Contributing guidelines, etc.

Finally, after the completion of the add-on, was mandatory to write a few entries in the official blog and a tutorial.

#### 4.2.4. Documentation

A well documented open source project is a must. Therefore, all the methods in our code are well documented using the *Google Style Python Docstrings Guide*.

The second requirement was to pass the add-on into *Sphinx* so it could build the local documentation. Additionally, every repo must be connected to *ReadTheDocs*, which is a hosting for documentation (free for open source projects). This tool builds the online documentation of the project with every new commit on Github. (See more: <http://orange3-recommendation.readthedocs.io/>)

<sup>2</sup><https://Github.com/biolab/orange3-recommendation>

<sup>3</sup><http://www.chiark.greenend.org.uk/~sgtatham/bugs.html>



### 4.2.5. Blog

Similar to the online documentation, every add-on or important feature in *Orange3* must have an entry in their official blog explaining the main aspects of the add-on in a really simple way. This blog is mostly oriented towards beginners or non-programmer scientists.

My entry is pretty visual, with plenty step-by-step pictures and short descriptions. In it, I explain the graphical part and as well as the scripting one. (See more: <http://blog.biolab.si/2016/08/19/making-recommendations/>)

### 4.2.6. Tutorials

*Orange3* tutorials are more like a splash screen with a list of projects, that consists of simple pipelines to do a specific task and lots of descriptions for each part or widget use it.

For this add-on I wrote two tutorials, the first one is a simple pipeline for baselines recommenders and the second one is a slightly more complex flow that shows how to use the rest of the widgets.

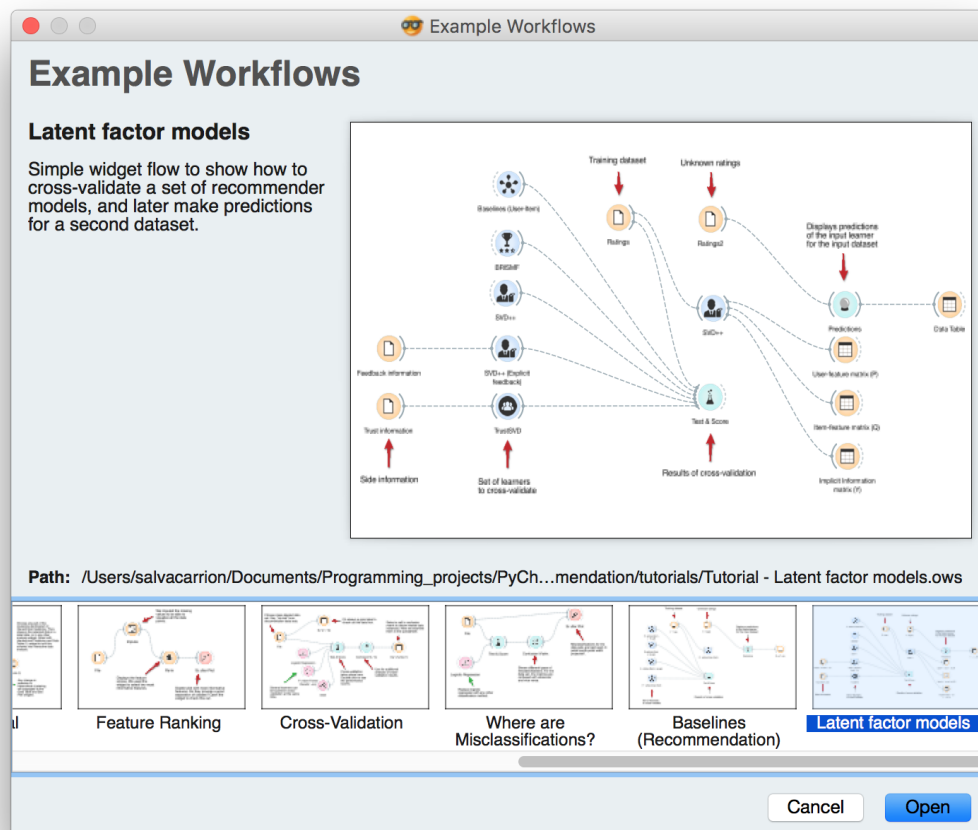


Figure 4.6: Orange3-Recommendation tutorial

### 4.2.7. Data Representation

Users usually rate a very small subset of available items. Since it wouldn't make sense to store all the missing information as a dense matrix, a sparse matrix data structure must be used.

To support this decision, several datasets have been studied under dense and sparse representations:

Dataset	Ratings	Dense-Matrix	Sparse-Matrix
MovieLens 100k	100,000	1.5MB	100KB
MovieLens 1M	1,000,209	23MB	1MB
Netflix Price	100,480,507	8GB	100MB

**Table 4.1:** Comparison of dense and sparse data representation.

After this trivial experiment, the need for a sparse representation is evident. Consequently, our first choice was to use *Scipy* sparse matrices so it's a package for numeric data represented as a sparse matrix with lots of functions.

There are many types of sparse matrices such as:

- BSR: Block Sparse Row matrix
- COO: A sparse matrix in COOrdinate format.
- CSC: Compressed Sparse Column matrix
- CSR: Compressed Sparse Row matrix
- DIA: Sparse matrix with DIAgonal storage
- DOK: Dictionary Of Keys based sparse matrix.
- LIL: Row-based linked list sparse matrix

Each of these matrices has specific properties, so we carefully study all of them, both theoretically and practically. In theory, Compressed Sparse Column matrix seemed to be a really good option but in practice was a complete disaster, and more or less the same with the other matrices. Finally, the reason behind this problem was because when a transpose was needed, *scipy* instead of simply change the view or the pointers of the matrix, it transforms the matrix to a different structure and then transforms the matrix back to its original structure. At the end, we were able to reduce the iteration time of a model from 2 days to 2.5 seconds. (See [4.2.7.1](#))

Fortunately, we didn't have to work too much with the format of the input data, so the vast majority of it was already defined by *Orange3*. With regard to the few problems that we had, such as the definition of timestamps, they were easy to solve by simply introducing minor modifications to the core of Orange.

Orange can read files in native tab-delimited format or load data from any major standard spreadsheet file types like CSV or Excel. Native format starts with a header row with feature (column) names. Second header row gives the attribute type, which can be continuous, discrete, string or time. The third header line contains meta information to identify dependent features (class), irrelevant features (ignore) or meta-features (meta). Here, we can see the first few lines from a typical dataset:

```
tid      user      movie     score
string  discrete  discrete  continuous
```

```
meta    row=1    col=1    class
1       Breza     HarrySally 2
2       Dana     Cvetje    5
3       Cene     Prometheus 5
4       Ksenija HarrySally 4
5       Albert   Matrix    4
...
```

The third row is mandatory in this kind of datasets, in order to know which attributes correspond to the users (row=1) and which ones to the items (col=1). For the case of big datasets, users and items must be specified as a continuous attributes due to efficiency issues. Again, here are the first few lines from a typical dataset:

```
user      movie      score      tid
continuous continuous continuous time
row=1     col=1     class     meta
196       242       3         881250949
186       302       3         891717742
22        377       1         878887116
244       51        2         880606923
166       346       1         886397596
298       474       4         884182806
...
```

#### 4.2.7.1. Results from the *TrustSVD* experiment

In other to tackle the efficiency problems related to the original implementation of *TrustSVD*, several implementations with different configurations of data structures were studied so that we could find a good trade-off between time and space requirements.

##### Approach 0

-----

- Data structure: Ratings\_LIL, Trust\_LIL, Feedback\_LIL
- Space complexity:  $1+1+1=3$  (100%)
- Time complexity: linear;  $O(d|R|+d|T|c^*)$
- Running time: 2 days

Note: Impractical

##### Approach 1

-----

- Data structure: Ratings\_CSR, Ratings\_CSC, Trust\_CSR, Trust\_CSC, Feedback\_CSR
- Space complexity:  $1+1+1+1+1=5$  (165%)
- Time complexity: linear;  $O(d|R|+d|T|c^*)$
- Running time: 168.738s

Note: CSR and CSC return a instance of itself when call `‘.getrow(i)’` or `‘.getcol(j)’`, then select tuple `‘[0]’` or `‘[1]’`, later cast `_numpy.int_` to python int for indices (array of `np.int32` to list of int)

##### Approach 2

-----

- Data structure: Ratings\_LIL, Trust\_LIL, Feedback\_LIL, cache\_users, cache\_items, cache\_trusters, cache\_trustees, cache\_feedback
- Space complexity:  $1+1+1+0.5+0.5+0.5+0.5+0.5=5.5$  (180%)
- Time complexity: linear;  $O(d|R|+d|T|c^*)$
- Running time: 30s

Note: The bottleneck in this function is the transpose (`ratings.T`, `trust.T`), precomputing this would make the algorithm faster (sacrificing memory)

##### Approach 3

-----

- Data structure: Ratings\_LIL, Ratings\_LIL\_T, Trust\_LIL, Trust\_LIL\_T, Feedback\_LIL, cache\_users, cache\_items, cache\_trusters, cache\_trustees, cache\_feedback
- Space complexity:  $1+1+1+1+1+0.5+0.5+0.5+0.5=7.5$  (250%)
- Time complexity: linear;  $O(d|R|+d|T|c^*)$
- Running time: 5s

Note: Too much memory consumption

## Approach 4

-----

- Data structure: Ratings\_LIL, Trust\_LIL, Feedback\_LIL, cache\_users, cache\_items, cache\_trusters, cache\_trustees, cache\_feedback
- Space complexity:  $1+1+1+0.5+0.5+0.5+0.5+0.5 = 5.5$  (180%)
- Time complexity: linear;  $O(d|R|+d|T|c^*)$
- Running time: 30s

Note: Approach 2 but transposing in the outer loops. There's is no gain due to the cache. (Everything is transposed once)

## Approach 5

-----

- Data structure: Ratings\_CSR, Ratings\_CSC, Trust\_CSR, Trust\_CSC, Feedback\_CSR, cache\_users, cache\_items, cache\_trusters, cache\_trustees, cache\_feedback
- Space complexity:  $1+1+1+1+1+0.5+0.5+0.5+0.5+0.5=7.5$  (250%)
- Time complexity: linear;  $O(d|R|+d|T|c^*)$
- Running time: Approach 2 < x < Approach 1

## Approach 6 (Final)

-----

- Data structure: Ratings\_LIL, Ratings\_LIL\_T, Trust\_LIL, Trust\_LIL\_T, Feedback\_LIL, cache\_users, cache\_trusters, cache\_feedback
- Space complexity:  $1+1+1+1+1+0.5+0.5+0.5=6.5$  (215%)
- Time complexity: linear;  $O(d|R|+d|T|c^*)$
- Running time: 2.5s (from 2 days to 2.5s)

Note: Cache rows of users, trusters and feedback, cache norms in a numpy array (python list are faster, 5s Vs. 12s; but numpy arrays allow vectorization (from 12s to 2.5s)) -> Method to vectorize what can be vectorized in that moment, and the rest is computed and stored in cache sequentially

#### 4.2.7.2. Issue with the *Prediction* widget

With *Orange 3.3.7*, the predictions widget should take more or less the same time as the required to compute the predictions. But instead of that, it takes too much time when it has to predict lots of samples ( $n > 100.000$ ).

A simple way to reproduce the behavior is to open the *Orange3* application, then take a learner which outputs a dumb prediction (e.g *return 0*) and finally try to predict lots of elements ( $n > 100,000$ ).

To show this issue, I did a simple experiment in which I used the widget Baselines (Global avg.) in add-on Orange3-Recommender with the MovieLens100K dataset as an input.

When I ran it from the terminal (scripting), I got these results:

- For 100,000 samples, it takes 0.001s.

- For 1,000,000 samples, it takes 0.010s.
- For 10,000,000 samples, it takes 0.150s.

But when I run the exact same model using the GUI (not in debugging mode), I got this:

- For 100,000 samples, it takes 23.04s.
- For 1,000,000 samples, it takes 240.03s (3m 50s)
- For 10,000,000 samples, it takes 2500s (40m)

This difference is due to the construction of PyQt table. The table should show the view of the returned array, not re-build it into a PyQt Table. This problem causes among other things, that Orange3 cannot be used to work big datasets. Something required for recommender systems.

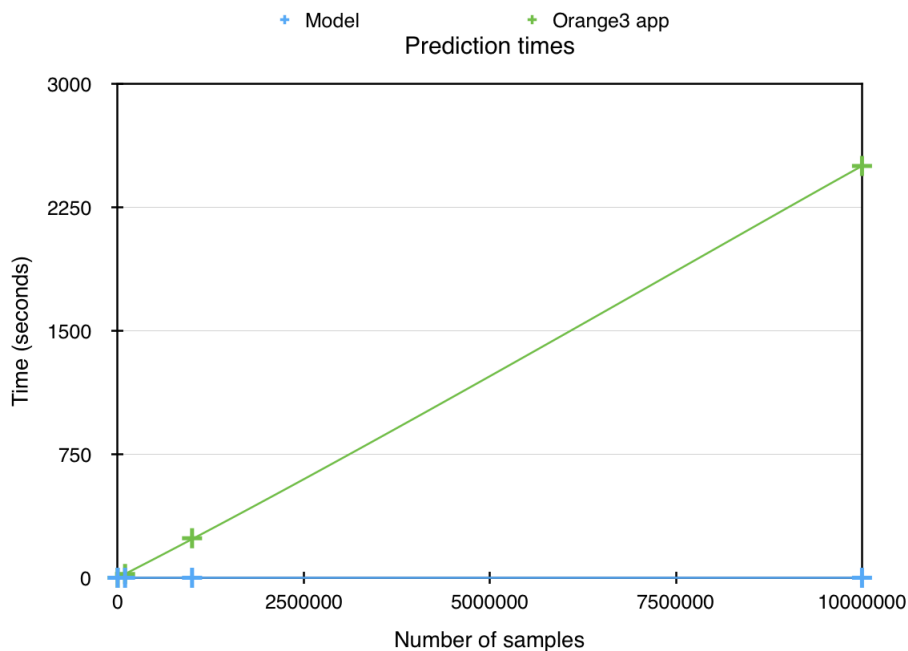


Figure 4.7: Prediction times

#### 4.2.8. Scripting

As *Orange3-Recommendation* follows a Model-View-Controller architecture, we can use it from the terminal or as a library.

For instance, let's train make a Python script to train a *BRISMF* model. First, we import Orange and the learner that we want to use:

```
1 import Orange
2 from orangecontrib.recommendation import BRISMFlearner
```

After that, we have to load a dataset:

```
1 data = Orange.data.Table('movielens100k.tab')
```

Then we set the learner parameters, and finally we train it passing the dataset as an argument (the returned object will be our model trained):

```

1 learner = BRISMFlearner(num_factors=15, num_iter=25, learning_rate=0.07,
2                           lambda=0.1)
3 recommender = learner(data)

```

Finally, we can make predictions (in this case, for the first three pairs in the dataset):

```

1 prediction = recommender(data[:3])
2 print(prediction)
3 >>> [ 3.79505151  3.75096513  1.293013 ]

```

Additionally, if we want to train more complex models that require additional information, we can write something as:

```

1 import Orange
2 from orangecontrib.recommendation import TrustSVDLearner
3
4 # Load data and train the model
5 data = Orange.data.Table('filmtrust/ratings.tab')
6 trust = Orange.data.Table('filmtrust/trust.tab')
7 learner = TrustSVDLearner(num_factors=15, num_iter=25, learning_rate=0.07,
8                           lambda=0.1, social_lambda=0.05, trust=trust)
9 recommender = learner(data)
10
11 # Make predictions
12 prediction = recommender(data[:3])
13 print(prediction)

```

So far, I have shown how to train and work with rating models. But what if we want to train a ranking model? This case is a little bit different. To explain it, let's make recommendations for a dataset in which only binary relevance is available. Specifically, the *CLiMF* model will suit our needs.

```

1 import Orange
2 import numpy as np
3 from orangecontrib.recommendation import CLiMFlearner
4
5 # Load data
6 data = Orange.data.Table('epinions_train.tab')
7
8 # Train recommender
9 learner = CLiMFlearner(num_factors=10, num_iter=10, learning_rate=0.0001,
10                        lambda=0.001)
11 recommender = learner(data)
12
13 # Make recommendations
14 recommender(X=5)
15 >>> [ 494, 803, 180, ..., 25520, 25507, 30815]

```

Similarly, if we want to evaluate our model with *MeanReciprocalRank* (for example), we can do it like this:

```

1 import Orange
2
3 # Load test
4 dataset_testdata = Orange.data.Table('epinions_test.tab')
5

```

```

6 # Sample users
7 num_users = len(recommender.U)
8 num_samples = min(num_users, 1000) # max. number to sample
9 users_sampled = np.random.choice(np.arange(num_users), num_samples)
10
11 # Compute Mean Reciprocal Rank (MRR)
12 mrr, _ = recommender.compute_mrr(data=testdata, users=users_sampled)
13 print('MRR: %.4f' % mrr)
14 >>> MRR: 0.3975

```

### 4.2.9. Widgets

Orange Widgets are components in Orange Canvas, a visual programming environment of Orange. They represent some self-contained functionalities and provide a graphical user interface (GUI). Widgets communicate with each other and pass objects through communication channels to interact with other widgets.<sup>4</sup>

For this add-on, each widget belongs to the *Recommendation* category and each one of them has an associated priority within that category. Currently, there are just four widgets available in the last release (v0.1.3):

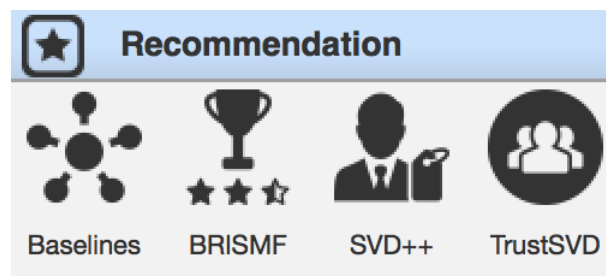


Figure 4.8: Widgets in the *Recommendation* category

### 4.2.10. Getting started

First of all, we need to start Orange3 app (GUI). Then, we only have to *drag&drop* widgets creating the workflow we need for our purposes. But enough theory, so let's get started!

#### 4.2.10.1. Training a model

To train a model we have to load the data as described above and connect it to the learner. (Don't forget to click apply)

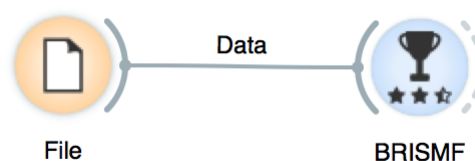
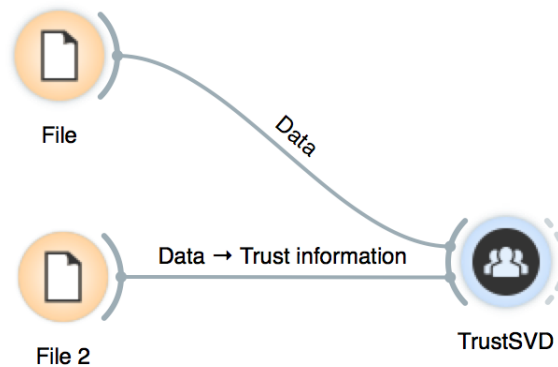


Figure 4.9: Feeding a model with ratings

<sup>4</sup>Orange Development 3 documentation - Widgets

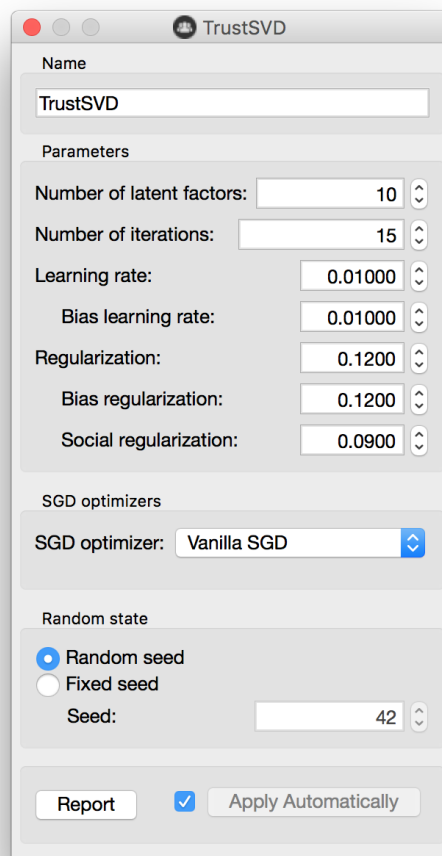


If the model uses side information, we only need to add an extra file.



**Figure 4.10:** Side/Trust information

In addition, we can set the parameters of our model by double-clicking it:

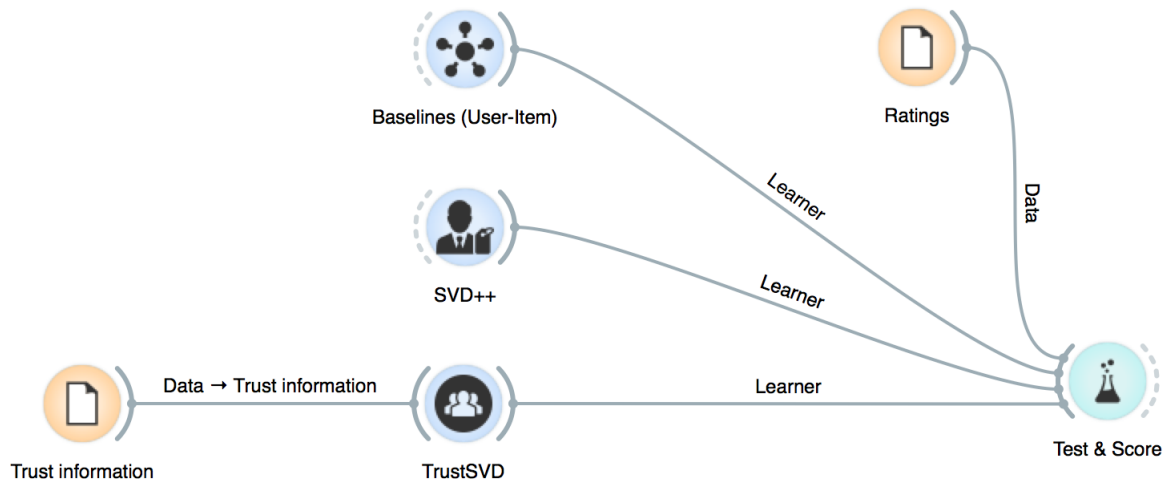


**Figure 4.11:** TrustSVD settings

By using a fixed seed, we make random numbers predictable. Therefore, this feature is useful if we want to compare results in a deterministic way.

### 4.2.10.2. Cross-Validation

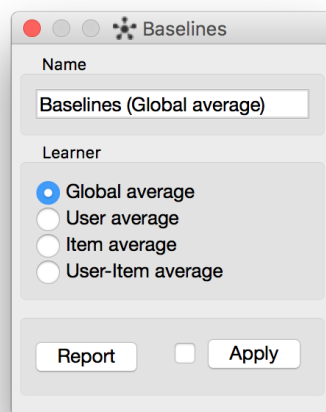
Cross-Validation in *Orange3* is as simple as it seems. The only thing to point out is that side information must be connected to the model we're evaluating.



**Figure 4.12:** Cross-Validation flow

Still, cross-validation is a robust way to see how our model will performs under different samples. I consider that it's a good idea to check how our model performs with respect to the baseline. This presents a negligible overload\* in our pipeline and makes our analysis more robust. (\*For 1,000,000 ratings, it can take 0.027s).

To do so, we can add a baseline learner to *Test&Score* and select the model we want to apply.



**Figure 4.13:** TrustSVD settings

### 4.2.10.3. Making recommendations

Then, we can make recommendations following the same pipeline as we would with the other widgets in *Orange3*.

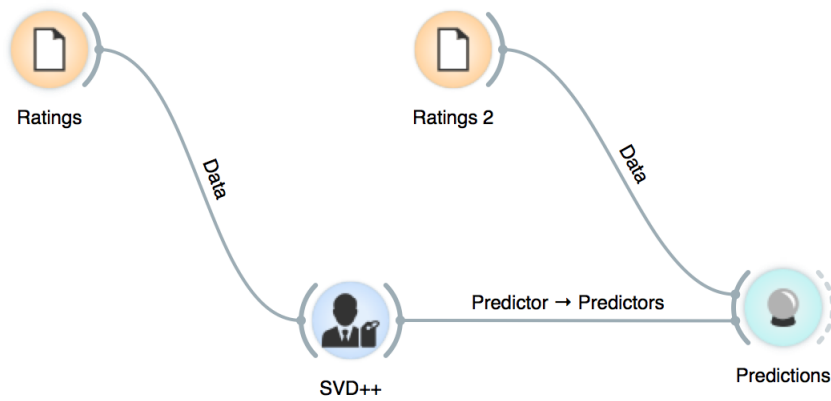


Figure 4.14: TrustSVD settings

#### 4.2.10.4. Analyzing low-rank matrices

Finally, we can output the resulting low-rank matrices after training our model so that they can be studied.

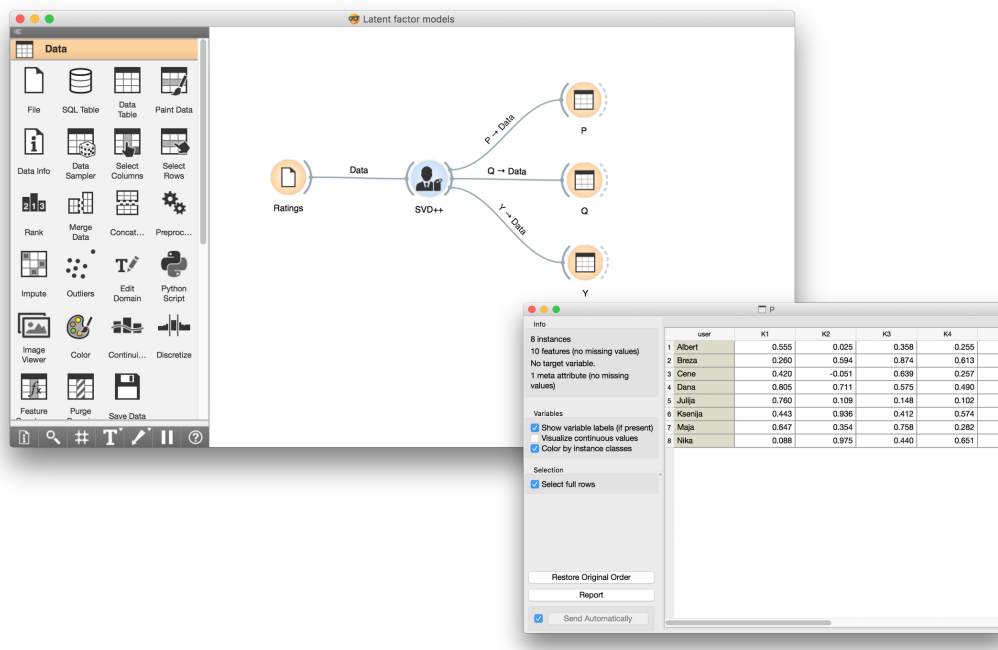


Figure 4.15: Low-rank matrices

Once we've output the low-rank matrices, we can play around the vectors in those matrices to discover hidden relations or understand the known ones. For instance, here we plot vector 1 and 2 from the item-feature matrix by simply connecting *Data Table* with selected instances to the widget *Scatter Plot*.

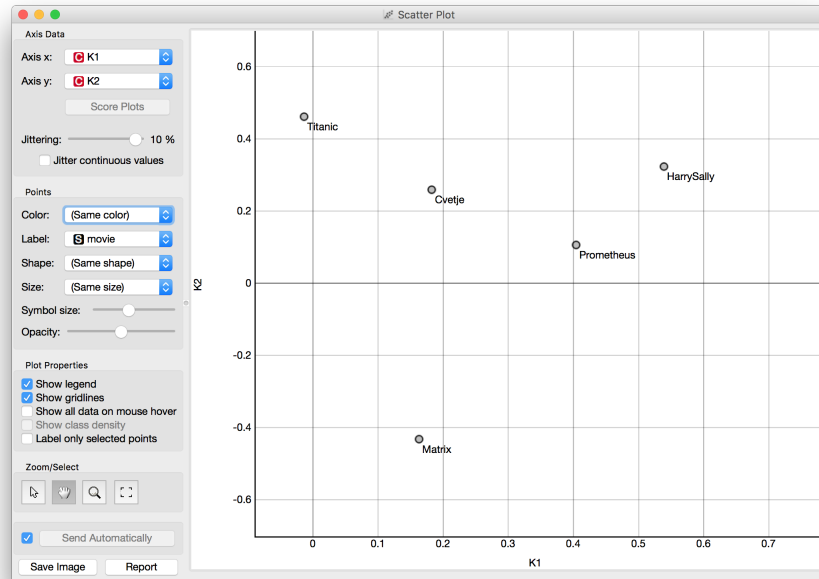


Figure 4.16: Visualizing items

Using similar approaches we can discover pretty interesting things like the similarity between movies or users, how movie genres relate to each other, changes in users' behavior, popularity of movies risen due to commercial campaigns, and many others.

Finally, a simple pipeline to do all of the above can be something like this:

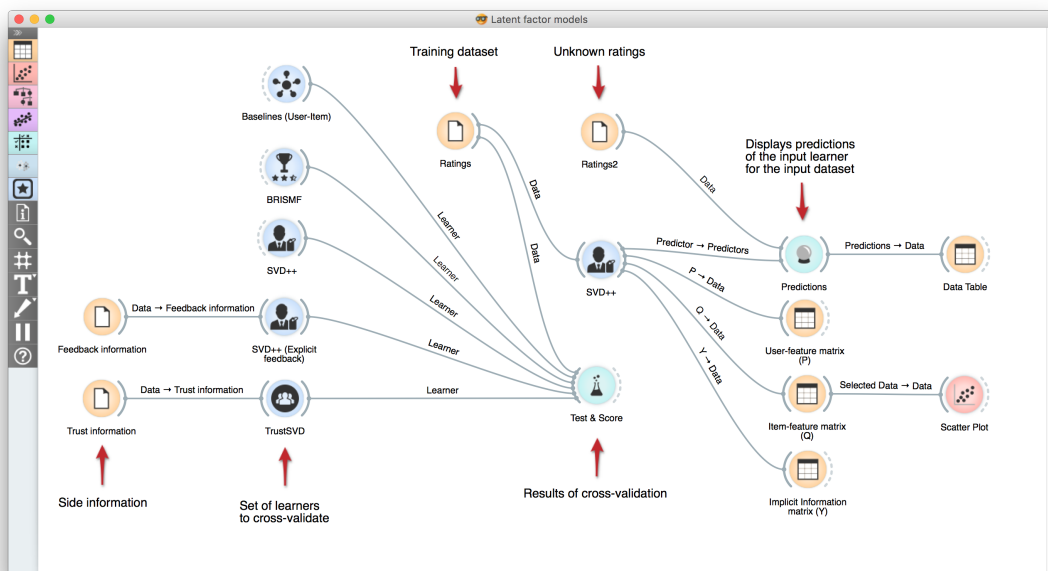


Figure 4.17: Recommendation workflow

On the left side we connected several models to the widget *Test&Score* in order to cross-validate them. Later, we trained a *SVD++* model, made some predictions, got the low-rank matrices learned by the model and plotted some vectors of the Item-feature matrix.

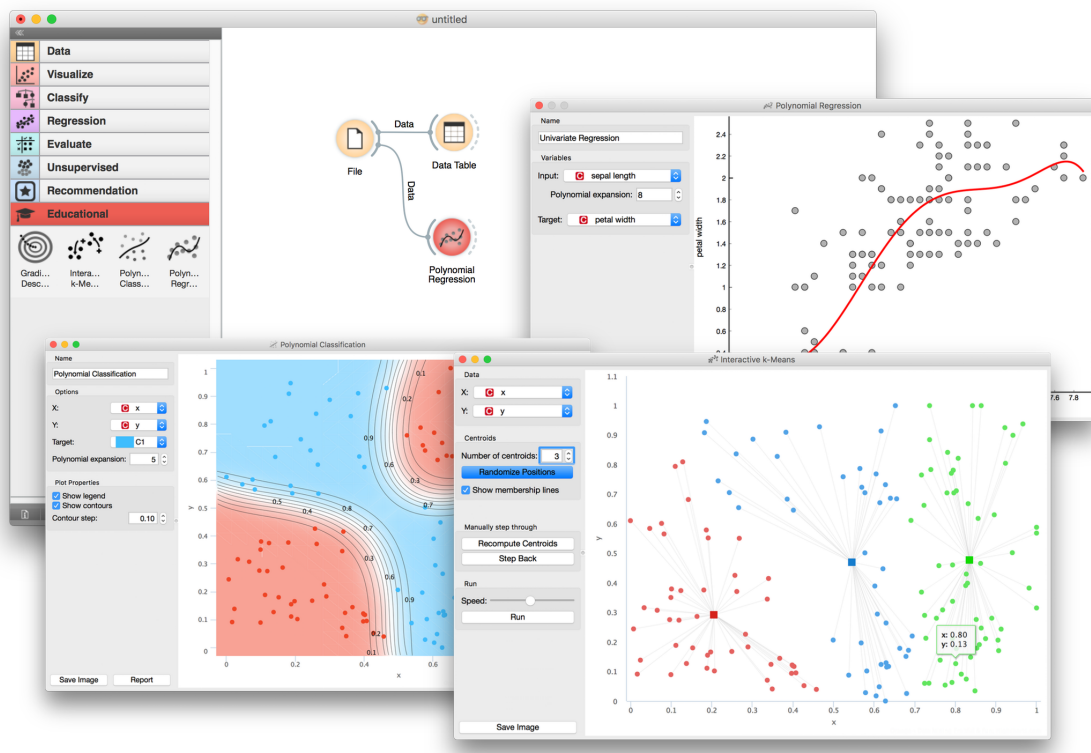
Additionally, a more comprehensive tutorial can be found at:

- **Blog:** <http://blog.biolab.si/2016/08/19/making-recommendations/>
- **Documentation:** <http://orange3-recommendation.readthedocs.io/en/latest/user/tutorial.html>

## 4.3 Orange3-Educational

*Orange3-Educational* is an educational add-on for machine learning and data mining in *Orange3*.

This add-on demonstrates several key data mining and machine learning procedures. Therefore, it is oriented towards beginners and teachers, so beginners can easily understand the inner working of key algorithms in the data mining, and teachers can visually explain these algorithms.



**Figure 4.18:** Orange3-Educational

### 4.3.1. Extension

In order to get a better understanding of the behavior of SGD optimizers under different set of problems, the *Gradient Descent* widget was extended to support this type of analysis.

Hence, it was extended to visualize the following optimizers: Vanilla SGD, Momentum, Nesterov Momentum, AdaGrad, RMSProp, AdaDelta, Adam and Adamax.

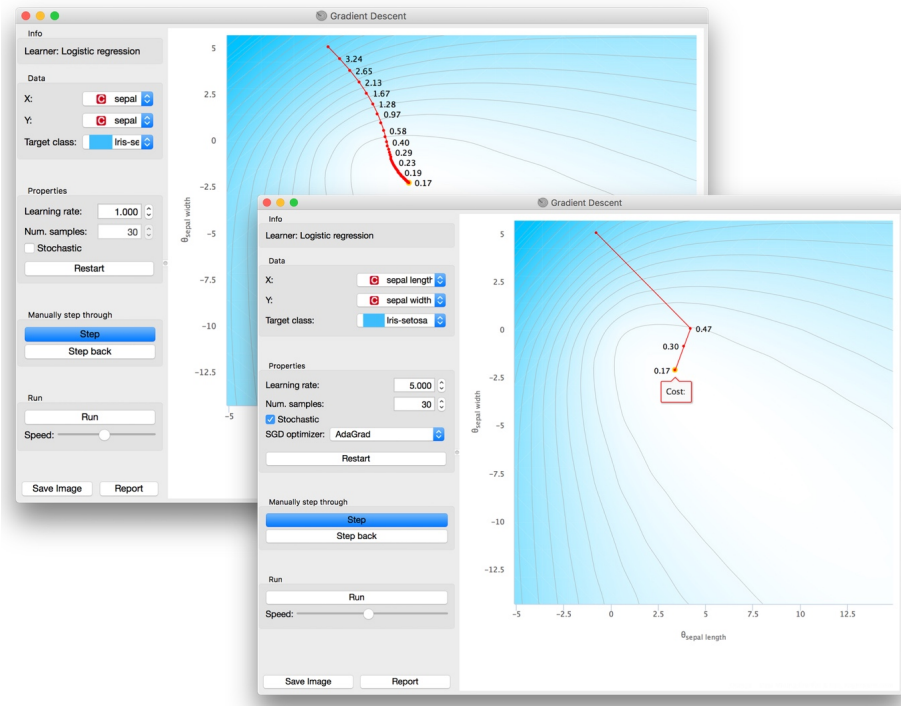


Figure 4.19: Gradient Descent Widget

So that these counter lines could be plotted, we have used an algorithm known as *Marching squares*, which allow us to generate contours for a two-dimensional scalar field[36]. The exact description of this algorithm is out of the scope of this project, but to grasp the intuition under it, it could be summarized as:

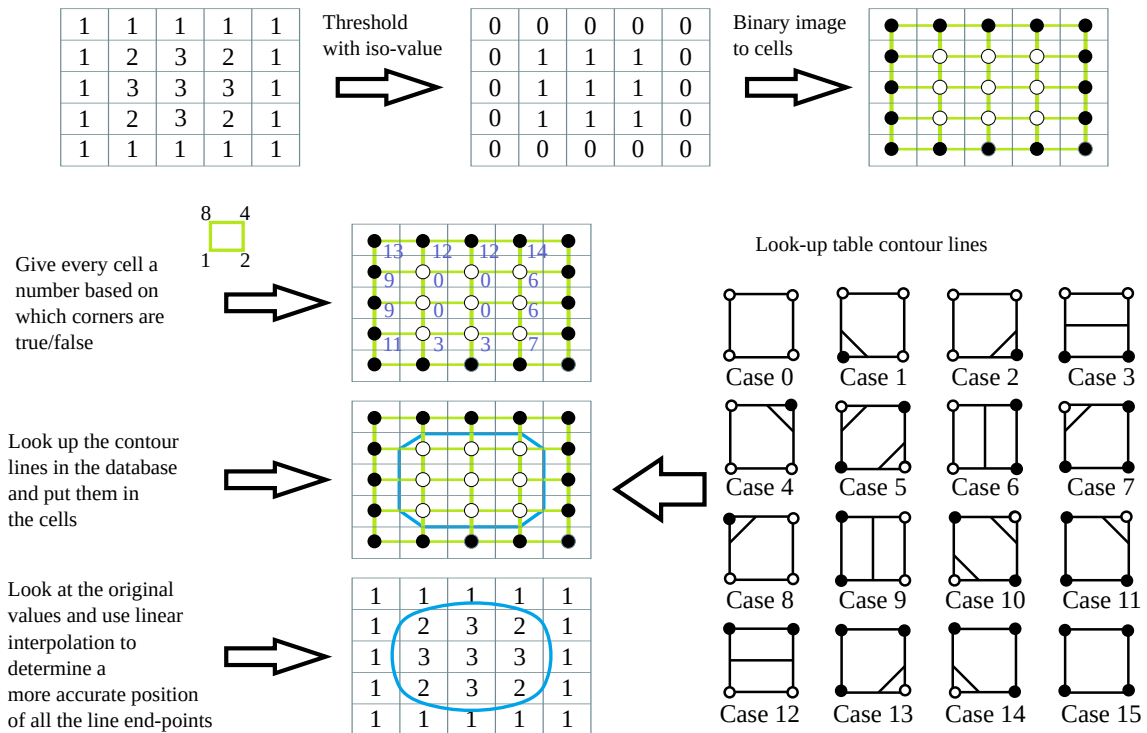


Figure 4.20: Marching squares algorithm[36]

---

---

# CHAPTER 5

## Experiments and Results

---

### 5.1 Datasets

---

Most of the algorithms have been tested with the main datasets used in each one of the original papers in which the model was presented. In addition, we have benchmarked all the algorithms implemented using popular datasets that exploit specific strengths of each model.

**Table 5.1:** Datasets overview

Dataset	Basic data					User context			Source
	Users	Items	Ratings	Scale	Density	Users	Links	Type	
<i>FilmTrust</i>	1,508	2,071	35,497	[0.5, 4.0]	1.14%	1,642	1,853	Trust	LibRec
<i>MovieLens100K</i>	943	1,682	100,000	[1, 5]	6.30%	-	-	-	GroupLens
<i>MovieLens1M</i>	6,040	3,706	1,000,209	[1, 5]	4.47%	-	-	-	GroupLens
<i>MovieLens10M</i>	71,567	10,681	10,000,054	[1, 5]	1.308%	-	-	-	GroupLens
<i>Epinions*</i>	4,718	49,288	23,590	[0*, 1]	0.0101%	-	-	-	Trustlet

### 5.2 Performance Comparison

---

In this section, I benchmark the implemented models in *Orange3* against popular datasets. All initializations have been computed in a deterministic way using 42 as a random seed. The reason behind this is simply to consistently evaluate the performance of the model without the interference of stochastics. In other words, by using a fixed seed, we make random numbers predictable. Therefore, we can compare results in a deterministic way.<sup>1</sup>

#### 5.2.0.1. Rating

**5.2.0.1.1. FilmTrust** *FilmTrust* is a small dataset crawled from the entire *FilmTrust* website in June, 2011.<sup>2</sup>

This dataset contains user-item ratings and a trust-network matrix made up by trusters and trustees.

Additional information:

- **Loading time:** 0.748s
- **Training dataset:** users=1,508; items=2,071; ratings=35,497; sparsity: 1.14%

---

<sup>1</sup>More information: <http://orange3-recommendation.readthedocs.io/en/latest/performance/benchmarks.html>

<sup>2</sup>Librec.net

- **Optimization:** No optimizers for SGD were used.

**Table 5.2:** Filmtrust benchmark results

Algorithm	RMSE	MAE	Train time	Settings
<i>Global Average</i>	0.919	0.715	0.000s	-
<i>Item Average</i>	0.861	0.674	0.000s	-
<i>User Average</i>	0.785	0.606	0.000s	-
<i>User-Item Baseline</i>	0.738	0.566	0.001s	-
<i>BRISMF</i>	0.712	0.551	0.820s/iter	num_factors=10; num_iter=15; learning_rate=0.01; lmbda=0.1
<i>SVD++</i>	0.707	0.546	1.974s/iter	num_factors=10; num_iter=15; learning_rate=0.01; lmbda=0.1;
<i>TrustSVD</i>	0.677	0.520	3.604s/iter	num_factors=10; num_iter=15; learning_rate=0.01; lmbda=0.12; social_lmbda=0.9

**5.2.0.1.2. MovieLens100K** *GroupLens Research has collected and made available rating data sets from the MovieLens website (<http://movielens.org>). The data sets were collected over various periods of time, depending on the size of the set<sup>3</sup>.*

Basically, it contains user-item information, timestamps, and a few demographic parameters.

Additional information:

- **Loading time:** 0.748s
- **Training dataset:** users=943; items=1,682; ratings=100,000; sparsity: 6.30%
- **Optimization:** No optimizers for SGD were used.

**Table 5.3:** MovieLens100K benchmark results

Algorithm	RMSE	MAE	Train time	Settings
<i>Global Average</i>	1.126	0.945	0.001s	-
<i>Item Average</i>	1.000	0.799	0.001s	-
<i>User Average</i>	1.031	0.826	0.001s	-
<i>User-Item Baseline</i>	0.938	0.738	0.001s	-
<i>BRISMF</i>	0.810	0.642	2.027s/iter	num_factors=15; num_iter=15; learning_rate=0.07; lmbda=0.1
<i>SVD++</i>	0.823	0.648	7.252s/iter	num_factors=15; num_iter=15; learning_rate=0.02; bias_learning_rate=0.01; lmbda=0.1; bias_lmbda=0.007

**5.2.0.1.3. MovieLens1M** *GroupLens Research has collected and made available rating data sets from the MovieLens website (<http://movielens.org>). The data sets were collected over various periods of time, depending on the size of the set<sup>3</sup>.*

Basically, it contains user-item information, timestamps, and a few demographic parameters.

Additional information:

- **Loading time:** 5.144s
- **Training dataset:** users=6,040; items=3,706; ratings=1,000,209; sparsity: 4.47%
- **Optimization:** No optimizers for SGD were used.

<sup>3</sup>GroupLens.org



**Table 5.4:** MovieLens1M benchmark results

Algorithm	RMSE	MAE	Train time	Settings
<i>Global Average</i>	1.117	0.934	0.010s	-
<i>Item Average</i>	0.975	0.779	0.018s	-
<i>User Average</i>	1.028	0.823	0.021s	-
<i>User-Item Baseline</i>	0.924	0.727	0.027s	-
<i>BRISMF</i>	0.886	0.704	19.757s/iter	num_factors=15; num_iter=15; learning_rate=0.07; lmbda=0.1
<i>SVD++</i>	0.858	0.677	98.249s/iter	num_factors=15; num_iter=15; learning_rate=0.02; bias_learning_rate=0.01; lmbda=0.1; bias_lmbda=0.007

**5.2.0.1.4. MovieLens10M** *GroupLens Research has collected and made available rating data sets from the MovieLens website (<http://movielens.org>). The data sets were collected over various periods of time, depending on the size of the set<sup>30</sup>.*

Basically, it contains user-item information, timestamps, and a few demographic parameters.

Additional information:

- **Loading time:** 55.312s
- **Training dataset:** users=71,567; items=10,681; ratings=10,000,054; sparsity: 1.308%
- **Optimization:** No optimizers for SGD were used.

**Table 5.5:** MovieLens10M benchmark results

Algorithm	RMSE	MAE	Train time	Settings
<i>Global Average</i>	1.060	0.856	0.150s	-
<i>Item Average</i>	0.942	0.737	0.271s	-
<i>User Average</i>	0.970	0.763	0.293s	-
<i>User-Item Baseline</i>	0.877	0.677	0.393s	-
<i>BRISMF</i>	-	-	230.656s/iter	num_factors=15; num_iter=15; learning_rate=0.07; lmbda=0.1

## 5.2.0.2. Ranking

**5.2.0.2.1. Epinions** This dataset is a modified version of the original Epinions dataset that only contains a binary relation of user-item interaction. In other words, it's a sparse boolean matrix which known interactions are set to 1, as opposed to missing information that remains unknown with 0 as an implicit value.

Additional information:

- **Loading time (training dataset):** 0.094s
- **Loading time (test dataset):** 1.392s
- **Training dataset:** users=4,718; items=49,288; ratings=23,590; sparsity: 0.0101%
- **Testing dataset:** users=4,718; items=49,288; ratings=322,445; sparsity: 0.1386%
- **Optimization:** No optimizers for SGD were used.

**Table 5.6:** Epinions benchmark results

Algorithm	MRR (train)	MRR (test)	Train time	Settings
<i>CLiMF</i>	0.0758	0.3975	1.323s/iter	num_factors=10; num_iter=10; learning_rate=0.0001; lmda=0.001

## 5.3 Visualization techniques

### 5.3.1. SGD optimizers

As it has been previously discussed in section 4.3.1, this extension allows us to visualize the behavior of different optimization techniques under different problems and configurations.

Generally, there is a trade-off between speed and determinism, this means that stochastic approaches converge much faster than deterministic ones but at the same time we can have more problems if not done correctly. In practice, SGD approaches are not an option so we have to play around with different optimizers and tunings to find one that suits our needs.

For example, these are a few experiments that we have done, initializing the solution always at the same point. It is not hard to see that some SGD optimizers converge much faster and smoothly than others, but this can vary depending on the problem.

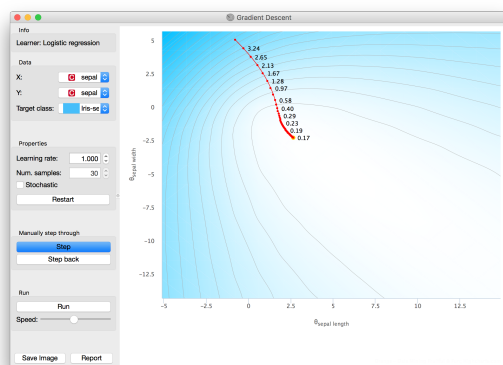


Figure 5.1: Gradient Descent;  $\alpha = 1.0$

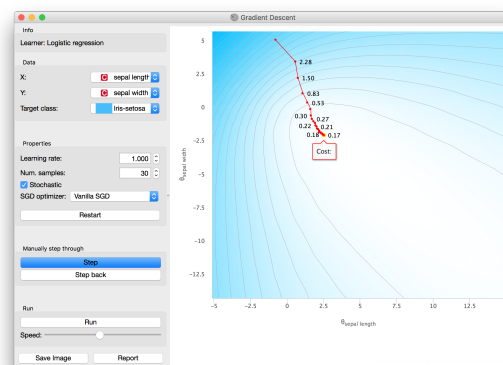


Figure 5.2: Vanilla SGD;  $\alpha = 1.0$

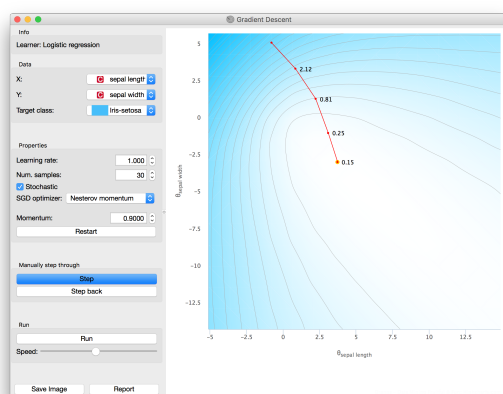


Figure 5.3: Nesterov Momentum;  $\alpha = 1.0$

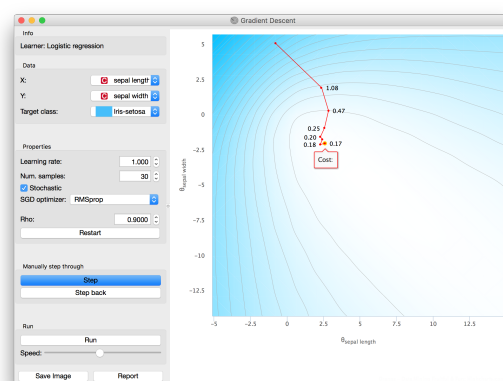


Figure 5.4: RMSProp;  $\alpha = 1.0$

If we increase the learning rate, we can find a (local) minimum in just a few iterations, but we have to be careful because if the learning rate is too large, the solution might diverge.

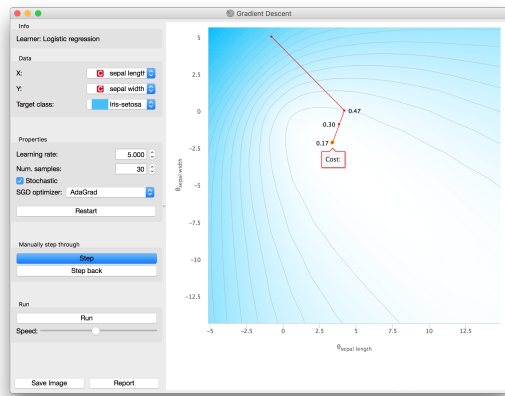


Figure 5.5: AdaGrad;  $\alpha = 5.0$

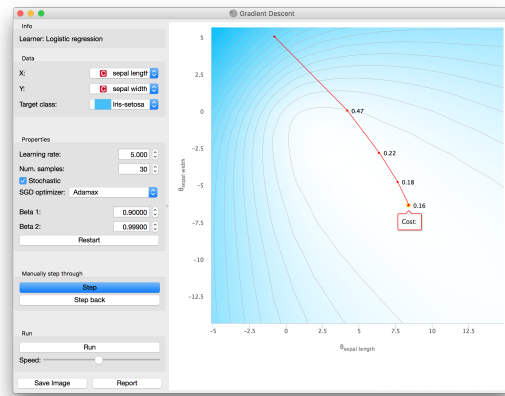


Figure 5.6: Adamax;  $\alpha = 5.0$

Now, let's take a look at these two new maps where we have changed the loss surface and a few settings. For the one on the left, the optimizer finds the minimum directly with no problems. On the contrary, the optimizer used in the second image first diverges and then corrects its deviation reaching the minimum again. (This last event might not happen in a different scenario)

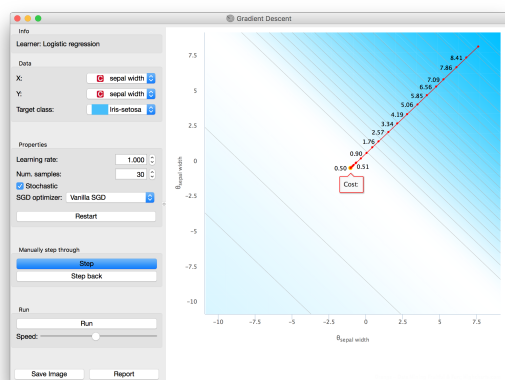


Figure 5.7: Different loss map

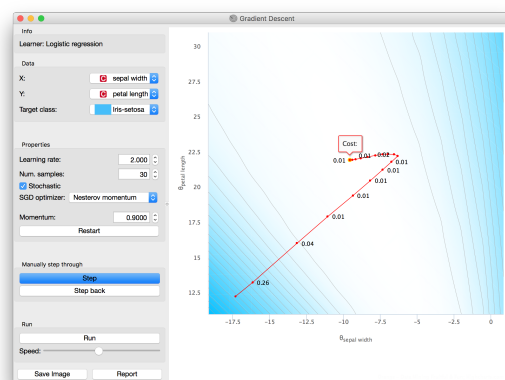


Figure 5.8: Small divergence

### 5.3.2. Latent factors

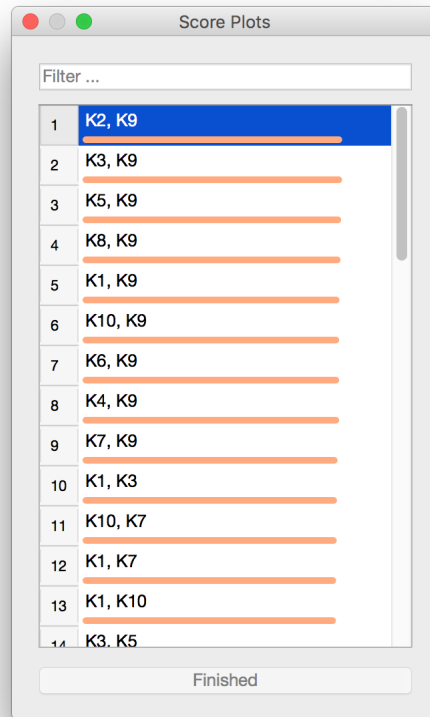
There are many ways to analyze the resulting low-rank matrices of the trained models, so here we are going to describe the very basic idea of this analysis.

First of all, we need to know what do we want to discover. For the sake of this example, we are going to try to find whether movie genres cluster in a specific way or not.

To do so, in the image below we have plotted just the top  $k$  most popular movies in 2-dimensions to make it easier to find patterns. Once they are plotted, we can use a different color to identify a specific genre category. In addition, we can play around with its settings by changing the colored category and the latent factors used as axes so that new patterns can emerge.

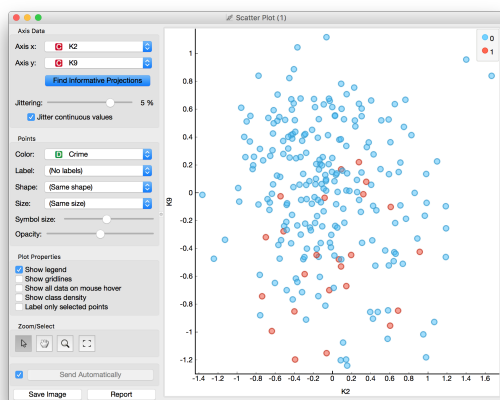
Of course this is a pretty simple technique just to do an exploratory analysis and see how difficult could be to extract relevant information from this set.



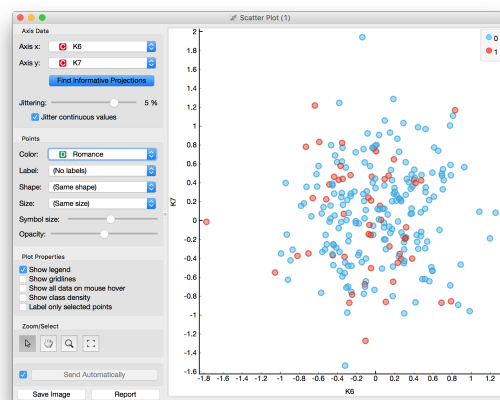


**Figure 5.12:** Informative projections

Now that we have scored the latent factors as axis (for a specific category), we can repeat the initial analysis using the resulting axes so that we can take a cleaner look at our visualization.

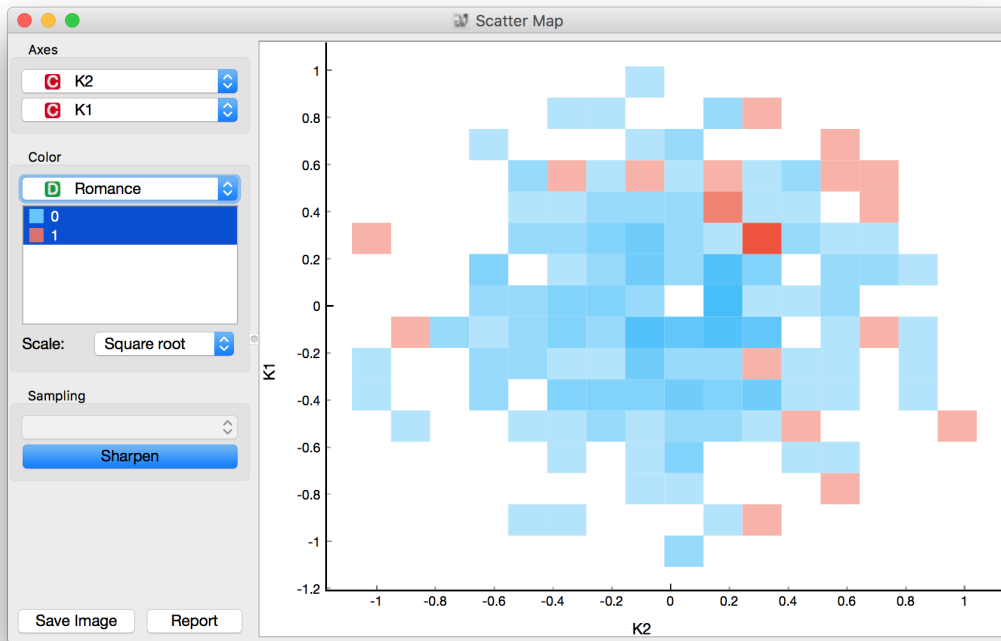


**Figure 5.13:** Ranked clusters (Crime)



**Figure 5.14:** Ranked clusters (Romance)

If this does not result, we can try using different visualization techniques such as heatmaps, dendrograms, sieve diagrams,...



**Figure 5.15:** Scatter map

Finally, if any of the above techniques results in solid conclusions or patterns, then we can work in the normalization of the low-rank matrices, side information, modify programmatically some values, combine interesting data with our results,... and so on, until either we find an interesting pattern or we prove that we cannot extract relevant information using this model.

Additionally, I'd like to point out that a Non-Negative Matrix Factorization tends to work better for this kind of analysis, but at the same time there are some other properties that we might lose.

# Conclusions and future work

---

This work proposed a research plan for Collaborative Filtering (CF) along with the development of a recommendation library for *Orange3 Data Mining*.

First I introduced the state-of-the-art for CF as well as the main types of CF, challenges, models for matrix factorization (MF), evaluation and optimization techniques. Then I described the architecture and implementation of a library for CF, and a few non-technical aspects related with its diffusion in an open source community. Additionally, I presented an extension for *Orange3-Educational* that allows to visualize the behavior of several SGD optimizers under different problems and settings. Finally, I benchmarked my library using the same datasets as the original papers in which the implemented algorithms were described for the first time. And later, I described a few visualization techniques for the stochastic optimizers in order to perform analysis over the resulted latent factors.

Future work involves a few interesting directions. First, I would like to develop a model to suit domains with side information, explicit and implicit trust information, and also, able to take into account the temporal dynamics. The intended model could be initially defined as something like this:

$$\hat{r}_{ui}(t) = \mu + b_u(t) + b_i(t) + q_i^\top \left( p_u(t) + \frac{1}{\sqrt{|N(u)|}} \sum_{j \in N(u)} y_j + \frac{1}{\sqrt{|T(u)|}} \sum_{v \in T(u)} w_v \right) \quad (6.1)$$

Second, it is interesting to transform the previously described model to be able to optimize it by maximizing the mean reciprocal rank (MRR) as it is done in *CLiMF*. In order to do so, we need to smooth the reciprocal rank function. To achieve this, we can take some direct continuous approximations such as the logistic function to transform the previous discrete function to a continuous function. Then, we just need to make sure that this function can be now optimized using gradient-based approaches. A way to work around this problem is by deriving a lower bound, therefore, applying the *Jensen's inequality* and taking advantage of the concavity and monotonicity of logarithms we can check for the existence of this bound. Finally, we can simply maximize this function or minimize its negative version.





# Bibliography

---

- [1] Loren Terveen and Will Hill. *Beyond Recommender Systems: Helping People Help Each Other*. Addison-Wesley. p. 6. Retrieved 16 January 2012.
- [2] Burr Settles. *Active Learning Literature Survey*. Computer Sciences Technical Report 1648, 2010.
- [3] Alexandros Karatzoglou, Xavier Amatriain, Linas Baltrunas and Nuria Oliver. *Multiverse Recommendation: N-dimensional Tensor Factorization for Context-aware Collaborative Filtering*. ACM Recommender Systems 2010.
- [4] Daniel M. Fleder and Kartik Hosanagar. *Blockbuster Culture's Next Rise or Fall: The Impact of Recommender Systems on Sales Diversity*. 2007.
- [5] Xiaoyuan Su and Taghi M. Khoshgoftaar. *A survey of collaborative filtering techniques*. Advances in Artificial Intelligence archive, 2009.
- [6] Simon Funk. *Netflix Update: Try This at Home*. Dec, 2006 <http://sifter.org/~simon/journal/20061211.html>
- [7] Yehuda Koren. *Factorization Meets the Neighborhood: a Multifaceted Collaborative Filtering Model*. KDD 2008.
- [8] Guibing Guo, Jie Zhang, Neil Yorke-Smith. *TrustSVD: Collaborative Filtering with Both the Explicit and Implicit Influence of User Trust and of Item Ratings*. AAAI 2015.
- [9] Yehuda Koren. *Collaborative Filtering with Temporal Dynamics*. KDD 2009.
- [10] Steffen Rendle, Christoph Freudenthaler, Zeno Gantner and Lars Schmidt-Thieme. *Bayesian Personalized Ranking from Implicit Feedback*. UAI 2009.
- [11] Yue Shi, Alexandros Karatzoglou, Linas Baltrunas, Martha Larson, Nuria Oliver, Alan Hanjalic. *CLiMF: Learning to Maximize Reciprocal Rank with Collaborative Less-is-More Filtering*. RecSys 2012.
- [12] Mohsen Jamali, Martin Ester. *A Matrix Factorization Technique with Trust Propagation for Recommendation in Social Networks*. RecSys 2010.
- [13] Gábor Takács, István Pilászy, Bottyán Németh, Domonkos Tikk. *Scalable Collaborative Filtering Approaches for Large Recommender Systems*. RecSys 2012.
- [14] Stephen Gower. *Netflix Prize and SVD*. 2014.
- [15] Yehuda Koren, Robert Bell and Chris Volinsky. *Matrix Factorization Techniques for Recommender Systems*. IEEE 2009.
- [16] Badrul Sarwar, George Karypis, Joseph Konstan, John Riedl. *Item-based Collaborative Filtering Recommendation Algorithms*. WWW10 2001.

- [17] Sebastian Ruder. *An overview of gradient descent optimization algorithms*. 2016.
- [18] Toby Segaran. *Programming Collective Intelligence. Building Smart Web 2.0 Applications*. O'Reilly Media, August 2007, ISBN 978-0-596-52932-1.
- [19] Jure Leskovec, Anand Rajaraman, Jeffrey David Ullman. *Mining Of Massive Datasets*. Wiley India, 2nd edition, 2016, ISBN-13 978-1316638491.
- [20] Ian H. Witten, Eibe Frank, Mark A. Hall. *Data Mining: Practical Machine Learning Tools and Techniques*. Morgan Kaufmann, 3rd edition (January 20, 2011), ISBN-13 978-0123748560.
- [21] John Duchi, Elad Hazan, Yoram Singer. *Adaptive Subgradient Methods for Online Learning and Stochastic Optimization*. JMLR 2011.
- [22] Tieleman, T. and Hinton, G. [Coursera] *Neural Networks for Machine Learning, Lecture 6.5 - rmsprop*. 2012. <http://americanhistory.si.edu/comphist/pr1.pdf>.
- [23] Matthew D. Zeiler. *ADADELTA: An Adaptive Learning Rate Method*. 2012.
- [24] Diederik Kingma, Jimmy Ba. *Adam: A Method for Stochastic Optimization*. ICLR 2015.
- [25] Zico Kolter *Convex Optimization Overview*, Oct. 2008.
- [26] Bro. David E. Brown *The Hessian matrix: Eigenvalues, concavity, and curvature*, BYU-Idaho Dept. of Mathematics. 2014.
- [27] Sebastien Bubeck. *Nesterov's Accelerated Gradient Descent* <https://blogs.princeton.edu/imabandit/2013/04/01/acceleratedgradientdescent/>
- [28] Sebastien Bubeck. *Revisiting Nesterov's Acceleration* <https://blogs.princeton.edu/imabandit/2015/06/30/revisiting-nesterovs-acceleration/>
- [29] Glenn Fox *Collecting and Connecting Millions of Opinions* <http://blog.ranker.com/category/data/data-science>
- [30] Orange3 <https://Github.com/biolab/orange3/wiki/Google-Summer-of-Code>
- [31] Wikipedia *Netflix Prize*. [https://en.wikipedia.org/wiki/Netflix\\_Prize](https://en.wikipedia.org/wiki/Netflix_Prize)
- [32] Wikipedia *Learning to rank* [https://en.wikipedia.org/wiki/Learning\\_to\\_rank](https://en.wikipedia.org/wiki/Learning_to_rank)
- [33] Brett Romero *Data Science: A Kaggle walkthrough - Creating a model* <http://bretttromero.com/wordpress/data-science-kaggle-walkthrough-creating-model/>
- [34] Wikipedia *Maxima and minima* [http://commons.wikimedia.org/wiki/File:Extrema\\_example.svg](http://commons.wikimedia.org/wiki/File:Extrema_example.svg)
- [35] Wikipedia *Saddle point* [https://en.wikipedia.org/wiki/Saddle\\_point](https://en.wikipedia.org/wiki/Saddle_point)
- [36] Wikipedia *Marching squares* [https://en.wikipedia.org/wiki/Marching\\_squares](https://en.wikipedia.org/wiki/Marching_squares)