What Should I Watch Next?

Federico Testa

Università degli Studi di Torino

Academic year 2023/2024

Contents

Chapter 1

The Problem and the Model

1.1 Introduction

In today's digital age, consumers are inundated with choices. E-commerce platforms and streaming services offer a vast selection of products and content, providing unprecedented opportunities to meet diverse needs and tastes. For retailers and content providers, matching users with the most appropriate products or content is crucial for ensuring both user satisfaction and loyalty.

Recommender systems analyze user interaction patterns and feedback to ensure good recommendations and they have gained significant attention as powerful tools for this purpose. These systems help users navigate through the overwhelming amount of available options by suggesting items that align with their preferences. Clearly, they find applications in a wide array of contexts.

This project focuses on analyzing movie recommender systems, which are widely used in popular streaming platforms and have a rich literature of methods and algorithms. The popularity of movie recommenders can be traced back to Netflix's 2006 challenge, which offered a million-dollar prize for beating their Cinematch recommendation algorithm. This competition fueled extensive research and innovation in the field over the three years of its duration, and it's a rich literature of methods and approaches.

1.1.1 Our Problem

The problem we will tackle is the following: assume we have access to a vast movie library and user base. Based on user ratings, we aim to make suitable movie recommendations for a chosen user, focusing on the "top-N recommendations" problem, which simply means that we care about the quality of the first N recommended titles. This scenario is common when a streaming service provides a ”You May Also Like” selection of titles, usually between 5 and 20, and we will output $N = 10$ recommendations. It will be clear soon that the choice of N, in the context of movies, is not really influential in the choice and the evaluation of the algorithm. Intuitively, this is because movie recommendations are meant to propose some suitable choices that offer some variety and may suit the needs of the user at the specific moments. The order between those N recommendation is not particularly relevant *per se*, compared to the presence of a good selection of titles. To an extent, the serendipity of a recommendation can also be an appreciable feature. This consideration will guide our choice of metrics for evaluating the quality of our algorithm.

1.2 Recommendation Systems: Overview of Approaches

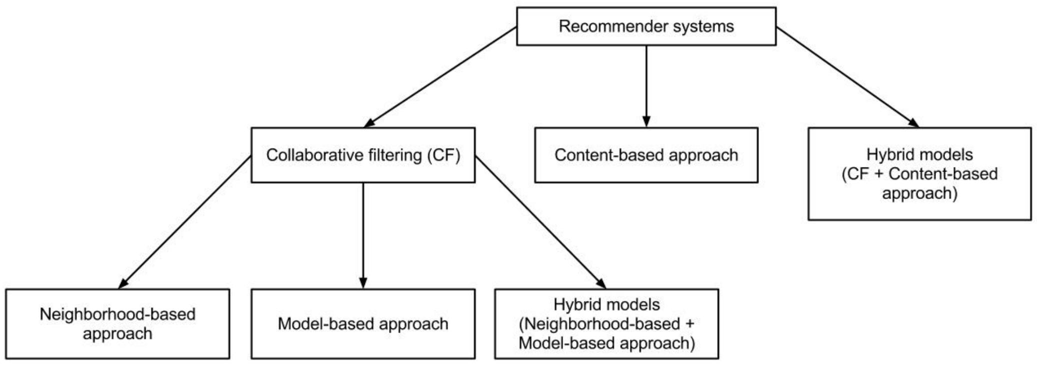
Recommender systems
Collaborative filtering (CF)
Content-based approach
Hybrid models (CF + Content-based approach)
Neighborhood-based approach
Model-based approach
Hybrid models (Neighborhood-based + Model-based approach)

Figure 1.1: Overview of approaches for recommender systems ([1]).

When tasked with recommending items to users, two main strategies have emerged: *content-based filtering* and *collaborative filtering*.

1.2.1 Content-Based Filtering

Content-based filtering is a straightforward approach that matches user preferences with item attributes. This method involves gathering user information, such as demographic details and questionnaire responses, to identify preferences and recommend items with compatible attributes. For example, a young user expressing a preference for superhero movies with thrilling action and stunning effects might be recommended something from the Marvel's Avengers saga. Conversely, a cult-film enthusiast who has also watched all other Tom Hanks movies but not "Forrest Gump" might receive that recommendation.

Content-based filtering relies heavily on accurate domain knowledge and may struggle to capture intricate or less evident drivers of user preferences.

1.2.2 Collaborative Filtering

The alternative approach, which has become dominant in recent years, is *collaborative filtering.* Collaborative filtering methods leverage user-item interaction data to make recommendations, without requiring detailed item attributes or explicit user profiles. These methods identify patterns and similarities among users and items to suggest content that similar users have enjoyed. A major appeal of this approach is that it is domain free and can potentially address data aspects that are difficult to profile using content based filtering. Collaborative filtering approaches can themselves be split into two major classes: *neighbourhood methods* and *latent factor methods.*

Neighbourhood Methods

Neighbourhood methods compute relationships between either items or users. Item-oriented methods predict the rating of an item based off of the ratings of other items they deem similar in some sense to the one of interest. A user-oriented approach, when tasked with predicting a rating for a user, will compute the neighbourhood of users that are "close" to the user of interest in terms of rating history and will propose the movies those users viewed frequently and rated highly.

Latent Factor Models

Latent factor models gained popularity during the Netflix Prize challenge and have since been investigated until state-of-the-art performance have been achieved by different variants. Essentially, we try to explain ratings by characterizing users and movies on a lower (compared to the number of users and movies) dimensional space, in which coordinates of users and movies, both mapped into the same space, are called factors and are inferred by the patterns observed in the ratings. We can think of latent factors as an attempt to infer a small number of relevant features in movies that allow to describe accurately a user's profile based on its liking to these factors and allows for direct comparison between users and movies as elements of the same vector space.

The most popular way to approach a latent factor model is *matrix factorization*. We will go into the details of the reasoning and formalization in the next section. It is worth mentioning that some alternatives have arisen, such as Neural Collaborative Filtering, where latent factors are captured using a neural network.

The method we will analyze in details in this project, Alternating Least Squares (introduced by [2]), falls into the category of latent factor models based on matrix factorization.

1.3 Matrix Factorization

Let us set some notation. Suppose we are considering m movies and n users interacting with them through ratings. Ratings are (ordinal) categorical variables, as usually we do not allow users to evaluate a movie on a continuous scale, but rather on a discrete grid between the minimum and the maximum rating. We will follow the rating system proposed in the MovieLens dataset on which the algorithm will be tested: ratings can take 10 values in the range from 0.5 to 5, with a 0.5 increase between one rating and the next.

We will use generic variables u, v to refer to users and i, j to refer to movies. Let us denote by $\mathcal{R} = \{(u, i) : \text{user } u \text{ rated movie } i\}$ and let r_{ui} be the observed rating associated to a couple $(u, i) \in \mathcal{R}$. Let us consider the user-item rating matrix $R \in \mathbb{R}^{n \times m}$ whose rows represent users and columns represent movies. We define its entries as

$$R_{ui} = \begin{cases} r_{ui} & \text{if } (u, i) \in \mathcal{R} \\ 0 & \text{else} \end{cases}$$

where the default value for missing ratings is set to zero, which is not allowed as a rating value. Naturally, users will watch only a subset of the large amount of movies and will rate an even smaller amount. This implies that in most instances, as will also happen in our dataset too, the matrix R is sparse: $|\mathcal{R}| << nm$.

Our goal is to infer all the missing rating from the available one, performing a "sparse-matrix completion" on the user-ratings matrix, obtaining the "full rating matrix" R_{full}. This provides a simple solution to the top-N recommendation problem: if we predict for each user u all ratings for all movies, we just need to sort the predicted ratings for the movies the user has not yet rated in descending order, and propose the first N.

The latent factor model relies on a *low-rank assumption* for matrix R_{full}: we assume the rating matrix can actually be well approximated by as a product $R_{full} \approx XY^T$ where $X \in \mathbb{R}^{n \times f}$ and $Y \in \mathbb{R}^{m \times f}$, for some $f << \min\{n, m\}$. This dimension f refers to the aforementioned latent factor dimension. The interpretation is the following: each user u can be represented as a vector $x_u \in \mathbb{R}^f$ and each movie as a vector $y_i \in \mathbb{R}^f$, both fully characterized by their coordinates in the latent space. In this representation, users and movies can be directly compared one with the other and ratings arise as an inner product between the two factors vectors:

$$r_{ui} \approx x_u^T y_i \quad \forall (u, i)$$

The main difference across matrix factorization approaches is the way X and Y are computed.

Many variations have been proposed, and the most popular are Asymmetric SVD, SVD++ and Alternating Least Squares.

The main reasons we choose ALS are that

- it's simple, yet effective: other methods, like asymmetric SVD and hybrid methods, are more complex and computationally intensive (especially if new movies or users are added), for little gains in terms of performance;

- it scales well: we will discuss the computational cost in details, but is worth mentioning that ALS is easily parallelized, making it an industry-favourite in this era (thanks also to the efficient Apache Spark implementation);

- it handles new entries well: it is not essential to our application, where users and movies are given and fixed, but ALS by design allow to adapt to "small" additions of ratings, users and movies with little computations.

Chapter 2

The Algorithm

2.1 ALS explained

Alternating Least Squares, introduced by [2], is a state-of-the-art method that is employed in large scale systems in different recommender systems, both for explicit feedback (like ratings) and implicit feedback (like clicks, interactions, number of times a song is listened to, ...). We will use a the regularized version ALS-WR, an improvement on the baseline model, proposed by [3].

Recall the goal of building a low-rank representation of the partially observed rating matrix $R \approx \hat{R} := XY^T$ with $X \in \mathbb{R}^{n \times f}$ is the user-factor matrix and $Y \in \mathbb{R}^{m \times f}$ is the movie-factor matrix, with $f << n \wedge m$ a latent dimension to be determined. Differently from canonical factorization, like SVD, we actually focus on accurately reconstructing R through $\hat{R}$ only for the actual ratings $\{r_{ui} : (u, i) \in \mathcal{R}\}$. Therefore, we consider the objective function to be minimized:

$$F(X, Y) = \sum_{(u,i) \in \mathcal{R}} (r_{ui} - x_u^T y_i)^2 + \lambda \left(\sum_u n_u ||x_u||^2 + \sum_i n_i ||y_i||^2 \right)$$

where the second term is a regularization applied to avoid overfitting, with λ to be determined, n_u the number of ratings for user u and n_i the number of ratings for movie i. We could also employ a standard regularization procedure using the squared L^2 norm of the factors, but these additional weights, according to the literature, make for a more robust regularization and give the name to the "ALS-WR" variant. Intuitively, they penalize the norm of the factors for users who rated a lot of movies and movies with a lot of ratings, preventing them from being too influential.

The procedure to compute the factors has just 4 steps:

1. Initialize matrix Y with just small random numbers except for the first row, where we use the average rating for that movie;
2. Fix Y, solve the minimization $X = \arg\min_A F(A, Y)$;
3. Fix X, solve the minimization $Y = \arg\min_A F(X, A)$;
4. Repeat steps 2 and 3 until a stopping criterion is reached.

The initialization is a good heuristic rule that has been found to be effective.

Once these factors have been computed, we can easily compute a predicted rating from a combination of user-item for which we do not have a rating. Let $(u, i) \notin \mathcal{R}$, the predicted rating will be $\hat{r}_{ui} = x_u^T y_i$. So, simply put, we make use of the indices in $\mathcal{R}$ to accurately reproduce existing rating and we get "for free" a dense matrix of predicted ratings $\hat{R} = XY^T$. Then, for a given user u, the top-N recommendations will simply be the columns achieving the highest predicted rating in the row $\hat{R}_{u\cdot}$. That is, the top-N recommendations will be

$$\begin{aligned} i_1 &= \underset{i \text{ s.t. } (u,i)\notin\mathcal{R}}{\arg\max} \; \hat{R}_{ui} \\ i_n &= \underset{i\neq i_1,\dots,i_{n-1} \text{ s.t. } (u,i)\notin\mathcal{R}}{\arg\max} \; \hat{R}_{ui} \quad \text{for } n = 2, 3, ..., N \end{aligned}$$

2.1.1 Alternated Least Square Optimization

The main advantage of ALS over other matrix factorization approaches is that by focusing on a single matrix of factors at a time the optimization is a least square problem with an explicit solution.

By differentiating the loss, we find that the minimum for Y fixed is achieved at user factor $x_u, j = 1, 2, ..., m$ given by:

$$x_u = (Y_{I_u}^T Y_{I_u} + \lambda n_u I_{f\times f})^{-1} Y_{I_u} R(u, I_u)^T$$

where I_u is the set of movies rated by user u, $R(u, I_u)$ is the column vector of ratings (extracted from R) provided by user u_j and $I_{f\times f}$ is the $f \times f$ identity matrix.

By a symmetric argument

$$y_i = (X_{I_i}^T X_{I_i} + \lambda n_i I_{f\times f})^{-1} X_{I_i} R(I_i, i)$$

where I_i is the set of users that rated movie i, and so on.

2.2 Stopping Criterion

We will see that ALS converges to a satisfying solution in a very small number of iterations, usually less than 10. The simplest option for a stopping criterion would be to just fix a maximum number of iteration.

A less naive approach involves computing the loss we are considering on a validation dataset and employ an early stopping strategy: given an integer patience parameter p, we stop if after p iterations the loss on the validation set has not improved.

The validation set can be extracted from the original user-item matrix by selecting randomly ratings but ensuring that no row or column is left empty. Indeed, ALS like most collaborative filtering methods suffers from the "cold-start" problem: it's incapable to make a recommendation to completely new users and movies, where no history is available. However, this is not a problem: for new users, until ratings are observed, it's enough to design a simple content-based recommender that proposes the most popular movies or is based on information gathered through a questionnaire. For new movies, we can propose a "What's New?" section until ratings are observed.

2.3 Evaluation Metrics

To evaluate the model, we rely on the most commonly used loss for recommender system, which is also the one that was proposed by Netflix for the Netflix Prize Challenge: Rooted Mean Square Error (RMSE).

Given two sets of ratings $S = \{r_{ui}\}_{(u,i)\in\mathcal{A}}$ and $S' = \{r'_{ui}\}_{(u,i)\in\mathcal{A}}$, of sizes $|\mathcal{A}| = |\mathcal{A}'| = k$ we define:

$$\text{RMSE}(S, S') = \sqrt{\frac{1}{k}\sum_{(u,i)\in\mathcal{A}} (r_{ui} - r'_{ui})^2}$$

The loss can either be computed on training data, by comparing R with the predicted ratings $\hat{R}$ on non-missing data, or on the validation/test set (between predicted data and hold-out data). Essentially, for the evaluation for rely on a measure of the goodness of the global reconstruction of the ratings matrix: if we are able to predict accurately the missing ratings, we can make good top-N recommendations.

Chapter 3

Analysis of the Algorithm and of the Complexity

In the notebook, we built two classes that work together to provide the top-N recommendations. The first one is the `UserItemMatrix` class, which is built by providing a path to a `.csv` file of ratings with at least 3 columns: `userId`, `movieId` and `rating`.

It allows to build the user-item rating matrix using a sparse representation (and, for completeness, a dense one for toy examples) and to interact with it extracting all useful information for ALS (number of nonzero-ratings per user and per movie, the ratings themselves, ...). It also contains attributes to associate to `userId`'s and `movieId`'s the row and column index respectively in the matrix of ratings. As for movies, a third column is provided which allows to also convert form index or id to the movie name, for clarity of the recommendations.

The second class is the `ALS-WR` class, which is the one used to perform ALS algorithm on a `UserItemMatrix` object. It also stores the computed factors, that can be used to perform rating predictions, model evaluation on test data and, of course, top-N recommendations.

We write a pseudo-code of how the algorithm unfolds from initializing `ALS-WR` to printing the top-N recommendations, and then focus on each step and its complexity.

3.1 Pseudo-Code

We consider two versions of the algorithm:

- with a fixed number of iterations;

- with a validation set and early stopping.

In both scenarios the initialization step of movie factors is the same:

Algorithm 1: Initialization

```
Data: $R$ user-item matrix, $f$ latent factor dimension
Initialise Movie Factors
    Initialize $X$ as a $n \times f$ array.
    Initialize $Y$ as a $f \times m$ array where:
    $Y[0,:] \leftarrow \text{mean}(R, axis = 0)$ average ratings per movie
    $Y[i,:] \sim N_m(0,1)$ small random values
    Return $X, Y$
end
```

Both versions also share the same updates of user factors and movie factors:

Algorithm 2: User Factor Update

```
Data: $R$ user-item matrix, $Y$ movie factors, $\lambda$ regularization
      parameter
Update User Factors
    For $u = 1, 2, ..., n$ users
        Compute $I_u$ vector of indices of movies rated by $u$
        $V \leftarrow Y[:, I_u]$
        $r \leftarrow R[u, I_u]$ ratings of the user
        $c \leftarrow |I_u|$ number of ratings for the user
        $X[u,:] \leftarrow (VV^T + \lambda cI)^{-1}Vr$
        Return $X$
    end
end
```

Algorithm 3: Movie Factor Update

```
Data: $R$ user-item matrix, $X$ user factors, $\lambda$ regularization
      parameter
Update Movie Factors
    For $i = 1, 2, ..., n$ movies
        Compute $I_i$ vector of indices of users who rated $i$
        $U \leftarrow X[I_i, :]$
        $r \leftarrow R[I_i, i]$ ratings of the movie
        $c \leftarrow |I_i|$ number of ratings for the movie
        $Y[:, i] \leftarrow (UU^T + \lambda cI)^{-1}Ur$
        Return $Y$
    end
end
```

In our object-oriented implementation we do not actually return the factors as they are stored as attributes of the class `ALS-WR` and we directly modify them.

Finally, the pseudo code for the recommendation step:

Algorithm 4: Top N Recommendations

```
Data: $R$ user-item matrix, $X$ user factors, $Y$ movie factors,
      $s = False$ boolean to include seen movies
Recommend Top $N = 10$ Movies
    $r \leftarrow X[u^*, :]Y$ predicted ratings
    Compute $I_{u^*}$ vector of indices of movies rated by $u^*$
    if $!s$ then
        $r[I_{u^*}] \leftarrow -\infty$
    end
    $M \leftarrow \text{argsort}(r)$ (indirect sorting, descending order)
    $M_N \leftarrow M[: N]$
    Print the movie titles corresponding to $M_N$.
    Return $M_N$
end
```

Combining the previous algorithms, with a fixed number of iterations the high-level pseudo-code is the following:

Algorithm 5: ALS-WR (max iter)

```
Data: $R$ user-item matrix, $u^*$ user to recommend to
Set parameters: $n_{iter}$ number of iterations; $\lambda$ regularization
    parameter; $f$ latent factor dimension
  Initialise Movie Factors
    Algorithm 1
  end
  For $t = 1, 2, ..., n_{iter}$
      Update User Factors
        Algorithm 2
      end
      Update Movie Factors
        Algorithm 3
      end
  end
  Recommend Top $N = 10$ Movies
    Algorithm 4, returns $M_N$ top-$N$ movie ID's
  end
  Print the movie titles corresponding to $M_N$
  Return $M_N$
```

In the implementation we also include the computation at each step of the training loss, which is then stored in a list and may be useful to monitor convergence but is not strictly needed. We do the same in the implementation of the second version (algorithm 7), which includes a validation procedure with early stopping and where we may want to plot the training vs validation loss to display both convergence and overfitting. The pseudocode for computing the loss at a given iteration on a given set of data (training or validation) is the following:

Algorithm 6: Compute the Training Loss

```
Data: $R$ user-item matrix (train or validation), $X$ current user
      factors, $Y$ current movie factors, $L$ current loss list
Compute RMSE
    $\ell = 0$
    For $u = 1, 2, ..., n$ users
        $I_u \leftarrow$ indices of movies rated by $u$
        $V \leftarrow Y[:, I_u]$
        $r \leftarrow R[u, I_u]$
        $\ell \leftarrow \ell + [\text{sum}(r - X[u, :]V)]^2$
        $\ell \leftarrow \sqrt{\frac{\ell}{|I_u|}}$
        $L$.append($\ell$)
    end
    Return $\ell$ (last value)
end
```

The validation loss is computed in a completely analogous manner, but predicting and comparing ratings from the validation set instead of the training set.

Finally, the pseudo-code for the version with the early stopping mechanism

Algorithm 7: ALS-WR (early stopping)

```
Data: $R$ training user-item matrix, $R_{\text{val}}$ validation user-item
      matrix $u^*$ user to recommend to
Set parameters: $n_{iter}$ max number of iterations; $p$ patience
  parameter; $\lambda$ regularization parameter; $f$ latent factor
  dimension;
  Initialise Movie Factors
    Algorithm 1
  end
  $c_p = 0$ patience counter
  $\ell_m = +\infty$ best loss
  For $t = 1, 2, ..., n_{iter}$
      Update User Factors
        Algorithm 2
      end
      Update Movie Factors
        Algorithm 3
      end
      Compute Validation Loss
        $\ell \leftarrow$ current loss from Algorithm 6
      end
      if $\ell < \ell_m$ then
        $\ell_m \leftarrow \ell$
        $c_p \leftarrow 0$
      end
      else
        $c_p \leftarrow c_p + 1$
      end
      if $c_p \geq p$ then
        Break
      end
  end
  Recommend Top $N = 10$ Movies
    Algorithm 4, returns $M_N$ top-$N$ movie ID's
  end
  Print the movie titles corresponding to $M_N$
  Return $M_N$
```

3.2 Complexity

We provide an analysis of the asymptotic complexity of each step, and then combine them to find the complexity of the two versions of the algorithm.

3.2.1 Sub-Module Complexities

Initializations

Before discussing algorithm 1, we should mention that we focus on the sparse instance of `UserItemMatrix`, as it has no downside in complexity for the required operations, but it has a significant gain in terms of memory occupation. For future reference, we mention the data structures it relies on and some of their pros:

- the user-item matrix is stored in two copies: in a Compressed Sparse Rows format and in a Compressed Sparse format (details at SciPy documentation). These formats for sparse matrices allow, respectively, for fast row slicing and fast column slicing. We found it more convenient than storing it in a single format and moving from one representation to the other when needed, since users and movies have a (almost) symmetric role.

- look-ups between ID's and indices of users and across ID's, indices and movie titles all happen in constant time. For users, a `bidict` object is used, which is a convenient way to handle a double dictionary, hence both ID's and indices can be treated as keys or values depending on the convenience. For movies, we use a table (`DataFrame`) to move from indices to titles and ID's, and a dictionary to link ID's (keys) with indices (values), ensuring all conversions in $O(1)$.

We assume the `UserItemObject` is already available (since it only performs some preprocessing and data manipulation) and focus on the `ALS-WR` algorithm. Let $n_r = |\mathcal{R}|$ be the number of available ratings for training. The initialization step (algorithm 1) requires initializing at random $nf + m(f-1)$ terms and computing the mean across columns of the user-item matrix. The first operation is $O((n+m)f)$, whereas the latter requires summing for each column the non-zero values, so the total complexity will be:

$$\sum_{i=0}^{m-1} \text{nnz}(R_{\cdot i}) \cdot O(1) = O(n_r)$$

So the total complexity of latent factor initialization is given by $O((n+m)f + n_r)$ or $O((n \vee m)f + n_r)$.

Regularization matrices are also initialized (so to compute them only once in the whole algorithm), but they are simply diagonal matrices whose elements are either 1 in the L2 case, and the number of ratings n_u and n_m for each user and movie. These latter values correspond to the nonzero values of row and columns and, for a fixed row or column, are accessed in constant time thanks to the sparse representation. So complexity is simply $O(n) + O(m) = O(n + m)$.

Resulting complexity for the initialization: $O((n + m)f + n_r)$ or $O((n \vee m)f + n_r)$.

Updating User Factors

We refer to algorithm 2. The Python code is

```python
def _update_user_factors(self):
    """
    Update user factors, movies are fixed.
    """
    for i in range(self.n_users):
        # get indices of non-zero ratings
        idx = self.user_item_matrix.where_nnz_user(
            self.user_item_matrix.get_user_id(i)
            )
        # get movie factors
        V = self.movie_factors[:, idx]
        # get ratings
        r = self.user_item_matrix.user_ratings(
            self.user_item_matrix.get_user_id(i)
            )
        # update user factors
        reg_i = self.reg_matrix_user[i,i]
        self.user_factors[i] = np.linalg.solve(
            V @ V.T + self.reg_param * reg_i, V @ r
            )
    return
```

Let us fix a user u. Thanks to the sparse representation in CSR and CSC format, accessing the vector of indices of non-zero elements in a row or column happens in constant time. The sliced copy of the factors is of size $f \times \text{nnz}(R_{u\cdot})$. The ratings vector instead will simply be of size $\text{nnz}(R_{u\cdot})$ and obtained in constant time, since the `.user_ratings()` method by default returns only the nonzero ratings which is part of the representation of the CSR format. It will be copied to be stored in r, requiring $\text{nnz}(R_{u\cdot})$ operations. All of these operations have a negligible cost asymptotically compared to the actual update of the factors.

Then, a linear system is solved. The matrix to be inverted is obtained as a multiplication of $f \times \text{nnz}(R_{u\cdot})$ and $\text{nnz}(R_{u\cdot}) \times f$ matrices, plus an additional term broadcast to a $f \times f$ shape which will be negligible in the complexity compare to the product. The matrix multiplication between dense matrices can be assumed to be $O(f^2 \cdot \text{nnz}(R_{u\cdot}))$ (even though in this case symmetry could be exploited to halve the number of operations). Matrix inversion for a $f \times f$ matrix requires $O(f^3)$ operations. The resulting $f \times f$ matrix, which is $(VV^T + \lambda n_u I)^{-1}$, is multiplied by V, requiring again $O(f^2 \cdot \text{nnz}(R_{u\cdot}))$. Finally, the multiplication by vector r requires $O(f \cdot \text{nnz}(R_{u\cdot}))$. To summarize, this linear algebra step for a fixed user has a complexity of $O(f^3 + f^2 \cdot \text{nnz}(R_{u\cdot}))$.

When we account for the iterations of the for cycle over n users:

$$\sum_{u=1}^{n} O(f^3 + f^2 \cdot \text{nnz}(R_{u\cdot})) = O(nf^3 + n_r f^2)$$

Updating Movie Factors

WE refer to the pseudocode 3. The code is analogous:

```
def _update_movie_factors(self):
    """
    Update movie factors, users are fixed.
    """
    for j in range(self.n_movies):
        # get indices of non-zero ratings
        idx = self.user_item_matrix.where_nnz_movie(
            self.user_item_matrix.get_movie_id(j)
            )
        # get user factors
        U = self.user_factors[idx]
        # get ratings
        r = self.user_item_matrix.movie_ratings(
            self.user_item_matrix.get_movie_id(j)
            )
        # update movie factors
        reg_j = self.reg_matrix_item[j,j]
        self.movie_factors[:, j] = np.linalg.solve(
            U.T @ U + self.reg_param * reg_j, U.T @ r
            )
    return
```

By a symmetric argument, the total complexity will be $O(mf^3 + n_r f^2)$.

Top N recommendations

We refer to pseudocode 4. The Python code is

```python
def recommend(self, user_id, n=10, include_seen=False, print=
                                True):
    """
    Recommend top n movies for a given user.

    Parameters:
    - user_id (int): User ID.
    - n (int): Number of movies to recommend (default: 10).
    - include_seen (bool): Flag to include movies already
                                seen by the user (default:
                                 False).
    """
    user_idx = self.user_item_matrix.get_user_idx(user_id)
    ratings = self.user_factors[user_idx] @ self.
                                movie_factors
    if include_seen:
        top_n = ratings.argsort()[-n:][::-1]
    else:
        idx = self.user_item_matrix.where_nnz_user(user_id)
        ratings[idx] = 0
        top_n = ratings.argsort()[-n:][::-1]
    if print:
        # print top n movie names
        for i in range(len(top_n)):
            print(f"Recommended {i+1}: {
                self.user_item_matrix.get_movie_name(top_n[i
                                ], from_idx=
                                True)
                }")

    # return top n movie ids, not idxs
    return self.user_item_matrix.get_movie_id(top_n).values
```

We will assume $N = O(1)$ since it's given and doesn't grow with the data. The complexity is given by:

- the first vector - matrix multiplication: $O(mf)$
- indices of rated movies, which are $n_u = \text{nnz}(R_{u\cdot})$, is accessed in constant time
- the if has a worst case of $O(n_u)$, a best of $O(1)$.
- sorting with an optimal algorithm is $O(m \log(m))$. By default, quick-sort is used. Any other optimal sorting could be used. Notice that

we only reverse the vector after slicing the last N elements, so this operation is $O(1)$.

- printing N elements is $O(N) = O(1)$.

The total complexity is given by $O(m(f + \log(m)))$.

Loss

Suppose n_r is the number of ratings of training data. Technically, for asymptotic analysis we could use n_r also in the case of validation data, since $n_v = 0.2 \cdot n_r$ or some fixed proportion of it. We refer to algorithm 6. Python code is

```python
def _compute_loss(self):
    """
    Compute the RMSE loss for the training data.
    """
    loss = 0
    for i in range(self.n_users):
        idx = self.user_item_matrix.where_nnz_user(
            self.user_item_matrix.get_user_id(i))
        V = self.movie_factors[:, idx]
        r = self.user_item_matrix.user_ratings(
            self.user_item_matrix.get_user_id(i))
        loss += np.sum((r - self.user_factors[i] @ V) ** 2)

    loss = loss / self.n_ratings
    loss = np.sqrt(loss) #empirical RMSE is computed

    self.training_loss.append(loss)
    return
```

For each user u with n_u ratings, computing the "row-MSE" requires a vector-matrix multiplication that is $O(fn_u)$, a vector difference that is of order $O(n_u)$ and summing over the entries of the resulting vectors for, again, $O(n_u)$. Given that this is repeated for all n users and the remaining operations are in constant time, the total complexity is $\sum_u O(n_u f) = O(n_r f)$.

3.2.2 Complexity of ALS with Fixed Number of Iterations

We refer to algorithm 5. We have:

- initialization: $O((n + m)f + n_r)$

- a for cycle of length n_{iter} with
  - an update of user factors: $O(nf^3 + n_r f^2)$
  - an update of movie factor: $O(mf^3 + n_r f^2)$
  - possibly, a loss computation: $O(n_r f)$
- the recommendations: $O(m(\log(m) + f))$

The resulting complexity is

$$n_{\text{iter}} \cdot O((n+m)f^3 + n_r f^2 + m\log(m)) = O(n_{\text{iter}}(n+m)f^3 + n_r f^2 + m\log(m))$$

which is a good outcome.
The factor dimension is much lower than the number of movies and users, and iterations are often set to less than 10 due to the high speed of convergence, so the model scales essentially linearly in the users and log-linearly in the movies.

3.2.3 Complexity of ALS with Early Stopping

We refer to algorithm 5. We have:

- initialization: $O(((n + m)f)$
- a for cycle of max length n_{iter} with
  - an update of user factors: $O(nf^3 + n_r f^2)$
  - an update of movie factor: $O(mf^3 + n_r f^2)$
  - one (validation) or two (validation and training) loss computations: $O(n_r f)$
- the recommendations: $O(m(\log(m) + f)))$

The total complexity is again

$$O(n_{\text{iter}}(n + m)f^3 + n_r f^2 + m\log(m))$$

Potentially lower if the early stopping criterion intervenes to stop the iterations considerably earlier than n_{iter} (which is unlikely given the small amount of iterations usually performed).

Chapter 4

Examples: Toy and Real Data

We have discussed that the latent factors built by ALS essentially represent more or less interpretable and complicated movie features and the way each user may or may not appreciate them.

To clarify this heuristic, we construct a very small example based on real-life data we sampled (from a friend group) that fit our purpose. Ratings are in the same format as the MovieLens data, which is the one previously discussed. We will consider 6 movies and 6 users, who rated all or some of them. The goal will be to illustrate how the matrix factorization works and to get an intuition behind what factors are meant to represent and what it means for ASL to compare them directly using the inner product.

Then we will move on to a proper testing sub-sample of the real data, to test the perfomance of the algorithm.

Finally, after introducing the MovieLens 100K dataset, we will use Apache Spark implementation for the full dataset.

4.1 An Example to Visualize

This is a quick section to get an intuitive understanding of ALS. As mentioned, we rely on data sampled from 6 friends on 6 popular movies.
First, we showcase the ”reconstructing ability” of ALS as a matrix factorization algorithm. Recall that users are rows and movies are columns.

```
# paths
ratings_path = "data/ratings_small.csv"
movie_path = "data/movies_small.csv"

# Initialize UserItemMatrix instance
uim = UserItemMatrix(ratings_path, movie_path, sparse=True)
```

```
# Initialize ALS_WR instance
als = ALS_WR(uim, factor_dim=2)

# Train the model
als.fit(n_iter=10, reg_param=0.05, regularization="weighted")

# show the reconstructed matrix
reconstructed_matrix = als.user_factors @ als.movie_factors
# set values to zero if the original matrix has zero values
reconstructed_matrix[uim.get_user_item_matrix().values == 0]
                               = 0
# round the reconstructed to the closest integer .5 value
reconstructed_matrix = np.round(reconstructed_matrix * 2) / 2
# clip values in [0,5]
reconstructed_matrix = np.clip(reconstructed_matrix, 0, 5)

plt.figure(figsize=(12, 6))
plt.subplot(1, 2, 1)
sns.heatmap(uim.get_user_item_matrix().values, cmap='coolwarm
                               ', cbar=True, annot = True,
                               fmt=".1f")
plt.title('Original Matrix')
plt.subplot(1, 2, 2)
sns.heatmap(reconstructed_matrix, cmap='coolwarm', cbar=True,
                                annot = True, fmt=".1f")
plt.title('Reconstructed Matrix')

plt.show()
plt.show()
```

The output shows a good accuracy despite the scarcity of data, the few iterations and the latent factor dimension of just 2.

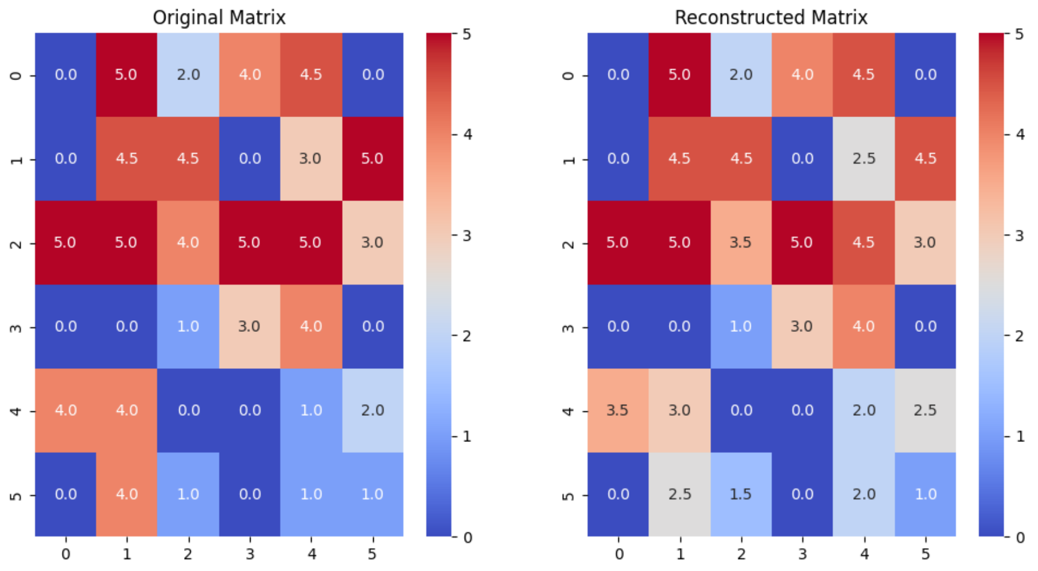

Figure 4.1: Reconstructing the Training Ratings.

We can easily predict missing ratings:

```
# heatmap (wiht values) of the estimated rating matrix
reconstructed_matrix = np.round(als.user_factors @ als.
                                movie_factors, 2).clip(0, 5)
plt.figure(figsize=(10, 6))
sns.heatmap(reconstructed_matrix, cmap='coolwarm', cbar=True,
                                 annot = True, fmt=".1f")
plt.title('Estimated Rating Matrix')
plt.show()
```

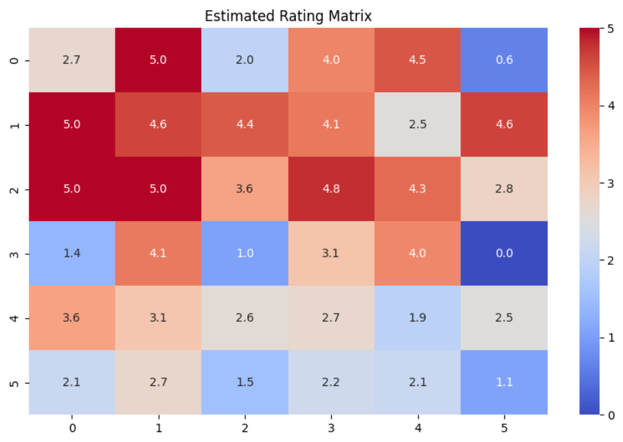

Figure 4.2: Full Reconstructed Ratings

Since we only care about the order of the predicted ratings, we don't adjust the matrix to fit the ordinal categorical rating values.
An example of recommendation:

```
# get recommendation for user "fede" for unseen movies
fede_rec_idx = als.recommend(0, n=2, include_seen=False,
                               print=False)
fede_rec = movies[movies['movieId'].isin(fede_rec_idx)]["
                               title"]
print(f"Recommendation for user 'Fede': ")
for rec in fede_rec:
    print(f"- {rec}")

# get recommendation for user "Lucio" for unseen movies
lucio_rec = als.recommend(3, n=3, include_seen=False, print=
                               False)
lucio_rec = movies[movies['movieId'].isin(lucio_rec)]["title"
                               ]
print(f"Recommendation for user 'Lucio': ")
for rec in lucio_rec:
    print(f"- {rec}")
```

```
Recommendation for user 'Fede':
- Forrest Gump
- LaLaLand
Recommendation for user 'Lucio':
- Forrest Gump
- Kill Bill
- Avengers
```

Clearly here we have to limit the recommendations to less than 10 titles.

We can also attempt at giving an interpretation to factors, by plotting the bi-dimensional representations of users and movies:

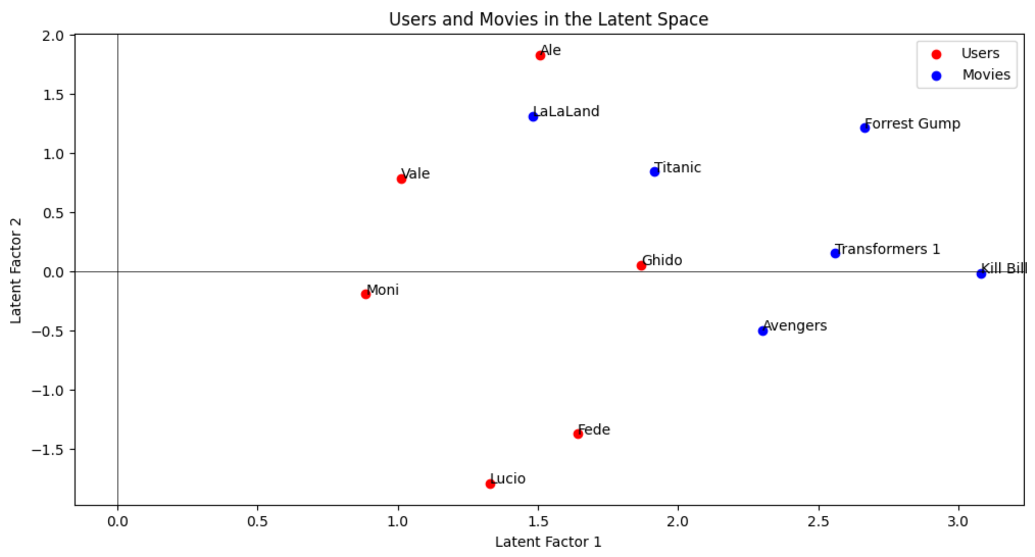

Figure 4.3: Users vs Movies in the factor space

Recall that a predicted rating is given by the inner product between the user-point and the movie-point. For example, factor 1 could refer to the level of action in a movie or some more abstract concept related to that (which would explain the two "clusters"), whereas factor 2 could be related to how emotional the movie is. Users' coordinates, tell us how they relate to this factors. For example, Ale values highly the emotional component, way more than the action. To Lucio instead, the emotional component of a movie lowers the score (since (almost) all factor 2 coordinates are positive for movies, but he has a negative coordinate instead).
Of course, this interpretation is somewhat stretched, but the point of using matrix factorization instead of content based recommenders is to let factors handle the identification of aspects of movies and users that are relevant to compute ratings, especially when not intuitive.

4.2 MovieLens 100k

MovieLens 100K is a popular dataset for testing algorithms involving recommendations. The data is collected by the GroupLens research group of university of Minnesota and is based on actual ratings they registered from users of their own recommender system, MovieLens.

As the name suggests, the dataset is comprised of around 100'000 reviews of more than 600 users and on almost 10'000 movies. Users are uniquely identified by a `userId`, and likewise movies with a `movieId`. The dataset provides additional information (such as links to the movies on their website, genres, tags, ...), but we will only need ratings, ID's and movie titles. The notebooks contains a quick exploratory data analysis, which resulted in a few considerations:

- users have all produced more than 20 reviews, making them all relevant;
- many movies have a single review and 50% of them only have 3 reviews. Even if we chose to remove them, we are facing a sparsity problem. We will retain all movies, as we may expect this to be a realistic case and we would like a bit of serendipity in the algorithm. However, we could make a case for removing them in case the performance of the algorithm would turn out to be unsatisfying;
- there are no missing values of duplicates to handle, so no particular data cleaning is needed.

4.3 A Proper Example of a Small Dataset

In the section of the notebook with this same name, we extract a subset of ratings with a heuristic approach and ensuring the needed constraints are satisfied (i.e.: no user or movie is left with no ratings). The subset if further split into train and validation data. Here we show how the algorithm can be used to perform a simple grid search for the hyperparameters to tune once all the needed data is stored in proper `.csv` file, and we display the early stopping approach.

```
    # Initialize UserItemMatrix instance
uim = UserItemMatrix(training_path, movie_path, sparse=True)

# perform a grid_search, pick the best hyperparameters, store
                                the validation and training
                               loss
results = []
for reg_param in [0.1, 0.2]:
    for n_factors in [5, 10]:
        als = ALS_WR(uim, factor_dim=n_factors)
        als.fit(n_iter=20, reg_param=reg_param,
                                    regularization="
                                    weighted",
                                    validation_path=
                                    validation_path,
                                    patience = 5)
        training_loss = als.get_loss()
        validation_loss = als.get_validation_loss()
        best_loss = als.get_best_loss()
        results.append([reg_param, n_factors, training_loss,
                                             validation_loss,
                                             best_loss])

# select best model as the one with the smallest validation
                               loss
results = pd.DataFrame(results, columns=["reg_param", "
                               n_factors", "training_loss", "
                               validation_loss", "best_loss"]
                               )
best_model = results.loc[results['best_loss'].idxmin()]

print(f"Best model hyperparameters: reg_param={best_model['
                               reg_param']}, n_factors={
                               best_model['n_factors']}")
print(f"Best model loss: {best_model['best_loss']}")

# plot the training and validation loss for the best model
plt.figure(figsize=(10, 6))
plt.plot(best_model['training_loss'], label='Training Loss')
plt.plot(best_model['validation_loss'], label='Validation
                               Loss')
plt.xlabel('Iteration')
plt.ylabel('RMSE Loss')
plt.title('ALS_WR Model Loss')
plt.legend()
plt.show()
```

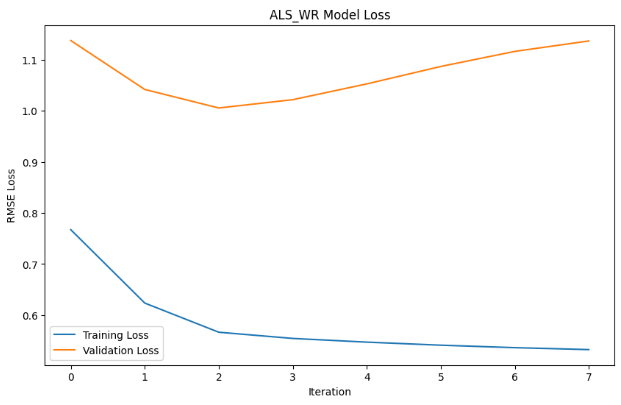

Figure 4.4: Training vs Validation Loss

4.4 Real Data

To test our algorithm on real data we rely on the implementation of ALS by Apache Spark, a multi-language engine for executing data engineering, data science, and machine learning on single-node machines or clusters. The code is highly optimized to perform parallel computations. Here is the code for a cross validation grid search:

```
# Create a Spark session
spark = SparkSession.builder.appName('recommender').
                              getOrCreate()

# Load the data
data = spark.read.csv('data/ratings.csv', inferSchema=True,
                              header=True)

# Split the data
(training, test) = data.randomSplit([0.8, 0.2])

# Build the recommendation model using ALS on the training
                              data
als = ALS(maxIter=10, userCol='userId', itemCol='movieId',
                              ratingCol='rating',
                              coldStartStrategy='drop')

# perform grid search
from pyspark.ml.tuning import ParamGridBuilder,
                              CrossValidator

# We use a ParamGridBuilder to construct a grid of parameters
                               to search over.
# TrainValidationSplit will try all combinations of values
                              and determine best model using
# the evaluator.
param_grid = ParamGridBuilder() \
    .addGrid(als.rank, [100, 150, 200]) \
    .addGrid(als.regParam, [.05, .1]) \
    .build()

# Define evaluator as RMSE
evaluator = RegressionEvaluator(metricName='rmse', labelCol='
                              rating', predictionCol='
                              prediction')

# Build CrossValidator
cv = CrossValidator(estimator=als, estimatorParamMaps=
                              param_grid, evaluator=
                              evaluator, numFolds=5,
                                    parallelism=8)

# Fit ALS model to training data
model = cv.fit(training)

# Extract best model from the tuning exercise using
                                    ParamGridBuilder
best_model = model.bestModel

# Generate predictions and evaluate using RMSE
predictions = best_model.transform(test)
rmse = evaluator.evaluate(predictions)

# Print evaluation metrics and model parameters
print("RMSE = " + str(rmse))
print("**Best Model**")
print(" Rank:", best_model.rank)
print(" MaxIter:", best_model._java_obj.parent().getMaxIter()
                                    )
print(" RegParam:", best_model._java_obj.parent().getRegParam
                                    ())
```

```
RMSE = 0.8691032268759084
**Best Model**
Rank:  100
MaxIter:  10
RegParam:  0.1
```

The recommendations then are simply given by:

```
# make top_n recommendations for a user
n = 10
user_id = 1 # choose a user
# get the user's movie ratings
user_ratings = data.filter(data['userId'] == user_id)
# get the movies the user has rated
rated_movies = user_ratings.select('movieId')
# get the movies the user has not rated
unrated_movies = data.select('movieId').subtract(rated_movies
                                    )
# rename the column to userId
unrated_movies = unrated_movies.withColumn('userId', lit(int(
                                    user_id)))
# generate predictions
predictions = best_model.transform(unrated_movies)
# get the top n recommendations
top_n = predictions.orderBy('prediction', ascending=False).
                                    limit(n)
# get the movie titles
df_movies = spark.read.csv('data/movies.csv', inferSchema=
                                                    True ,  header=True)
# get the movie titles
top_n_titles = top_n.join(df_movies, 'movieId', 'inner').
                                                    select('title', 'movieId')
# show the recommendations
top_n_titles.show()
```

and the recommended movie titles are displayed. In the notebook we also provide a simple visual comparison of the performance with different parameters by drawing the test loss in a simpler training-test split procedure, with no CV. The outcomes differ from the previous ones, but we should consider the best model identified through cross validation as the preferred one, since CV is generally more reliable in predicting the generalization ability of a model.

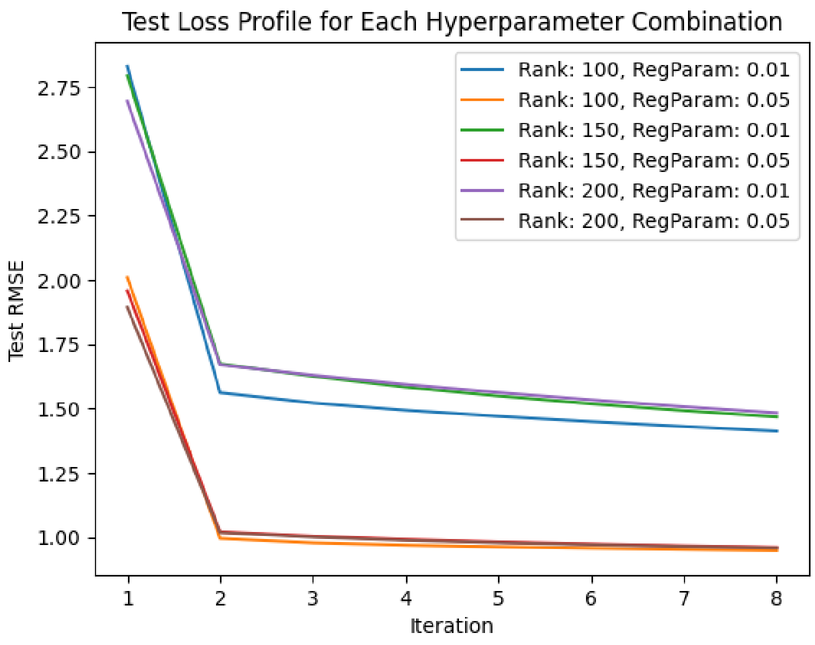

Figure 4.5: Test Losses for different parameters

Chapter 5

Conclusions

ALS-WR is a powerful matrix-factorization technique for recommender systems. It is a fairly simple algorithm, but accurate and very flexible, as it can be used both for explicit and implicit feedback.
The Apache Spark implementation showcases one of the most interesting aspects of ALS that made it one of the most popular factorization algorithms for recommendations: its scalability. The algorithm allows for data partitioning and, given its alternating nature, makes it easy to parallelize computations at each alternating update. Another interesting aspect is the relative simplicity with which new data can be incorporated without retraining the whole model. If a few users are added (with some ratings), it can be enough to perform a single user factor update without going through the whole process to ensure a good performance.

Obviously, no algorithm is perfect. Historically, ALS performed slightly worse that some more tailored and complicated models on the Netflix Prize data. Usually hybrid models incorporating both model-based approaches and neighbourhood based approaches outperform, in terms of evaluation metrics, the predictive power of ALS and most other "single-approach" methods. However, these models are more computationally intensive, less flexible when it comes to new data and not as easily parallelizable, in general.

When discussing the flaws of ALS, and of collaborative filtering in general, we need to mention the cold-start problem. The algorithm is, by design, incapable of making predictions for data for which no rating is available. To avoid this limitation, we could consider a hybrid model incorporating content-based recommendations for new users (e.g.: recommending the most popular/trending movies to new users) and a different recommender for newly available movies that could highlight them. Realistically, this can be done in different section of a streaming service/recommender system. In other words, ALS is one of the best solutions for the "top-N recommendations" problem only for users that have a history of ratings.

We conclude by highlighting that the proposed evaluation metric is not particularly "expressive" in understanding the model performance. Other metrics have been proposed, but there doesn't seem to be general consensus on any. The best "test" for our system, and for recommenders in general, would be to receive human feedback from its use.

An interesting extension to this work, if data on user interactions with movies beside ratings becomes public, would be to test how two ALS models can be integrated to make use of implicit and explicit feedback to make recommendations.

Sources

[1] https://commons.wikimedia.org/w/index.php?curid=23323794
By Moshanin Own work, CC BY-SA 3.0.

[2] Yifan Hu, Yehuda Koren, and Chris Volinsky. Collaborative filtering for implicit feedback datasets. In *2008 Eighth IEEE International Conference on Data Mining*, pages 263–272, 2008.

[3] Yunhong Zhou, Dennis Wilkinson, Robert Schreiber, and Rong Pan. Large-scale parallel collaborative filtering for the netflix prize. pages 337–348, 06 2008.