Project - High Dimensional Data Analysis

Recommendation systems via approximate matrix factorization

Semenova Diana, Luzinsan Anastasia, Soluyanova Zlata

November 27, 2024

Abstract

This report will explore several matrix factorization techniques for recommendation, including variations such as Non-negative matrix factorization (NMF), Alternating Least Squares (ALS) et cetera. The report also presents the tuning of haperparameters to improve the results of the algorithms.

1 Introduction

Recommendation systems aim to predict user preferences for products by leveraging collective user behavior, a technique known as collaborative filtering. This field holds significant economic potential, particularly for e-commerce and online platforms like Netflix, as effective recommendations directly increase sales conversion rates. The substantial financial incentives inherent in improved recommendation accuracy are exemplified by the 2007 Netflix Prize competition, which offered a $1 million reward for a 10% improvement in prediction accuracy. This competition highlights the considerable economic value associated with accurate and personalized recommendations and the ongoing drive for innovation in this area. The success of a recommendation system depends heavily on the quality of its prediction algorithm, with various techniques, including matrix factorization, nearest neighbor methods, and reinforcement learning, continuously being developed and refined.

Matrix factorization is a highly effective model frequently employed in practical recommendation systems. This approach decomposes the user-item interaction matrix into lower-dimensional latent factor matrices, capturing underlying user preferences and item characteristics. The aim of this project is at first to develop and analyze one or more methods for solving the problem. We will then try to improve the model to obtain better predictions.

2 The problem statement

We seek to decompose a matrix X into two low-rank matrices $U \in R^{m \times r}$ and $V \in R^{n \times r}$, such that:

$$X \approx UV^T$$

The optimization problem can be written as:

$$\min_{U,V} \|W \odot (X - UV^T)\|_F^2 + \lambda_{\text{reg}} \left(\|U\|_F^2 + \|V\|_F^2\right)$$

where:

- W is a binary mask matrix indicating observed entries ($W_{ij} = 1$ if X_{ij} is observed, 0 otherwise).

- $\odot$ represents element-wise multiplication (Hadamard product).

- λ_{reg} is a regularization parameter to prevent overfitting.

- $\|\cdot\|_F$ denotes the Frobenious norm.

3 Description of the models used

3.1 Non-negative matrix factorization (NMF)

The **Multiplicative Update (MU)** algorithm iteratively updates W and H by minimizing the reconstruction error: $\min_{W,H} ||X - WH||_F^2$ subject to $W \geq 0$ and $H \geq 0$. Here, $||\cdot||_F$ denotes the Frobenious norm.

Update Equations

The MU algorithm applies the following element-wise update rules for W and H:

1. Update for H: $[H \leftarrow H \circ \frac{W^T X}{W^T W H}]$

2. Update for W: $[W \leftarrow W \circ \frac{X H^T}{W H H^T}]$

where $(\circ)$ denotes element-wise multiplication, and division is also element-wise.

3.2 Block coordinate descent method with Alternating Least Squares (ALS)

3.2.1 Problem for Rank $r = 1$

We are given a sparse matrix $X \in R^{m \times n}$, where only some elements $(i, j) \in \Omega$ are observed. The goal is to approximate X as the outer product of two vectors:

$$X \approx uv^\top, \quad u \in R^m, v \in R^n$$

Objective Function

The approximation error is defined using the observed entries. Let $W \in \{0, 1\}^{m \times n}$ be a binary mask, where:

$$W_{ij} = \begin{cases} 1, & \text{if } (i, j) \in \Omega, \\ 0, & \text{otherwise.} \end{cases}$$

The optimization problem is formulated as:

$$\min_{u,v} \sum_{(i,j)\in\Omega} (X_{ij} - u_i v_j)^2$$

or equivalently, using W:

$$\min_{u,v} \|W \odot (X - uv^\top)\|_F^2$$

where $\odot$ denotes the elementwise product and $\|\cdot\|_F$ is the Frobenius norm.

Alternating Updates

1. **Fix u, update v:** Solve for v while holding u fixed:

$$v_j = \arg\min_{v_j} \sum_{i\,:\,W_{ij}=1} (X_{ij} - u_i v_j)^2$$

This is a least squares problem with the solution:

$$v_j = \frac{\sum_{i\,:\,W_{ij}=1} u_i X_{ij}}{\sum_{i\,:\,W_{ij}=1} u_i^2}$$

2. **Fix v, update u:** Similarly, update each u_i by solving:

$$u_i = \arg\min_{u_i} \sum_{j\,:\,W_{ij}=1} (X_{ij} - u_i v_j)^2$$

The closed-form solution is:

$$u_i = \frac{\sum_{j\,:\,W_{ij}=1} v_j X_{ij}}{\sum_{j\,:\,W_{ij}=1} v_j^2}$$

3.2.2 Generalization to Rank $r > 1$

When $r > 1$, the matrix X is approximated as the product of two low-rank matrices:

$$X \approx UV^\top, \quad U \in R^{m\times r}, V \in R^{n\times r}$$

Objective Function

The generalized objective function becomes:

$$\min_{U,V} \|W \odot (X - UV^\top)\|_F^2$$

We also add a regularization term to avoid overfitting:

$$\min_{U,V} \|W \odot (X - UV^\top)\|_F^2 + \lambda_{\text{reg}} \left(\|U\|_F^2 + \|V\|_F^2\right)$$

where λ_{reg} is a hyperparameter controlling the regularization strength.

Alternating Updates

As with the rank-1 case, we use BCD to alternately update U and V.

1. **Fix U, update V:** For each column j of V, solve:

$$V[j,:] = \arg\min_{v_j} \sum_{i\,:\,W_{ij}=1} \left(X_{ij} - U[i,:]v_j^\top\right)^2 + \lambda_{\text{reg}} \|v_j\|_2^2$$

Let known_idx $= \{i : W_{ij} = 1\}$. Define:

$$U_{\text{known}} \in R^{|\text{known\_idx}|\times r}, \quad X_{\text{known}} \in R^{|\text{known\_idx}|}$$

The update is obtained by solving the normal equations:

$$A = U_{\text{known}}^{\top} U_{\text{known}} + \lambda_{\text{reg}} I_r, \quad b = U_{\text{known}}^{\top} X_{\text{known}}$$

$$V[j,:] = A^{-1} b$$

2. **Fix V, update U:** Similarly, for each row i of U, solve:

$$U[i,:] = \arg\min_{u_i} \sum_{j\,:\,W_{ij}=1} \left(X_{ij} - u_i V[j,:]^{\top}\right)^2 + \lambda_{\text{reg}} \|u_i\|_2^2$$

Let known_idx $= \{j : W_{ij} = 1\}$. Define:

$$V_{\text{known}} \in R^{|\text{known\_idx}| \times r}, \quad X_{\text{known}} \in R^{|\text{known\_idx}|}$$

Solve:

$$A = V_{\text{known}}^{\top} V_{\text{known}} + \lambda_{\text{reg}} I_r, \quad b = V_{\text{known}}^{\top} X_{\text{known}}$$

$$U[i,:] = A^{-1} b$$

3.3 Block Coordinate Gradient Descent Method

Let the objective function to be minimized be represented as:

$$L(U,V) = \frac{1}{2} \|W \odot \left(X - UV^T\right)\|_F^2 + \lambda \left(\|U\|_F^2 + \|V\|_F^2\right)$$

where:

- $X \in R^{m \times n}$ is the observed matrix,

- $U \in R^{m \times k}$ and $V \in R^{n \times k}$ are the factor matrices to be optimized,

- $\| \cdot \|_F$ denotes the Frobenius norm,

- W is a binary mask matrix indicating observed entries ($W_{ij} = 1$ if X_{ij} is observed, 0 otherwise).

- $\odot$ represents element-wise multiplication (Hadamard product).

- λ is a regularization parameter.

The goal is to find the matrices U and V that minimize the loss function $L(U,V)$.

The BCGD method optimizes the factor matrices U and V by alternately updating each block while keeping the other fixed.

1. **Update** U: The gradient of the loss function with respect to U is given by:

$$\nabla_U L(U,V) = -2(X - UV^T)V + 2\lambda U$$

The update for U is performed by applying gradient descent:

$$U \leftarrow U - \alpha_U \nabla_U L(U,V)$$

where α_U is the learning rate (step size).

2. **Update** V: Similarly, the gradient of the loss function with respect to V is:

$$\nabla_V L(U, V) = -2(X - UV^T)^T U + 2\lambda V$$

The update for V is performed by:

$$V \leftarrow V - \alpha_V \nabla_V L(U, V)$$

where α_V is the learning rate for V.

3.3.1 Adaptive Step Size Update

The step sizes α_U and α_V are dynamically adjusted based on the observed decrease in the loss. If the loss does not decrease after a certain number of steps, the learning rate is decreased by a factor of γ (the decay factor).

3.3.2 Armijo Rule Step Size Update

The **Armijo rule** adaptively chooses a step size α that ensures sufficient decrease in the objective.

1D Armijo Rule

In a 1D optimization problem, consider minimizing a function $f(x)$ with gradient descent:

$$x^{(k+1)} = x^{(k)} - \alpha \nabla f(x^{(k)}),$$

where $\alpha > 0$ is the step size. The Armijo condition states:

$$f(x^{(k+1)}) \leq f(x^{(k)}) - \gamma\alpha |\nabla f(x^{(k)})|^2,$$

for some $\gamma \in (0, 1)$. This ensures the step size α decreases the objective sufficiently relative to the magnitude of the gradient.

From 1D to Multidimensional Case

In a multidimensional setting, we extend $x \in R^n$ to a vector and generalize the gradient descent update:

$$\mathbf{x}^{(k+1)} = \mathbf{x}^{(k)} - \alpha \nabla f(\mathbf{x}^{(k)}),$$

with the Armijo condition:

$$f(\mathbf{x}^{(k+1)}) \leq f(\mathbf{x}^{(k)}) - \gamma\alpha \|\nabla f(\mathbf{x}^{(k)})\|_2^2,$$

where $\|\nabla f(\mathbf{x})\|_2^2$ is the squared Euclidean norm of the gradient.

Armijo Rule for Block Updates

When optimizing $L(\mathbf{U}, \mathbf{V})$, the update for $\mathbf{U}$ becomes:

$$\mathbf{U}^{(k+1)} = \mathbf{U}^{(k)} - \alpha \nabla_{\mathbf{U}} L(\mathbf{U}^{(k)}, \mathbf{V}^{(k)}),$$

where $\nabla_{\mathbf{U}} L$ is the gradient of the loss with respect to $\mathbf{U}$. Similarly, for $\mathbf{V}$:

$$\mathbf{V}^{(k+1)} = \mathbf{V}^{(k)} - \alpha \nabla_{\mathbf{V}} L(\mathbf{U}^{(k)}, \mathbf{V}^{(k)}).$$

The Armijo condition for the matrix case is:

$$L(\mathbf{U}^{(k+1)}, \mathbf{V}) \leq L(\mathbf{U}^{(k)}, \mathbf{V}) - \gamma\alpha\|\nabla_{\mathbf{U}} L(\mathbf{U}^{(k)}, \mathbf{V})\|_F^2,$$

and similarly for $\mathbf{V}$:

$$L(\mathbf{U}, \mathbf{V}^{(k+1)}) \leq L(\mathbf{U}, \mathbf{V}^{(k)}) - \gamma\alpha\|\nabla_{\mathbf{V}} L(\mathbf{U}, \mathbf{V}^{(k)})\|_F^2.$$

4 Description of the algorithms

4.1 Non-negative matrix factorization (NMF)

This algorithm iteratively refines two non-negative matrices, $\mathbf{U}$ and $\mathbf{V}$, to approximate the input matrix $\mathbf{X}$ ($\mathbf{X} \approx \mathbf{U}\mathbf{V}^T$). Multiplicative update rules ensure non-negativity. Convergence is achieved when the change in loss falls below a tolerance or the maximum iteration count is reached. A small constant is added to denominators to prevent division by zero.

Algorithm 1 Non-negative Matrix Factorization (NMF) with Multiplicative Updates

procedure NMF(max_iter)
  for $idx = 0$ **to** $max\_iter - 1$ **do**
    $\mathbf{V} \leftarrow \mathbf{V} \odot \frac{\mathbf{U}^T\mathbf{X}}{\mathbf{U}^T\mathbf{U}\mathbf{V}+10^{-20}}$ ▷ Update V
    $\mathbf{U} \leftarrow \mathbf{U} \odot \frac{\mathbf{X}\mathbf{V}^T}{\mathbf{U}\mathbf{V}\mathbf{V}^T+10^{-20}}$ ▷ Update U
    if converged **then**
      break
    end if
  end for
  return PREDICT($\mathbf{U}$, $\mathbf{V}$)
end procedure

4.2 Block coordinate descent method

This algorithm factorizes a matrix $\mathbf{X}$ into $\mathbf{U}$ and $\mathbf{V}$ by iteratively updating $\mathbf{U}$ and $\mathbf{V}$. It solves a least squares problem for each matrix ($\mathbf{U}$, $\mathbf{V}$) while keeping the other fixed, using only the known entries specified by the weight matrix $\mathbf{W}$. L2 regularization prevents overfitting. The algorithm stops when the change in training loss is below a tolerance or the maximum iteration count is reached.

Algorithm 2 Block Coordinate Descent (BCD) with Alternating Least Squares (ALS) for Matrix Factorization

procedure BCD(max_iter, tol, lambda_reg)
  for $idx = 0$ **to** $max\_iter - 1$ **do**
    for $j = 0$ **to** self.n - 1 **do**
      ▷ Update V column j
      $known\_idx \leftarrow$ indices of nonzero elements in $self.W[:, j]$
      $\mathbf{U}_{known} \leftarrow self.U[known\_idx, :]$
      $\mathbf{x}_{known} \leftarrow self.X[known\_idx, j]$
      $\mathbf{A} \leftarrow \mathbf{U}_{known}^T \mathbf{U}_{known} + \lambda\_reg \mathbf{I}_r$
      $\mathbf{b} \leftarrow \mathbf{U}_{known}^T \mathbf{x}_{known}$
      $self.V[:, j] \leftarrow$ solve$(\mathbf{A}, \mathbf{b})$
    end for
    for $i = 0$ **to** self.m - 1 **do**
      ▷ Update U row i
      $known\_idx \leftarrow$ indices of nonzero elements in $self.W[i, :]$
      $\mathbf{V}_{known} \leftarrow self.V[:, known\_idx]$
      $\mathbf{x}_{known} \leftarrow self.X[i, known\_idx]$
      $\mathbf{A} \leftarrow \mathbf{V}_{known} \mathbf{V}_{known}^T + \lambda\_reg \mathbf{I}_r$
      $\mathbf{b} \leftarrow \mathbf{V}_{known} \mathbf{x}_{known}^T$
      $self.U[i, :] \leftarrow$ solve$(\mathbf{A}, \mathbf{b})$
    end for
    if `converged` **then**
      break
    end if
  end for
  return PREDICT(U, V)
end procedure

4.3 Block Coordinate Gradient Descent Method

This algorithm also factorizes $\mathbf{X}$ into $\mathbf{U}$ and $\mathbf{V}$, using gradient descent with a backtracking line search to update each matrix block ($\mathbf{U}$ and $\mathbf{V}$). It incorporates momentum and early stopping for improved convergence. The step size is adjusted dynamically. L2 regularization prevents overfitting. The learning rate can be specified or automatically determined.

Algorithm 3 Block Coordinate Gradient Descent (BCGD)

procedure BCGD(lr, sigma, gamma, max_iter, lambda_reg)
    $lr\_U \leftarrow lr$
    $lr\_V \leftarrow lr$
    function UPDATE_BLOCK(B, $fixed\_B$, WR_f, $grad\_f$, lr)
        $WR \leftarrow WR\_f(B, fixed\_B)$
        $grad\_B \leftarrow grad\_f(WR, B, fixed\_B)$
        if lr is None or $lr = 0$ **then**
            $\alpha \leftarrow \frac{2||B||_2}{||grad\_B||_2}$
        else
            $\alpha \leftarrow lr$
        end if
        $error\_0 \leftarrow \text{RMSE}(WR)$
        $updated\_B \leftarrow B - \alpha * grad\_B$
        $error\_1 \leftarrow \text{RMSE}(WR\_f(updated\_B, fixed\_B))$
        for $k = 0$ **to** $patience - 1$ **do**
            if $error\_0 \leq error\_1$ **then**
                break
            end if
            $updated\_B \leftarrow B - \alpha * grad\_B$
            $error\_1 \leftarrow \text{RMSE}(WR\_f(updated\_B, fixed\_B))$
            $\alpha \leftarrow \alpha/\gamma$
        end for
        return $updated\_B$
    end function

    for $idx = 0$ **to** $max\_iter - 1$ **do**
                                                                ▷ Update U
        $self.U \leftarrow \text{UPDATE\_BLOCK}(self.U, self.V, W * (X - UV), -2WRV^T + \lambda\_reg * (U^2), lr\_U)$
                                                                ▷ Update V
        $self.V \leftarrow \text{UPDATE\_BLOCK}(self.V, self.U, W * (X - UV), -2WR^TU + \lambda\_reg(V^T)^2, lr\_V)$
    end for
    if converged then
        break
    end if
    return PREDICT(U, V)
end procedure

4.4 Block Coordinate Gradient Descent Method with Armijo Rule

Algorithm 4 Block Coordinate Gradient Descent with Armijo Rule

procedure BCGD(c, beta, gamma, max_iter, log_step, tol, patience, lambda_reg)
  function ARMIJO_RULE(f, x, p)
    $\alpha \leftarrow beta^c$
    for $k = 0$ **to** $patience - 1$ **do**
      if $f(x - \alpha p) - f(x) > -gamma * \alpha * ||p^T p||_F^2$ **then**
        $c \leftarrow c + 1$
        $\alpha \leftarrow beta^c$
      else
        break
      end if
    end for
    return α
  end function

  function UPDATE_BLOCK(B, $fixed\_B$, WR_f, $grad\_f$)
    $WR \leftarrow WR\_f(B, fixed\_B)$
    $grad\_B \leftarrow grad\_f(WR, B, fixed\_B)$
    $loss\_f(b) \leftarrow mse(WR\_f(b, fixed\_B))$
    $\alpha \leftarrow$ armijo_rule(idx, $loss\_f$, B, $grad_B$)
    $updated\_B \leftarrow B - \alpha * grad_B$
    $error_1 \leftarrow rmse(WR_f(updated_B, fixed_B))$
    return $updated_B$
  end function

  for $idx = 0$ **to** $max\_iter - 1$ **do**
    ▷ Update U
    $self.U \leftarrow$ UPDATE_BLOCK($self.U, self.V, W * (X - UV), -2WRV^T + \lambda\_reg * (U^2), lr\_U$)
    ▷ Update V
    $self.V \leftarrow$ UPDATE_BLOCK($self.V, self.U, W * (X - UV), -2WR^T U + \lambda\_reg(V^T)^2, lr\_V$)
  end forreturn predict(U, V)
end procedure

5 Results and Discussion

5.1 Non-negative Matrix Factorization (NMF)

Synthetic Dataset

A small synthetic dataset with missing values was used to test the NMF implementation. The algorithm was initialized with random values for **U** and **V** and run for 1000 iterations with a tolerance of 10^{-10}. The training loss decreased steadily from 1.03874 to 0.41904, indicating successful convergence. The relatively low final training loss suggests a good fit to the observed data.

Kaggle Dataset

The NMF algorithm was applied to a larger Kaggle dataset ('inputX.mat'). The dataset was split into training and validation sets (80/20 split). The algorithm was run for 100 iterations with a tolerance of 10^{-4} and using 15 latency factors (rank). The training and validation losses are shown in the following table.

The training loss shows a decreasing trend, although convergence within the tolerance is not fully

Iteration	Training Loss	Validation RMSE
1	3.21362	3.22326
11	3.13064	3.1578
21	2.97416	3.02788
31	2.93171	2.99097
41	2.91461	2.97552
51	2.9044	2.96591
61	2.89826	2.96004
71	2.89451	2.95634
81	2.89198	2.95377
91	2.89013	2.95193
100	2.88887	2.95063

Table 1: Training and Validation Loss for Kaggle Dataset

achieved within 100 iterations. The validation RMSE also decreases but shows a smaller rate of improvement compared to the training loss, this suggests potential overfitting to the training data. After 9000 iterations of training an RMSE of approximately 1.1098 was obtained on the test set. The result on the validation set was 1.08976. The optimal number of factors appeared to be around 15, indicated by the balance between validation and test set errors.

Conclusions

Unfortunately, the NMF algorithm demonstrated effective performance on Kaggle dataset only after 9000 iterations, that is very computational expensive. Further tuning of latent factors may lead to improvement in prediction accuracy, and more sophisticated regularization techniques might mitigate potential overfitting issues observed in the larger dataset.

5.2 Block Coordinate Descent (BCD) with Alternating Least Squares (ALS)

Synthetic Dataset

A small synthetic dataset (with missing values) was used for initial testing. The algorithm was initialized with random U and V matrices and run for 10 iterations with a regularization parameter (λ_{reg}) of 0.01 and a tolerance of 10^{-4}. The training loss decreased from 0.20609 to 0.03176, showing a clear trend toward convergence.

Kaggle Dataset

Hyperparameter tuning was performed on a larger Kaggle dataset ('inputX.mat') using Optuna. The objective function was the validation loss. The hyperparameters tuned included the rank, the initialization method, and the regularization parameter. The search space for rank ranged from 2 to 20, initializations cases was included 'random', 'eye', 'zeros', and 'SVD' initializations, and regularization parameter was sampled logarithmically from 10^{-10} to 1. A 80/20 train/validation split was used, with the Optuna study configured to run for 1000 trials.

Optuna is a hyperparameter optimization framework designed to efficiently explore the parameter space of machine learning models. This targeted exploration reduces the computational cost associated with exhaustive hyperparameter tuning and improves the likelihood of identifying optimal configurations. The framework operates by iteratively conducting trials, each representing a training run with a specific hyperparameter configuration suggested by the chosen optimization algorithm. After each trial, the model's performance, measured by a user-defined objective function, is evaluated.

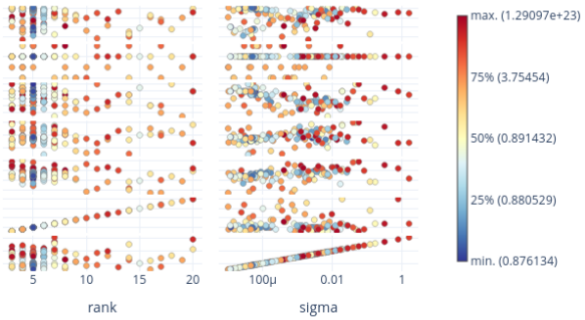


Figure 1: Results of Optuna for the rank

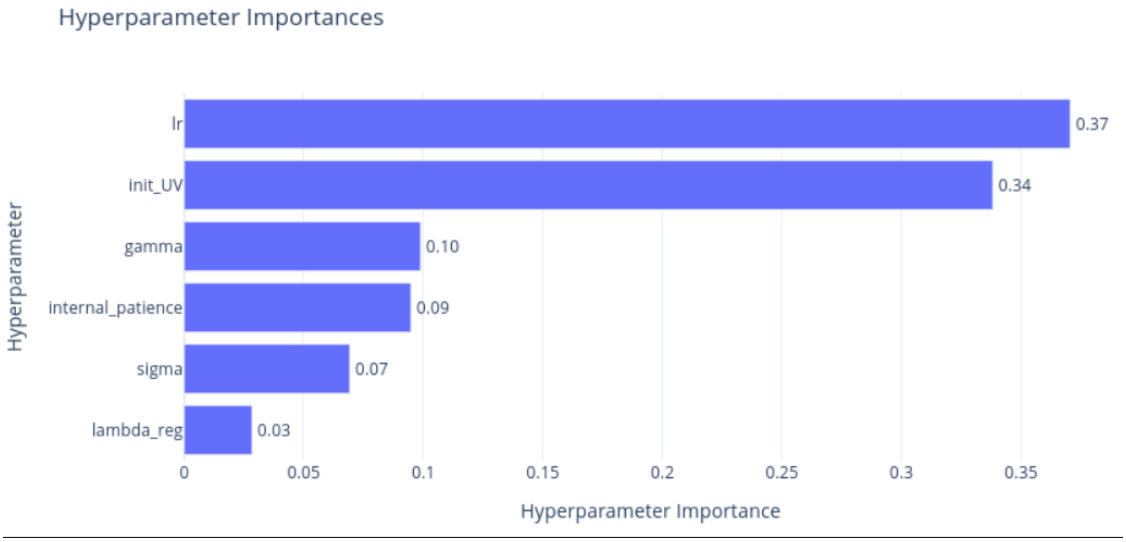


Figure 2: Hyperparameters importances.

The results of this hyperparameter optimization are summarized in Table 2.

Hyperparameter	**Value**
Best Rank (r)	5
Initialization Method ('init_UV')	SVD
Regularization Parameter (λ_{reg})	0,99

Table 2: Best Hyperparameters from Optuna Optimization

These parameters were used to train the final BCD model for 500 iterations, converging at iteration 151.

The training and validation losses are summarized below:

- Converged at iteration 151
- Training Loss = 0.79937
- Validation RMSE = 0.87613

The final model achieved a validation RMSE of 0.87613, indicating an improvement from earlier iterations.

Test Set Performance

The best model was then evaluated on a held-out test set ('inputEval.mat'), resulting in an RMSE of approximately 0.9386. A comparison of this result with the validation RMSE (0.7264) indicates that the model generalizes reasonably well to unseen data; however, some level of overfitting remains, as suggested by the slightly higher test RMSE compared to the validation RMSE. Further investigation into the differences in data distributions among training, validation, and test sets may be needed to understand this divergence.

Conclusions

The Block Coordinate Descent algorithm demonstrated effective performance in matrix factorization across both synthetic and Kaggle datasets, better then Nonnegative matrix factorisation. The hyperparameter tuning process using Optuna facilitated effective parameter selection, leading to improved model generalization as indicated by comparisons between validation and test set errors. Both training loss and validation loss show a clear pattern that aids in identifying optimal hyperparameters.

The optimal rank was found to be 5. While the model exhibits good generalization capabilities, some degree of overfitting warrants further investigation and potential refinements to reduce discrepancies between validation and test set performance. Additionally, exploring different initialization strategies and robust regularization techniques could enhance model performance.

5.3 Block Coordinate Gradient Descent Method

This section analyzes the results obtained using the Block Coordinate Gradient Descent (BCGD) algorithm with Armijo rule for line search, applied to a Kaggle dataset. The algorithm was executed for a total of 100 iterations, achieving a validation RMSE of 0.8761.

Kaggle Dataset and Hyperparameter Tuning

Hyperparameter tuning was performed on the Kaggle dataset using Optuna, aiming to minimize the validation loss. The Optuna study ran for 50 trials. The best trial ('best_trial_BCGD') yielded an initial validation RMSE of 3.7532.

Hyperparameter	Best Value
r	5
`init_UV`	SVD
λ_{reg}	0.0562
c	12
`Internal Patience`	9
γ	0.0240
β	0.0416
`Initial Validation RMSE`	3.7532

Table 3: Optuna Hyperparameter Optimization Results

Using the best hyperparameters from Optuna, the BCGD algorithm was executed. The training process revealed that the algorithm reached a plateau very early. The validation RMSE remained at approximately 3.75344 after only a few iterations, indicating a failure to converge to a better solution within the given constraints (max iterations = 100). The lack of improvement in the validation RMSE, despite the algorithm continuing to run, suggests that the algorithm was not converging effectively to

a better solution. Further investigation into the algorithm's behavior is needed to pinpoint the cause of this convergence failure.

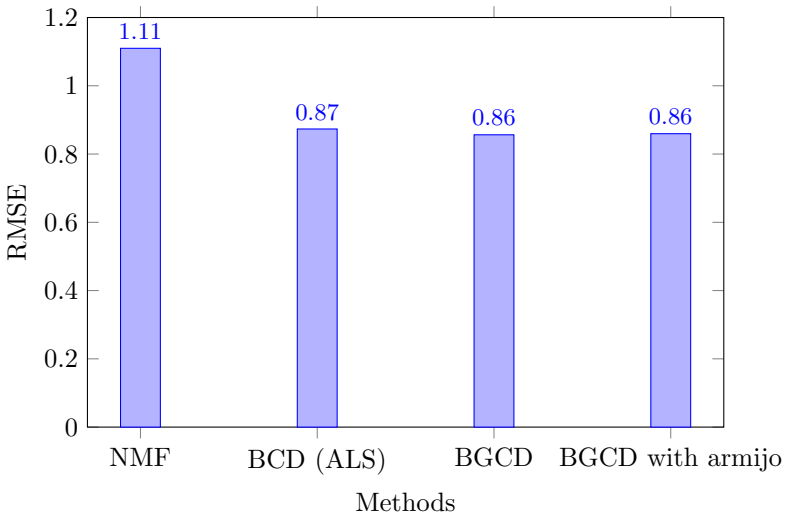
The final results from BCGD indicated a validation RMSE of 0.8761 after completing all iterations, which is significantly lower than the initial value but still requires further analysis to ensure robustness.

Conclusions

The BCGD shows the best rmse score on the test set with the value of 0.8564.

6 Overall Conclusions

This report provides a comparison of theese three algorithms to solve the problem: Non-Negative matrix factorization (NMF), Block Coordinate Descent with Alternating Least Squares and Block Coordinate Gradient Descent (BCGD). In addition to random initialization, initialization using SVD and a single matrix have also been tested. A histogram with the best results achieved for each model:



The Armiho rule was also used to calculate the step, which showed generally good results, although it took a long time to calculate. The quality of the reconstruction (measured by the RMSE error) may vary. In some cases, NMF can give better results, especially if decomposition with a more "physical" meaning is required (for example, in image analysis, where non-negativity is important for interpretation). BCD and BGCD tend to aim for local minima of the error function. In our case, NMF showed the worst result, but with further tuning of the parameters, its performance may improve. The best result was obtained using block gradient descent, in this case the step was selected adaptively

Future Directions

- Developing more adaptive initialization strategies that leverage dataset characteristics to provide more informed starting points for each algorithm.
- Investigating the interplay between initialization techniques, algorithm selection, and regularization methods to optimize performance and robustness.
- Exploring different line search methods or gradient descent variants (e.g., Adam, RMSprop).
- Re-evaluating the hyperparameter search space, potentially using more advanced optimization techniques or a wider range of values.

- Analyzing the data for potential characteristics that cause convergence difficulties (e.g., sparsity, noise levels, data distribution) - conduct EDA

These avenues of research will contribute towards developing more robust and efficient matrix factorization methods for various applications.