Analysis and Application of a Research Paper on Collaborative Filtering

Samuel METIN & Yacine ZALFANI

August 2025

Abstract

This report provides an overview of a research paper on collaborative filtering with a focus on item-based methods and the integration of Singular Value Decomposition (SVD). We summarize the main contributions, discuss their relevance, and propose our own small-scale experimental application.

Contents

1 Document Analysis

1.1 Introduction

Recommender systems are now essential in e-commerce and online services, where they must process millions of users and items in real time. Traditional item-based collaborative filtering (CF) relies on user ratings and item similarities, but struggles with scalability, data sparsity, and the inability to adapt to changes in user preferences.

The paper we study proposes two main improvements. First, it gives more weight to recent ratings, capturing the idea that preferences evolve over time. Second, it uses SVD to reduce dimensionality and handle sparsity, making the system faster and more memory-efficient. Together, these elements create a model that is both more accurate and more responsive to user behavior.

1.2 Related Work

Recommender systems usually follow either a collaborative or a content-based approach. Both face issues such as sparse data, scalability, and temporal variation. To overcome these, many authors have proposed hybrid models, improved similarity measures, or temporal graphs. Dimensionality reduction and distributed computing have also been explored to reduce cost.

The general trend is clear: modern recommendation research focuses on handling sparsity, scaling to large datasets, and incorporating temporal dynamics.

1.3 Item-Based Collaborative Filtering

In item-based CF, a user's preference for an item is predicted from similarities between items rather than between users. Two items i and j are considered similar if they are consistently rated in a similar way. The Pearson correlation is commonly used:

$$w_{i,j} = \frac{\sum_{u \in U}(r_{u,i} - \overline{r_i})(r_{u,j} - \overline{r_j})}{\sqrt{\sum_{u \in U}(r_{u,i} - \overline{r_i})^2} \cdot \sqrt{\sum_{u \in U}(r_{u,j} - \overline{r_j})^2}} \tag{1}$$

Here $r_{u,i}$ is the rating of user u for item i, $\overline{r_i}$ is the average rating of i, and U is the set of users who rated both i and j.

Once similarities are computed, the predicted rating of user a on item i is:

$$p_{a,i} = \frac{\sum_N (w_{i,N} \times r_{a,N})}{\sum_N |w_{i,N}|} \tag{2}$$

This amounts to a weighted average of the user's ratings on items similar to i. Although efficient, this method still suffers from sparsity and scalability issues, which motivates the use of SVD.

1.4 Singular Value Decomposition

SVD decomposes a matrix A of size $m \times n$ as:

$$A = U \times S \times V^T \tag{3}$$

U and V contain the left and right singular vectors, while S is diagonal with non-negative singular values $s_1 \geq s_2 \geq \cdots \geq s_r$. Truncated SVD keeps only the top k values:

$$A_k = U_k \times S_k \times V_k^T \tag{4}$$

This reduced representation captures the main user–item interactions while discarding noise. It decreases computational costs and reveals latent structures that improve recommendation accuracy.

1.5 SVD in Item-Based CF

Applying SVD to the rating matrix projects users and items into a lower-dimensional latent space. For user u and item i, the predicted rating can be written as:

$$p_{u,i} = A_u + U_k\sqrt{S_k}(u) \cdot \sqrt{S_k}V_k^T(i) \tag{5}$$

where A_u is the average score of user u, and the two factors represent their positions in the latent space. The entire matrix can be approximated as:

$$A_{m\times n} \approx U_{m\times k}\sqrt{S_{k\times k}} \cdot \sqrt{S_{k\times k}}V_{n\times k}^T \tag{6}$$

This approach reduces dimensionality, captures hidden relationships, and allows faster, more accurate predictions.

1.6 Proposed Solution

The paper extends item-based CF by introducing temporal dynamics. Each rating $r_{n,m}$ is associated with its age, i.e., the time between the rating date and the prediction date. A weighting function is applied:

$$Age_w = m \cdot Age_r + n \tag{7}$$

where Age_r is the raw age, and m, n are parameters controlling the decay. Older ratings thus receive less weight, while recent ones count more. This makes predictions more aligned with a user's current interests.

2 Experimental Study - Methodology

Our work builds on item-based collaborative filtering, extended with temporal dynamics and dimensionality reduction. The implementation proceeds in several stages.

2.1 Data Preprocessing

The raw datasets provide tuples of the form (user, item, rating, timestamp). If a user rated the same item multiple times, we kept only the most recent one. We also restricted the dataset to the last 10 years for Amazon dataset and 2 years for MovieLens' , and limited the analysis to the most active users and most frequently rated items to ensure sufficient density.

From the timestamp, we computed the "age" of each rating, i.e., how long ago it was given relative to the most recent rating in the dataset. This allows us to distinguish between old and recent ratings.

2.2 Temporal Weighting

To account for temporal dynamics, older ratings are down-weighted before model training. The intuition is that recent ratings better reflect current user preferences, while very old ratings should have less influence. After this adjustment, ratings are rescaled back into the standard $[1, 5]$ or $[0.5, 5]$ range.

2.3 Train–Test Split

We split the data chronologically: older ratings are used for training, and the most recent ones for testing. This setup ensures that the model is always predicting future behavior from past information.

2.4 Item-Based Collaborative Filtering

We build a user–item rating matrix and for the train matrix we center each user's ratings around their personal average. Item–item similarities are then computed using cosine similarity. Predictions for an unrated item are obtained as a weighted combination of the user's ratings on similar items. After prediction, the scores are shifted back to the original scale and rescaled to the $[1, 5]$ interval.

2.5 SVD Enhancement

To reduce sparsity and capture latent factors, we also apply truncated Singular Value Decomposition (SVD) to the training matrix. This provides a lower-rank approximation of the data, which is then used in the collaborative filtering pipeline. We evaluate both traditional collaborative filtering and time-weighted collaborative filtering in combination with SVD.

2.6 Neighborhood Selection

Finally, we refine predictions by restricting the similarity matrix to the top-N most similar items for each target item. This neighborhood approach reduces noise from weak similarities and improves accuracy.

2.7 Evaluation

Model performance is measured with Root Mean Squared Error (RMSE), reported as a percentage relative to the 1–5 rating scale. We compare traditional item-based CF, time-aware CF, and SVD-enhanced versions, with and without neighborhood selection.

3 Experimental Study - Code

Mining Massive Dataset - Project

Source : Collaborative Filtering with Temporal Dynamics with Using Singular Value Decomposition - Cigdem BAKIR (Netflix dataset) ___

Our Datasets :

MovieLens 25M : https://grouplens.org/datasets/movielens/25m/

Amazon Movies and TV - ratings only : https://nijianmo.github.io/amazon/index.html

```
import pandas as pd

names = ['userID', 'itemID', 'rating', 'timestamp']

#df = pd.read_csv('ratings.csv') ; df.columns = names

df = pd.read_csv('Movies_and_TV.csv', header=None, names=names)

print(df['rating'].min(), df['rating'].max())
df.head()
```

```
1.0 5.0
```

	userID	itemID	rating	timestamp	datetime	Age_r
0	0001527665	A3478QRKQDOPQ2	5.0	1362960000	2013-03-11	5.563313
1	0001527665	A2VHSG6TZHU1OB	5.0	1361145600	2013-02-18	5.620808
2	0001527665	A23EJWOW1TLENE	5.0	1358380800	2013-01-17	5.708419
3	0001527665	A1KM9FNEJ8Q171	5.0	1357776000	2013-01-10	5.727584
4	0001527665	A38LY2SSHVHRYB	4.0	1356480000	2012-12-26	5.768652

```
def add_age(df):
  df['datetime'] = pd.to_datetime(df['timestamp'], unit='s')
  last = df['datetime'].max()
  df['Age_r'] = (last - df['datetime']).dt.days / 365.25 # age in
years
  return df

df = add_age(df)

print(df.groupby('userID').size().median())
df.head()
```

```
4.0
```

	userID	itemID	rating	timestamp	datetime	Age_r
0	0001527665	A3478QRKQDOPQ2	5.0	1362960000	2013-03-11	5.563313
1	0001527665	A2VHSG6TZHU1OB	5.0	1361145600	2013-02-18	5.620808
2	0001527665	A23EJWOW1TLENE	5.0	1358380800	2013-01-17	5.708419

```
3  0001527665  A1KM9FNEJ8Q171     5.0  1357776000 2013-01-10  5.727584
4  0001527665  A38LY2SSHVHRYB     4.0  1356480000 2012-12-26  5.768652
```

```
age_max = 10
filtered_df = df.copy()

idx = df.groupby(['userID', 'itemID'])['Age_r'].idxmin()
filtered_df = filtered_df.loc[idx].copy() # if a user rated the same
item more than once we chose the last one

filtered_df = filtered_df[filtered_df['Age_r'] < age_max]

print("user*item all ratings : ", df['userID'].nunique(),
df['itemID'].nunique())
print(f"user*item last {age_max} years ratings : ",
filtered_df['userID'].nunique(), filtered_df['itemID'].nunique())

top_items = filtered_df['itemID'].value_counts().head(3000).index
filtered_df = filtered_df[filtered_df['itemID'].isin(top_items)]

top_users = filtered_df['userID'].value_counts().head(15000).index
filtered_df = filtered_df[filtered_df['userID'].isin(top_users)]

print("user*item final filtering : ", filtered_df['userID'].nunique(),
filtered_df['itemID'].nunique())
print(filtered_df.groupby('userID').size().median())
```

```
user*item all ratings :  182032 3826085
user*item last 10 years ratings :  170873 3513856
user*item final filtering :  15000 2993
11.0
```

Traditional Item Based CF

```
from sklearn.metrics import mean_squared_error
from math import sqrt
import numpy as np

def weight_ratings(df, n, m):
  df['rating'] = df['rating'] / (m * df['Age_r'] + n) # temporal
dynamics
  df['rating'] = 5 * df['rating'] / df['rating'].max() # rescaling
  return df

def train_test_split(df, n=0, m=1, age_threshold=21/12):
    user_item = df.pivot_table(index='userID', columns='itemID',
values='rating', fill_value=0)
    users = user_item.index
    items = user_item.columns
```

```
    train_df = df[df['Age_r'] >= age_threshold].copy()
    test_df = df[df['Age_r'] < age_threshold].copy()

    if n != 0 and m != 1:
        train_df = weight_ratings(train_df, n, m)
    train = train_df.pivot_table(index='userID', columns='itemID',
values='rating', fill_value=0)
    test = test_df.pivot_table(index='userID', columns='itemID',
values='rating', fill_value=0)

    train = train.reindex(index=users, columns=items,
fill_value=0).values
    test = test.reindex(index=users, columns=items,
fill_value=0).values

    # Test and training are truly disjoint
    assert(np.all((train * test) == 0))
    return train, test


def compute_similarity(ratings, kind='user', epsilon=1e-9):
    # epsilon -> small number for handling dived-by-zero errors
    if kind == 'user':
        sim = ratings.dot(ratings.T) + epsilon
    elif kind == 'item':
        sim = ratings.T.dot(ratings) + epsilon
    norms = np.array([np.sqrt(np.diagonal(sim))])
    return (sim / norms / norms.T)

def predict_simple(ratings, similarity, kind='user'):
    if kind == 'user':
        return similarity.dot(ratings) /
np.array([similarity.sum(axis=1)]).T
    elif kind == 'item':
        return ratings.dot(similarity) /
np.array([similarity.sum(axis=1)])

def get_rmse(pred, actual):
    # Ignore nonzero terms.
    pred = pred[actual.nonzero()].flatten()
    actual = actual[actual.nonzero()].flatten()
    return sqrt(mean_squared_error(pred, actual))

def standardize(train):
    train_std = train.copy().astype(float)
    user_means = []
    user_stds = []
    for i in range(train_std.shape[0]):
        row = train_std[i, :]
        mask = row != 0
```

```
        count = np.count_nonzero(mask)
        if count == 0:
            mean = 0.0
            std = 1.0
        else:
            mean = row[mask].mean()
            std = row[mask].std()
            if std == 0:
                std = 1.0 # iif the user rated all item the same score
        std = 1
        user_means.append(mean)
        user_stds.append(std)
        train_std[i, mask] = (row[mask] - mean) / std
    return train_std, np.array(user_means), np.array(user_stds)

def destandardize(predictions, user_means, user_stds):
    pred_destandardized = predictions.copy()
    for i in range(predictions.shape[0]):
        pred_destandardized[i, :] = user_stds[i] *
pred_destandardized[i, :] + user_means[i]
    return pred_destandardized

def predictions_rescaled(predictions, new_min=1, new_max=5):
    old_min = np.min(predictions)
    old_max = np.max(predictions)
    scaled = (predictions - old_min) / (old_max - old_min)
    return scaled * (new_max - new_min) + new_min

"""
# First version - No standardizing

train, test = train_test_split(filtered_df)
item_similarity = compute_similarity(train, kind='item')
item_prediction = predict_simple(train, item_similarity, kind='item')
"""

train, test = train_test_split(filtered_df)
train_std, user_means, user_stds = standardize(train)
item_similarity = compute_similarity(train_std, kind='item')
item_prediction_std = predict_simple(train_std, item_similarity,
kind='item')
item_prediction = destandardize(item_prediction_std, user_means,
user_stds)
item_prediction = predictions_rescaled(item_prediction)

error_rate = 100 * get_rmse(item_prediction, test) / 5
print('Traditional Item Based CF Error Rate (%): ', error_rate)

Traditional Item Based CF Error Rate (%):  37.7748924641529
```

```
total = train + test
nb_total = np.count_nonzero(total)
nb_train = np.count_nonzero(train)
nb_test = np.count_nonzero(test)
sparsity = 1 - (nb_total / total.size)

print(f"Total ratings       : {nb_total}")
print(f"Sparsity       : {sparsity:.2%}")
print(f"Train ratings (old) : {nb_train} ({nb_train / nb_total:.2%})")
print(f"Test ratings  (recent)    : {nb_test} ({nb_test /
nb_total:.2%})")
```

```
Total ratings       : 281846
Sparsity       : 99.37%
Train ratings (old) : 250912 (89.02%)
Test ratings  (recent)    : 30934 (10.98%)
```

```
print(filtered_df['Age_r'].max())
```

```
9.998631074606434
```

RMSE error rate obtained with temporal dynamics item-based CF

```
import matplotlib.pyplot as plt

Age_w = [1.1, 1.5, 2, 3, 5, 8, 10]
n_values = [0.25, 0.5, 0.75]
m_values = []
for i in range(len(n_values)):
  m_values.append([])
  for age in Age_w:
    m_values[i].append((age-n_values[i]) / 10)

Error_rate = []

n0, m0 = n_values[0], m_values[0][0] # n and m that minimize error
rate

for i in range(len(n_values)) :
  Error_rate.append([])
  for j in range(len(m_values[i])) :
    temp_df = filtered_df.copy()

    train, test = train_test_split(temp_df, n=n_values[i],
m=m_values[i][j])
    train_std, user_means, user_stds = standardize(train)
    item_similarity = compute_similarity(train_std, kind='item')
    item_prediction_std = predict_simple(train_std, item_similarity,
kind='item')
    item_prediction = destandardize(item_prediction_std, user_means,
```

```
user_stds)
    item_prediction = predictions_rescaled(item_prediction)
    error_rate = 100 * get_rmse(item_prediction, test) / 5

    print(error_rate)

    Error_rate[i].append(error_rate)

    if n_values[i] == n0 and m_values[i][j] == m0 :
      error_rate0 = error_rate
    if error_rate < error_rate0 :
      n0, m0 = n_values[i], m_values[i][j]
      error_rate0 = error_rate

  plt.plot(Age_w, Error_rate[i], marker='o', label=f'n =
{n_values[i]}')

plt.xlabel('Age_w')
plt.ylabel('Percentage Error')
plt.legend()
plt.show()

print(n0, m0, error_rate0)
```

```
0.25 0.975 22.650893097115613
```

Item based with temporal dynamics combined SVD and item based combined SVD

```
from sklearn.decomposition import TruncatedSVD

k_values = [5, 10, 20, 30, 40, 50, 100]

Error_rate_time = []
Error_rate_trad = []

for i in range(2):
    temp_df = filtered_df.copy()
    if i == 0 :
        train, test = train_test_split(temp_df, n=n0, m=m0) # First :
Temporal Dynamics CF
    else :
        train, test = train_test_split(temp_df) # Second : Traditional
CF

    for k in k_values:
        svd = TruncatedSVD(n_components=k, random_state=42)

        U = svd.fit_transform(train)
        Vt = svd.components_
        train_approx = np.dot(U, Vt)
```

```
        train_std, user_means, user_stds = standardize(train_approx)
        item_similarity = compute_similarity(train_std, kind='item')
        item_prediction_std = predict_simple(train_std,
item_similarity, kind='item')
        item_prediction = destandardize(item_prediction_std,
user_means, user_stds)
        item_prediction = predictions_rescaled(item_prediction)
        error_rate = 100 * get_rmse(item_prediction, test) / 5
        print(error_rate)

        if i == 0 :
            Error_rate_time.append(error_rate) # First : Temporal
Dynamics CF
        else :
            Error_rate_trad.append(error_rate) # Second : Traditional
CF

plt.plot(k_values, Error_rate_time, marker='o', color='red',
label='Time Dependent CF with SVD')
plt.plot(k_values, Error_rate_trad, marker='o', color='blue',
label='Traditional CF with SVD')

plt.xlabel('Number of components k')
plt.ylabel('Percentage Error')
plt.legend()
plt.show()
```

Traditional Item based CF and item based CF with temporal dynamics results by taking neighborhoods

```
import matplotlib.pyplot as plt
from sklearn.decomposition import TruncatedSVD

def top_N_similarity_matrix(similarity_matrix, N):
    n_items = similarity_matrix.shape[0]
    reduced_sim = np.zeros_like(similarity_matrix)
    for i in range(n_items):
        sim_row = similarity_matrix[i].copy()
        sim_row[i] = -np.inf  # ignore self-similarity

        top_N_idx = np.argpartition(sim_row, -N)[-N:]
        for j in top_N_idx:
            reduced_sim[i, j] = similarity_matrix[i, j]

    return reduced_sim


k0 = 50
Error_rate_time = []
```

```
Error_rate_trad = []
N_values = [50, 300, 600, 900, 1200, 1500, 1800, 2100, 2400, 2700,
2993]

svd = TruncatedSVD(n_components=k0, random_state=42)

for i in range(2):
    temp_df = filtered_df.copy()
    if i == 0 :
        train, test = train_test_split(temp_df, n=n0, m=m0) # First :
Temporal Dynamics CF
        U = svd.fit_transform(train)
        Vt = svd.components_
        train = np.dot(U, Vt)
    else :
        train, test = train_test_split(temp_df) # Second : Traditional
CF

    train_std, user_means, user_stds = standardize(train)
    item_similarity = compute_similarity(train_std, kind='item')

    for N in N_values :
        top_N_item_similarity =
top_N_similarity_matrix(item_similarity, N) # Similarity on the
neighborhood
        item_prediction_std = predict_simple(train_std,
top_N_item_similarity, kind='item')
        item_prediction = destandardize(item_prediction_std,
user_means, user_stds)
        item_prediction = predictions_rescaled(item_prediction)
        error_rate = 100 * get_rmse(item_prediction, test) / 5
        print(error_rate)

        if i == 0 :
            Error_rate_time.append(error_rate) # First : Temporal
Dynamics CF
        else :
            Error_rate_trad.append(error_rate) # Second : Traditional
CF


plt.plot(N_values, Error_rate_time, marker='o', color='red',
label='Time Dependent CF with SVD computed on neighborhoods')
plt.plot(N_values, Error_rate_trad, marker='o', color='blue',
label='Traditional CF computed on neighborhoods')

plt.xlabel('Top N neighbors')
plt.ylabel('Percentage Error')
plt.legend()
plt.show()
```

4 Results

4.1 Amazon Dataset

First Step : Traditional CF Item-Based

Train + Test Setup

We use the last 10 years of ratings for training and testing.

Test Results by Maximum Age

Test maximum age	Test ratings (count, %)	Trad. CF Item-Based Error (%)
20 months	28017 (9.94%)	84.74%
21 months	30934 (10.98%)	84.90%
22 months	33689 (11.95%)	85.09%

Table 1: Traditional CF Item-Based Percentage Error by Test Age

The percentage error is almost the same across different test ages. However, the test set must:

- represent at least 10% of all ratings,
- be as recent as possible, since we will predict ratings with a time-dependent CF and cannot use the date of a hypothetical rating.

Therefore, we select **21 months** for the rest of the experiment.

Sparsity of the User-Item Matrix

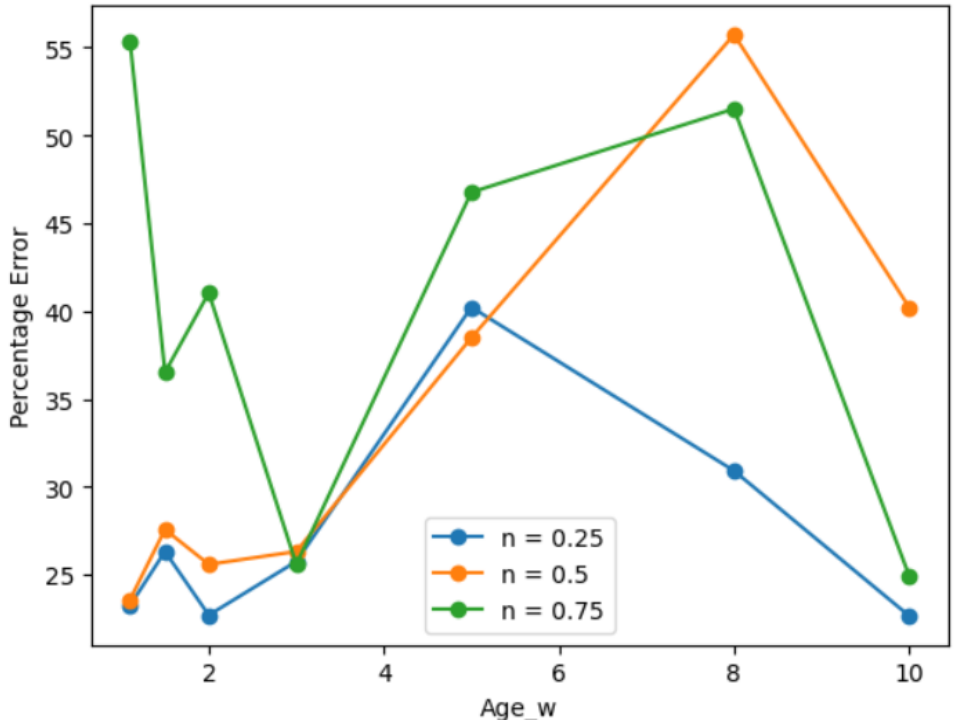
Our user-item matrix has a sparsity of 99.37%, which is a bit too high for a collaborative filtering approach.

Impact of Centering and Standardization

- Trad. CF Item-Based Error after centering the train: **37.77%**
- Trad. CF Item-Based Error after standardizing the train: **44.05%**

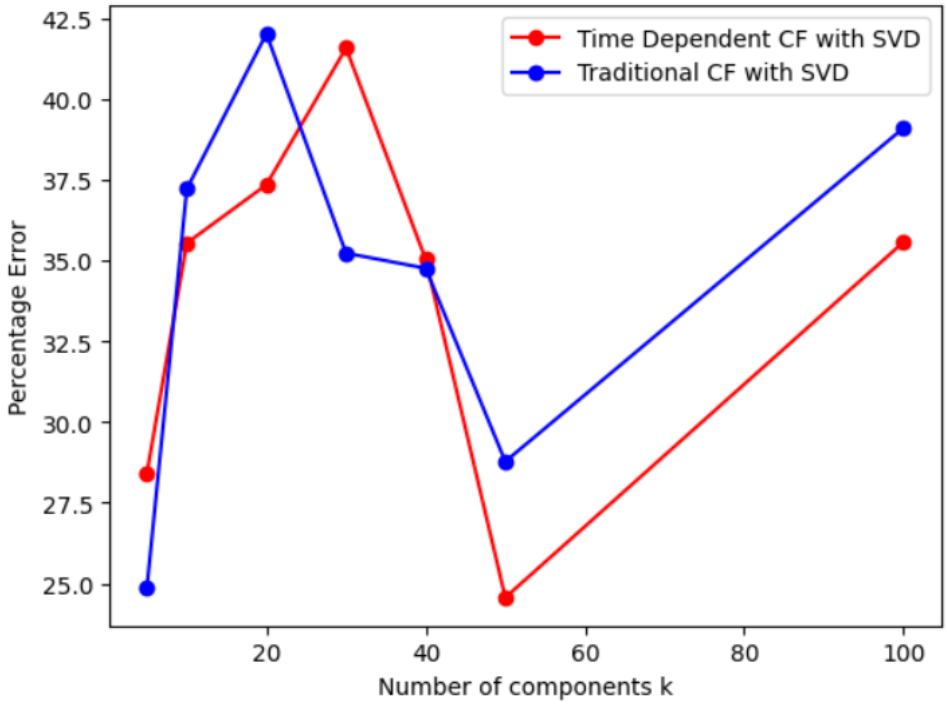
Thus, we choose to **center the train data** for the rest of the experiment.

Second Step : Temporal Dynamics CF Item Based



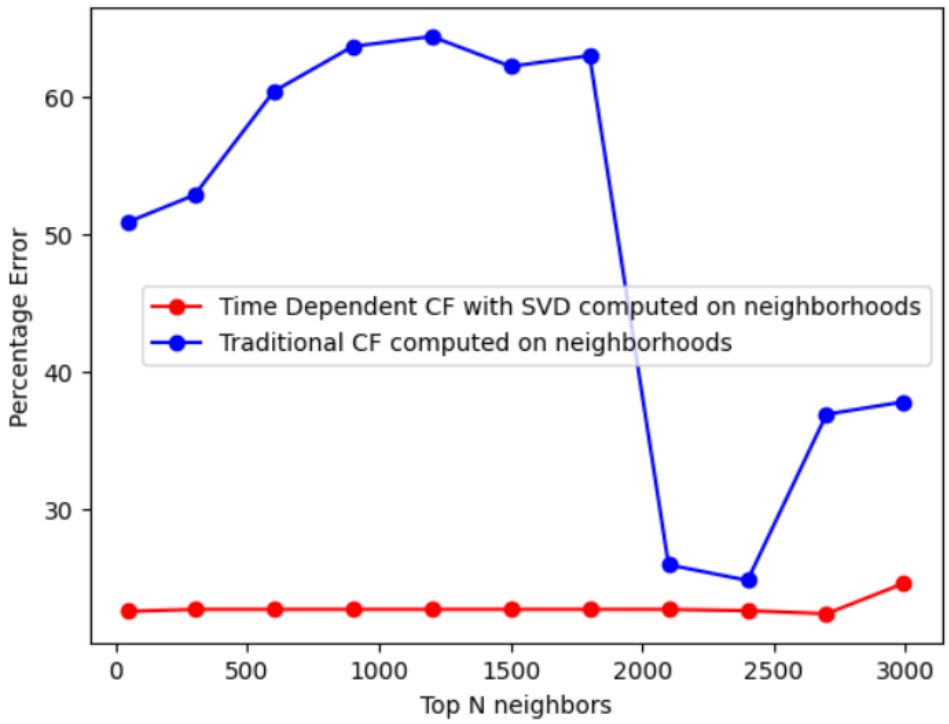
The optimal parameters are obtained for $n = 0.25$ and $Age_w = 10.0$, which corresponds to $m = 0.975$. This configuration minimizes the error at **22.65%**.

Third Step : Adding SVD method



For the time-dependent CF with SVD, the best performance is achieved at $k = 50$, with an error of **24.56%**. This is worse than the best result obtained without SVD, showing that dimensionality reduction did not improve accuracy in this case.

Fourth Step : CF on neighborhoods



When restricting to neighborhoods, the time-dependent CF with SVD achieves better results than the traditional CF approach, highlighting the effectiveness of combining temporal dynamics with SVD.

4.2 MovieLens Dataset

First step : Trad CF Item Based

- **Train** + **Test** = last 2 years ratings
- **Test maximum age / Test ratings / Trad CF Item Based Percentage Error**

Test maximum age	Test ratings	Percentage Error
2 months	164398 (7.55%)	74.31%
3 months	262100 (12.04%)	74.85%
4 months	343990 (15.81%)	74.57%

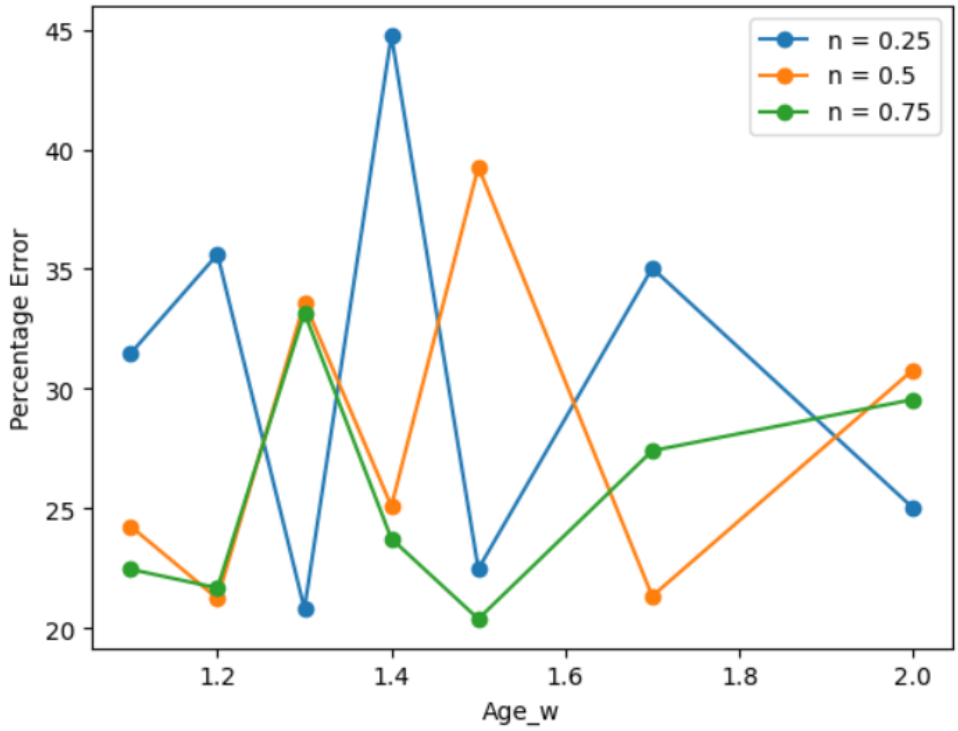
- **Choice of test set:** Percentage error almost the same but the test should be:
  - at least 10% of all ratings,

– as recent as possible (for the same reason as before).

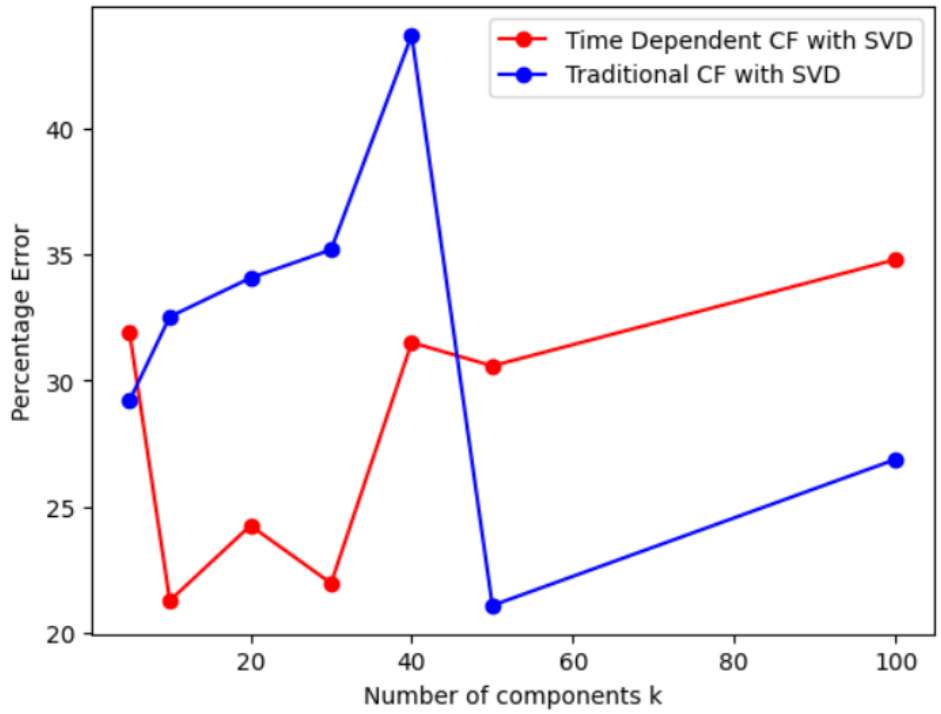
⇒ We take **3 months** for the rest of the experiment.

- **User-item matrix sparsity:** 95.16% → usual value for a CF.
- **Trad CF Item Based Percentage Error:**
  - after centering the train: 25.77%
  - after standardizing the train: 24.00%

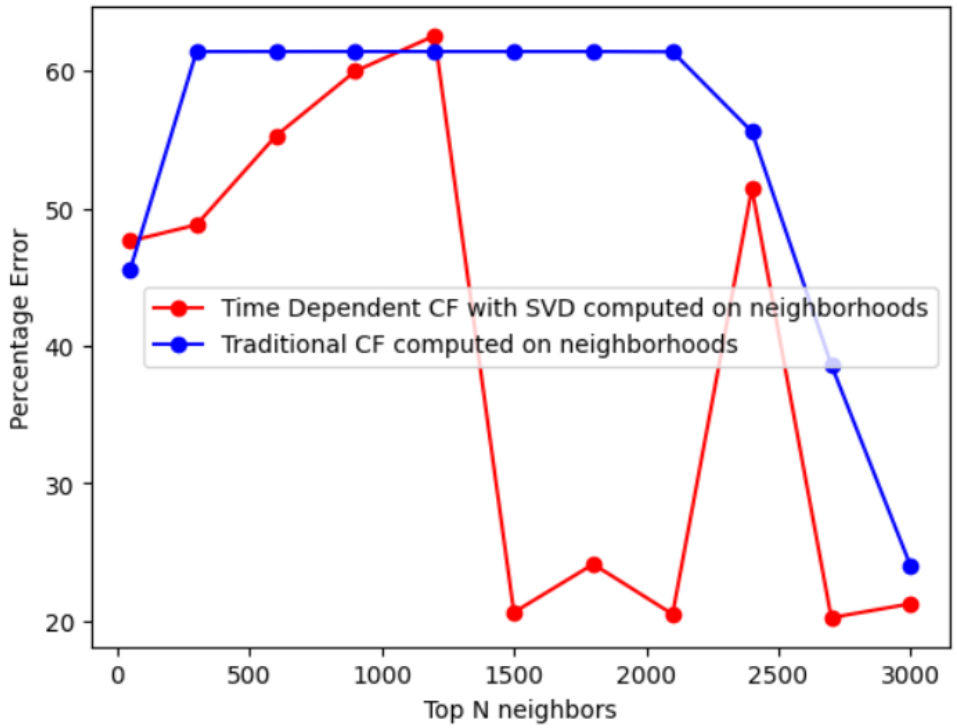
  ⇒ We **standardize the train** for the rest of the experiment.



The optimal parameters are obtained for $n = 0.75$ and $Age_w = 1.5$, which corresponds to $m = 0.375$. This configuration minimizes the error at **20.38%**.



For the time-dependent CF with SVD, the best performance is achieved at $k = 10$, with an error of **21.26%**. However, this remains worse than the best result obtained without SVD, showing that dimensionality reduction did not improve accuracy in this case.



When restricting to neighborhoods, the improved time-dependent CF yields better results compared to the traditional CF baseline, highlighting the benefit of incorporating temporal dynamics and adjustments.

5 Conclusion

The time-dependent CF approach consistently improved performance over traditional item-based CF across datasets, confirming that accounting for the temporal dimension of ratings enhances recommendation quality. However, the integration of SVD yielded mixed results: while it did not improve error rates in all cases (e.g., MovieLens with $k = 50$), it stabilized and improved neighborhood-based methods.