Realization of A Funk SVD Based Music Recommendation

Yigang Meng

07/06/2023

Introduction: Funk SVD - What's in it?

Funk SVD is an algorithm introduced by Simon Funk, which is a matrix factorization algorithm that can be used in giving recommended items to users. In this project, we will dive into several parts of it, which are the math behind the Funk SVD, the limitation of the traditional singular values decomposition in recommendation systems, the implementation including iteratively finding the constructing matrix using SGD(Stochastic gradient descent), and the comparison to the Funk SVD function from `recommenderlab`.

Background: Why Funk SVD? (Why not SVD?)

Rating table are often sparse. And when it comes to a recommendation system, we really want to "fit" the missing values based on the ratings in the original matrix. Given a real matrix R .The traditional SVD is given by:

$$R = U\Sigma V^T$$

, where U is orthogonal whose columns are the left singular vectors of R . Σ is a diagonal matrix consists of the singular values, and V^T is also orthogonal and its rows are the right singular vectors of R .

While as we mentioned in the introduction, the traditional is naturally not able to handle the missing values, and it can be really computationally expensive. One want to do SVD on an incomplete matrix, one must fill out the data using other values such as global mean $\bar{X}$. And then we can perform SVD to get the estimated $\hat{R}$.

Funk SVD is a model that has latent factors. Latent factors are implicitly defined by the model itself, and it's hard to interpret sometimes, but it can be really helpful in finding the underlying pattern that is driven by user-item matrices.(Figure1) One can think of latent factors as the "gradients" of the items, and we add different amount of each gradient to construct a portrait of a user.

lfactor1	1.2161765	0.2051823	1.3332590	1.3744080	0.4355446
lfactor2	0.7455931	2.1740190	-0.2468692	0.9019031	0.9019031
lfactor3	1.0275480	0.4775867	0.7218195	1.0332071	1.3603743

lfactor1	lfactor2	lfactor3
1.973623	-0.103482	1.579164
0.372411	1.967844	1.045997
1.421002	1.529389	1.147124

	item1	item2	item3	item4	item5
User1	4	1	4	4	3
User2	3	5	1	3	3
User3	4	4	2	5	3

Figure1

A Funk SVD defined on a matrix $R_{m\times n}$ is as follow

$$R \approx \mathrm{UV^T}$$

where $U_{m\times k}$ and $V^T_{k\times n}$ are two low rank matrices which has the latent factors of the users and the items, respectively. Our goal is to find the U_i and V_j^T via minimizing the loss function:

$$J(U_i, V_j^T) = \min \Sigma_{\mathrm{i,j\in train}} \left(\mathrm{R_{ij} - U_i V_j^T}\right)^2 + \lambda \left(\|\mathrm{U_i}\|^2 + \|\mathrm{V_j^T}\|^2\right)$$

, where λ is the regularization parameter. (expalain what is regularization parameter). And we take the partial derivatives with respects to U_i and V_i^T, respectively, we get:

$$\frac{\partial J\left(U_i, V_j^T\right)}{\partial U_i} = \sum_{j\in \text{ Train}} \left[-2\left(R_{ij} - U_i V_j^T\right) V_j^T\right] + 2\lambda U_i$$

$$\frac{\partial J\left(U_i, V_j^T\right)}{\partial V_j} = \sum_{i\in \text{ Train}} \left[-2\left(R_{ij} - U_i V_j^T\right) U_i\right] + 2\lambda V_j$$

In Stochastic gradient descent method, the hyper parameter $\alpha = 2c$ that is served as the how precisely we want to move in a direction. And one updates two variables along with the opposite of the gradient(Yadav, 2020) to find U_i and V_i^T :

$$U_i \leftarrow U_i + \alpha \cdot \left(\left(R_{ij} - U_i V_j^T\right) V_j - 2\lambda U_i\right)$$

$$V_j \leftarrow V_j + \alpha \cdot \left(\left(R_{ij} - U_i V_j^T\right) U_i - 2\lambda V_j\right)$$

Perform on the Amazon digital music data base

Computation:

Data importing, cleaning, and filtering

```
library(dplyr)
```

```
##
## Attaching package: 'dplyr'
```

```
## The following objects are masked from 'package:stats':
##
##     filter, lag
```

```
## The following objects are masked from 'package:base':
##
##     intersect, setdiff, setequal, union
```

```
library(tidyverse)
```

```
## ── Attaching core tidyverse packages ──────────────────────── tidyverse 2.0.0 ──
## ✔ forcats   1.0.0     ✔ readr     2.1.4
## ✔ ggplot2   3.4.1     ✔ stringr   1.5.0
## ✔ lubridate 1.9.2     ✔ tibble    3.1.8
## ✔ purrr     1.0.1     ✔ tidyr     1.3.0
```

```
## ── Conflicts ────────────────────────────────────────── tidyverse_conflicts() ──
## ✖ dplyr::filter() masks stats::filter()
## ✖ dplyr::lag()    masks stats::lag()
## ℹ Use the ]8;;http://conflicted.r-lib.org/conflicted package]8;; to force all conflicts to become errors
```

```
df <- read.csv("ratings_Digital_Music.csv")
colnames(df) <- c("user","item","rating")
df <- df[, -4]
# Filter out users with fewer than 10 ratings
df <- df %>% group_by(user) %>% filter(n() >= 30)

# Filter out items with fewer than 10 ratings
df <- df %>% group_by(item) %>% filter(n() >= 40)
user_item_matrix <- df %>%
  spread(key = item, value = rating)
```

```
df1 <- as.data.frame(user_item_matrix)
rownames(df1) <- df1[,1]
df1 <- df1[,-1]
head(df1)
```

```
##                B00000016W B000001A5X B0000025F7 B0000025RI B000002KH3
## A103W7ZPKGOCC9         NA          5         NA          5         NA
## A12R54MKO17TW0          5          5         NA         NA          5
## A12W8NRSYR593I         NA          5         NA         NA         NA
## A13IKSGDYNBNQS          5         NA         NA         NA         NA
## A149OQJD0ASI07         NA         NA         NA          5         NA
## A14GK0E64J0WAS         NA         NA         NA         NA         NA
##                B000002LGQ B000002NJS B000002OPL B00004T9UF B00005O54Q
## A103W7ZPKGOCC9         NA         NA         NA         NA         NA
## A12R54MKO17TW0         NA         NA          5         NA         NA
## A12W8NRSYR593I         NA         NA         NA         NA         NA
## A13IKSGDYNBNQS         NA         NA         NA         NA         NA
## A149OQJD0ASI07         NA         NA         NA          5         NA
## A14GK0E64J0WAS         NA         NA         NA         NA         NA
##                B00005YW4H B000084T18 B0000AGWFX B0000DD7LC B0007NFL18
## A103W7ZPKGOCC9         NA         NA         NA         NA         NA
## A12R54MKO17TW0         NA         NA         NA         NA         NA
## A12W8NRSYR593I         NA         NA         NA         NA         NA
## A13IKSGDYNBNQS         NA         NA         NA         NA         NA
## A149OQJD0ASI07         NA          3         NA          3         NA
## A14GK0E64J0WAS          5         NA         NA         NA         NA
##                B000N2G3RY
## A103W7ZPKGOCC9         NA
## A12R54MKO17TW0         NA
## A12W8NRSYR593I         NA
## A13IKSGDYNBNQS          4
## A149OQJD0ASI07         NA
## A14GK0E64J0WAS          4
```

```
dim(df1)
```

```
## [1] 284  16
```

```
R <- df1
colnames(R) <- NULL
rownames(R) <- NULL
dim(R)
```

```
## [1] 284  16
```

```
R <- as.matrix(R)
```

Split the data into training and testing sets

We randomly hide 5 ratings from the matrix, thereby we can construct a training data set.

```
n <- 5  # The number of ratings to hide per user
train_data <- R
test_data <- matrix(NA, nrow = nrow(R), ncol = ncol(R))

for (i in 1:nrow(R)) {
  rated_items <- which(!is.na(R[i, ]))
  if (length(rated_items) > n) {
    test_items <- sample(rated_items, n)
    test_data[i, test_items] <- R[i, test_items]
    train_data[i, test_items] <- NA
  }
}
```

Train on training data set

For the parameters, we want to get the optimal number of the latent factors with a small mean square error and smaller index(the number of factors also needs to be less than the number of columns as well). It is because we need to consider the trade-off between the accuracy and complexity, where we avoid the potential over-fitting of our model.

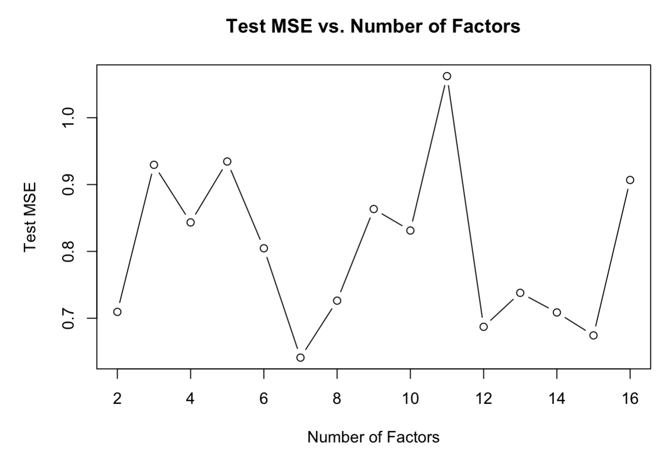
```
set.seed(123)
n_users <- nrow(R)
n_items <- ncol(R)
learning_rate <- 0.01
n_epochs <- 100
tolerance <- 1e-4  # Set a tolerance for the change in error
lambda <- 0.05  # Set the regularization parameter
errors <- c()  # Initialize a vector to store the errors
n_factors_range <- 2:16
mse_results <- c()
for (n_factors in n_factors_range) {

  P <- matrix(runif(n_users * n_factors), n_users, n_factors)
  Q <- matrix(runif(n_items * n_factors), n_items, n_factors)
  errors <- c()

  for (epoch in 1:n_epochs) {
    total_error <- 0
    for (i in 1:n_users) {
      for (j in 1:n_items) {
        if (!is.na(train_data[i, j])) {
          error <- train_data[i, j] - P[i, ] %*% Q[j, ]
          total_error <- total_error + error^2
          P[i, ] <- P[i, ] + learning_rate * (as.numeric(error) * Q[j, ] - lambda * P[i, ])
          Q[j, ] <- Q[j, ] + learning_rate * (as.numeric(error) * P[i, ] - lambda * Q[j, ])
        }
      }
    }
    errors <- c(errors, total_error)
    if (epoch > 1 && abs(errors[epoch] - errors[epoch-1]) < tolerance) {
      break
    }
  }

  R_hat <- P %*% t(Q)
  test_mse <- mean((test_data - R_hat)^2, na.rm = TRUE)
  mse_results <- c(mse_results, test_mse)
}

plot(n_factors_range, mse_results, type = 'b', xlab = 'Number of Factors', ylab = 'Test MSE', main = 'Test MSE v
s. Number of Factors')
```




```
optimal_n_factors <- min(order(mse_results)[1:2])
```

Run the Funk SVD with the optimal number of latent factors

In the code chunk below, we inherent the optimal number of latent factors we got from above, and run the Funk SVD by epochs of 100. Moreover, we set a tolerance to end the calculation early when the error is smaller than it. This is a trade off between the time consumed and the accuracy, which will be furtherly discussed after. The time consumed is shown below.

```
set.seed(123)
n_users <- nrow(R)
n_items <- ncol(R)
learning_rate <- 0.01
n_epochs <- 100
tolerance <- 1e-4  # Set a tolerance for the change in error
gamma <- 0.01  # Set the regularization parameter
errors <- c()  # Initialize a vector to store the errors
n_factors <- optimal_n_factors

start_time <- Sys.time()
for (epoch in 1:n_epochs) {
    total_error <- 0  # Initialize the total error for this epoch

    for (i in 1:n_users) {
        for (j in 1:n_items) {
            if (!is.na(train_data[i, j])) {
                # Calculate the error
                error <- train_data[i, j] - P[i, ] %*% Q[j, ]
                # Add the squared error to the total error
                total_error <- total_error + error^2

                # Update P and Q matrices using gradient descent
                P[i, ] <- P[i, ] + learning_rate * (as.numeric(error) * Q[j, ] - gamma * P[i, ])
                Q[j, ] <- Q[j, ] + learning_rate * (as.numeric(error) * P[i, ] - gamma * Q[j, ])
            }
        }
    }

    # Store the total error for this epoch
    errors <- c(errors, total_error)

    # Check if the change in error is less than the tolerance
    if (epoch > 1 && abs(errors[epoch] - errors[epoch-1]) < tolerance) {
        break  # Stop training
    }
}

end_time <- Sys.time()
time_consumed <- end_time - start_time
print(time_consumed)
```

```
## Time difference of 1.031238 secs
```

Calculate the MSE for the given model

We calculate the mean squares error between the test hat and the test data, and it's around 0.8, which is a acceptable result given that the rating spreads from 1 to 5.

```
test_hat <- P %*% t(Q)

# Calculate the MSE on the test data
test_mse <- mean((test_data - test_hat)^2, na.rm = TRUE)
print(test_mse)
```

```
## [1] 0.9731667
```

Comparison

We now use the `funkSVD` function from `recommenderb` package, we get the test_hat_RCML and the MSE of it. Compared to our Funk SVD implementation, it consumed more time, but get a smaller MSE.

There are several reasons related to the result:

1. The `funkSVD` function doesn't have a tolerance when iteratively finding U and V^T , and it may take more time but get a relatively accurate prediction.
2. The initialization of the user and item matrices is different where `funkSVD` use the default values of 0.1's for them, this might lead to the fact that it can somehow finding the global minima instead of local.
3. The most possible reason is the convergence criteria. Inside of the `funkSVD`, the algorithm is still training until both conditions about the min_improvement and the min_epochs are met, which means that it will get more accurate result by making sure the number of epochs is always larger than the given one. And it further explain why it takes more time than ours in average.

```
library(recommenderlab)
```

```
## Loading required package: Matrix
```

```
##
## Attaching package: 'Matrix'
```

```
## The following objects are masked from 'package:tidyr':
##
##     expand, pack, unpack
```

```
## Loading required package: arules
```

```
##
## Attaching package: 'arules'
```

```
## The following object is masked from 'package:dplyr':
##
##     recode
```

```
## The following objects are masked from 'package:base':
##
##     abbreviate, write
```

```
## Loading required package: proxy
```

```
##
## Attaching package: 'proxy'
```

```
## The following object is masked from 'package:Matrix':
##
##     as.matrix
```

```
## The following objects are masked from 'package:stats':
##
##     as.dist, dist
```

```
## The following object is masked from 'package:base':
##
##     as.matrix
```

```
## Registered S3 methods overwritten by 'registry':
##   method               from
##   print.registry_field proxy
##   print.registry_entry proxy
```

```
n_factors <- optimal_n_factors
start_time <- Sys.time()

result <- funkSVD(train_data, k = n_factors, lambda = 0.01, gamma = 0.01, min_epochs = 100, max_epochs = 100, ver
bose = FALSE)

end_time <- Sys.time()
time_consumed <- end_time - start_time
print(time_consumed)
```

```
## Time difference of 1.111222 secs
```

```
test_hat_RCML <- result$U %*% t(result$V)
test_mse_RCML <- mean((test_data - test_hat_RCML)^2, na.rm = TRUE)
print(test_mse_RCML)
```

```
## [1] 0.7351172
```

Conclusion

In conclusion, we have explored the concept of Funk SVD, a matrix factorization algorithm used in recommendation systems. We began by comparing it to traditional SVD, highlighting the limitations of the latter in the context of recommendation systems. We then delved into the mathematics underpinning Funk SVD, explaining how it works and why it is effective for generating recommendations from a sparse user-item matrix.

We implemented the Funk SVD algorithm, focusing on the iterative process of finding the optimal matrices using Stochastic Gradient Descent (SGD). We also discussed how to determine the optimal number of factors.

Finally, we compared our implementation with the **`funkSVD`** function from the **`recommenderlab`** package. We found that while our implementation was able to achieve a reasonable MSE, the **`recommenderlab`** function achieved a lower MSE, possibly due to differences in the convergence criteria used in the training process.

Reference

Gupta, P. (2017, November 16). *Regularization in machine learning*. Medium. https://towardsdatascience.com/regularization-in-machine-learning-76441ddcf99a#:~:text=Regularization%2C%20significantly%20reduces%20the%20variance,impact%20on%20bias%20and%20variance.

Improving regularized singular value decomposition for collaborative ... (n.d.). https://www.cs.uic.edu/~liub/KDD-cup-2007/proceedings/Regular-Paterek.pdf

Yadav, A. (2020, February 20). *Why we move opposite to gradients in gradient descent??*. Medium. https://medium.com/analytics-vidhya/why-we-move-opposite-to-gradients-in-gradient-descent-9077b9aa68e4

Zhang, Y. (2022, March 12). *An introduction to matrix factorization and factorization machines in recommendation system, and beyond*. arXiv.org. https://arxiv.org/abs/2203.11026