Weighted Alternating Least Squares with Implicit Feedback Data[1]

William Morgan, STP 598 Spring 2018

I. INTRODUCTION

A. Recommender Systems

Online retailers and media services in the modern world have vast catalogues available to their clientele. Without any guidance consumers must sift through numerous options for even the simplest item. Consumers faced with too many options can experience decision fatigue, an idea that describes the decline in decision-making quality over long sessions [2]. It can even cause consumers to avoid decisions entirely. Overall, more consumers are less satisfied with their choices and are less likely to return in the future. This is unsatisfactory for both consumers and retailers. Hence, there is a major incentive in simplifying the menu of choices for a particular item. To accomplish this, firms typically employ recommendation systems, whereby data associated with users and products is used to create personalized recommendations.

Recommendation systems are broadly categorized in two ways - content-based and collaborative filtering. Content-based recommendation systems profiles users and items based on their characteristics in order to build recommendations. This can include survey responses, demographics, and other personal characteristics for users, while content profiles for items are more domain-dependent. As an example, the cast, budget, and genre could all be used in a content-based recommendation system for movies. On the other hand, collaborative filtering techniques rely on aggregating user-item interactions. Essentially, past user behavior is used to predict future behavior. One of the more famous examples of collaborative filtering being used was during the Netflix Prize competition, where participants competed in building collaborative filtering algorithms for predicting user ratings for films. In this example, the winners developed the BigChaos algorithm [8] using a training set of more than 100 million spanning 480,000 users and nearly 18,000 movies [12].

B. Explicit and Implicit Feedback

An important distinction to make in the Netflix dataset is that the user-item interactions are movie ratings. To be included in the data set, users must (of their own volition) choose to rate the movies they watch. Thus, this data set relies entirely on users who choose to rate. This is problematic as it inherently excludes users who do not give their opinion, potentially creating issues with selection bias. Furthermore, ratings can be prohibitively expensive to collect. As a result, many collaborative filtering data sets instead quantify the number of interactions a user has with a particular item. Interactions can be defined in a variety of ways depending on the domain, but it is generally thought of as a user accessing an item in a catalogue. This type of data works around these restrictions at the cost of additional noise. In general, these characteristics define what are known as explicit and implicit feedback data sets.

Implicit feedback data sets come with several caveats that limit the ways they can be used to build recommender systems. Most importantly, these data sets do not contain any negative feedback since they simply count the number of times a user interacts with an item. This skews our representation of a user's preferences and forces us to creatively make use of missing entries, where most of the negative feedback would be found. A related problem with implicit feedback is that low-frequency entries add a significant amount of noise to the data. Items with one or two accesses do not necessarily capture a user's motives and in fact can represent something else entirely. It could be the case that a user was purchasing an item as a gift, or even that they fell asleep and the TV continued playing. We can obviously assume that an item with many accesses from a user implies positive feedback, but this assumption is not as nearly as strong for items with few or no accesses. In a sense, the values in these data sets indicate confidence in user preferences towards items.

Finally, evaluating implicit feedback models and more generally recommender systems is not as clear cut as it is in regression or binary classification models. RMSE is one option, but the task is not necessarily accurately predicting user ratings. Instead, the goal of these models is to identify items users are likely to click on, buy, listen to, etc. In this regard, a more sensible metric of success should account for the precision of our recommendations. In other words, are users acting on our recommendations? We discuss this in greater detail in section 3.

C. Matrix Factorization

Collaborative filtering problems can be defined in a variety of ways, but generally speaking they are either neighborhood-based models or matrix factorization/latent factor models. Neighborhood models usually define some sort of metric of similarity between users or items and then make predictions for users based on items that similar users have listened to or items that are similar to the ones they have already enjoyed. Alternatively, latent factor models attempt to define the abstract underlying features that determine user-item interactions. Many of of these models are carried out through low-rank approximations of the original user-item interaction matrix R. Often Singular Value Decomposition

[1] All code, and tables are included at the very end of this document (after the references)

is used to solve matrix approximation problems but due to the sparsity of implicit and explicit feedback datasets this is not feasible, hence the need for alternative algorithms like Weighted Alternating Least Squares (W-ALS).

II. Related Work

A majority of the work done in this paper is inspired by [5], the originators of the W-ALS algorithm for implicit feedback data sets. This paper is considered one of the seminal works of recommender systems with these type of data and much work has come about from their original paper. Hidasi and Tikk focus on improving W-ALS by proposing a methodology for initializing feature vectors based on similarity between users and items [4]. Likewise, He et. al design a technique for weighting missing data and also an algorithm they claim is significantly faster than Hu et. al [3]. Conjugate gradient descent has also been implemented successfully in solving this problem [11].

Others have adopted Probabilistic Matrix Factorization (PMF) approaches to solve implicit feedback problems. Before Yifan, Salakhutdinov and Mnih introduced their PMF model and showed an 8% improvement compared to the Netflix recommendation system [9]. More recently, Christopher Johnson presented an algorithm used by Spotify called Logistic Matrix Factorization that can model the probability that a user will prefer a specific item [6].

Many practitioners have also made contributions to this area, writing blog posts and creating open-source libraries for end-users outside of the industry. Ben Frederickson has been particularly active in this regard and has written two very detailed posts on Matrix Factorization for music recommendation systems. In addition, he's written one of the most accessible and efficient libraries for collaborative filtering for implicit datasets in Python [1]. Likewise, Dmitriy Selivanov has created an R package for statistical learning on sparse data with a particular focus on applications for recommender systems [10]. Finally, Will Kirwin from Activision Game Science wrote a lengthy post in 2016 discussing an implementation of W-ALS that includes user and item biases [7].

III. Problem Definition

We define $R = (r_{ui}) \in \mathbb{R}^{mxn}$ to be the user-item interaction matrix. Each entry r_{ui} specifies the rating of item i by user u and can be thought of as a single observation. There are m users rating n items. In explicit feedback settings the goal is usually to decompose R into two lower-rank matrices representing latent user and item factors. More formally, the goal is to find a vector $x_u \in \mathbb{R}^f$ for each user u and one for each item i, $y_i \in \mathbb{R}^f$, where f represents the number of latent features we want to estimate. The model is then:

$$\min_{x,y} \sum_{u,i} (r_{ui} - x_u^T y_i)^2 + \lambda(|x_u|^2 + |y_i|^2)$$

A. Preference and Confidence

Because the direct ratings r_{ui} are not observed, this baseline model needs to be adjusted to account for the fact that our observations reflect our confidence in a user's attitudes towards a particular item. For this we introduce two new sets of variables p_{ui} and c_{ui} that make the distinction between a user's perceived preference and our confidence in that guess, respectively. The p_{ui} variables are simply binary values for having detected some sort of user interaction with a particular item. That is,

$$p_{ui} = \begin{cases} 1 & r_{ui} > 0 \\ 0 & r_{ui} = 0 \end{cases} \tag{1}$$

We have some flexibility in deciding how to define the our confidence in a particular observation. Obviously, our confidence should be increasing in r_{ui}, but the rate at which our confidence increases is up to us. Yifan et al. give two recommendations in making this choice [5]:

$$c_{ui} = 1 + \alpha r_{ui}, \tag{2}$$

$$c_{ui} = 1 + \alpha \log(1 + \frac{r_{ui}}{\epsilon}) \tag{3}$$

No matter the choice, cross-validation is used to determine appropriate values for α and ϵ.

B. Loss Function

Now that we have accounted for the characteristics of implicit feedback dataset, we can formalize the model. Just to reiterate, our goal is to to find two matrices X and Y with dimensions $f \times m$, $f \times n$ such that $X^T Y \approx R$. We refer to the columns of these matrices as the latent user-factors and item-factors respectively. The loss function we use to calculate these matrices is thus:

$$\min_{x,y} \sum_{u,i} c_{ui}(p_{ui} - x_u^T y_i)^2 + \lambda(\sum_u |x_u|^2 + \sum_i |y_i|^2) \tag{4}$$

The second half of the equation is for $L2$ regularization of the model to prevent overfitting. As with most regularization methods, λ is determined by cross validation.

The solutions for X, Y are solved iteratively, holding one constant while solving for the other. When we fix Y constant the loss function is quadratic in X, allowing us to find a global minimum. Likewise, holding X fixed allows us to solve for Y. This is where the name "alternating least squares" comes from. At each iteration of this algorithm, the solutions for x_u and y_i (depending on which is being solved for) are found by the following equations:

$$x_u = (Y^T C^u Y + \lambda I)^{-1} Y^T C^u p(u) \tag{5}$$

$$y_i = (X^T C^i X + \lambda I)^{-1} X^T C^i p(i) \tag{6}$$

For each user u, the diagonal $n \times n$ matrix C^u represents the confidence values for each item i. Additionally, the vector $p(u)$ contains all the preferences of u (the p_{ui} values).

Similarly, C^i contains the confidences of each user's rating for item i and the vector $p(i)$ contains all the preferences for that item.

C. Making and Evaluating Recommendations

Once the user and item factors have been found we can predict an individual's preference toward any item by calculating $\hat{p}_{ui} = \hat{x}_u^T \hat{y}_i$. Recommendations are then made by selecting the top K items for which $\hat{p}_{ui}$ is greatest.

This prediction methodology lends itself to the Mean Average Precision (MAP) and Normal Discounted Cumulative Gain (NDCG) evaluation metrics. MAP and NDCG are computed after the practitioner decides how many items to recommend (K) and so these metrics are usually referred to as MAP@K and NDCG@K. Both metrics have values between 0 and 1, allowing us to compare performance across models with different hyperparameters.

MAP@K measures the precision of K recommendations using binary relevance - whether or not the item is relevant to the user. By classifying each item recommendation as 1 or 0 we can measure the average precision of a recommendation to a particular user by averaging the 1s divided by their ranking in the list. MAP then averages that value across all users. For example, if a user is recommended five items and the first two and last two are relevant, we have the sequence $\{1, 1, 0, 1, 1\}$. Relative to their ranking in this list, that sequence becomes $\{\frac{1}{1}, \frac{1}{2}, 0, \frac{3}{4}, \frac{4}{5}\}$, and averaging the non-zero values gives an average precision score of .7625.

NDCG@K is slightly more complex in that it allows for real-valued item relevances and discounting for items occurring later in the list, but the general idea is the same.

IV. Experimental Study

A. Data Description

This analysis will be done using the publicly available Last.fm Music Recommendation Dataset. This data contains (user, artist, plays) tuples for approximately 360,000 users and 186,000 artists collected from the Last.fm API. With this data our task will be to recommend artists to individual users, and entries in our user-item interaction matrix r_{ui} will be the number of times a user u has listened to artist i. Although we have around 17.5 million (user, artist, plays) observations, R will still be approximately 99.97% sparse.

B. Training and Tuning

Implicit feedback datasets often have some sort of time dimension separating two or more periods of observed activity. This is incredibly useful as it creates a natural testing dataset on which to evaluate models. Unfortunately, this dataset lacks this so we have to artificially create our own test set from the original. Briefly put, we select 30,000 users to be part of our test set and of the 30,000 users we randomly select 50% of their non-zero observations to use as a historical/future split. The validation strategy is as follows:

- Calculate user and item factor vectors using the training set ($\approx$ 330,000 users)
- Set solved item-factor vector fixed, and calculate user-factor for historic testing set ($\approx$ 30,000 users)
- Predict using newly calculated user-factors and evaluate using user-factor future testing set ($\approx$ 30,000 users)

Because of the massive computation time it is infeasible to use k-fold cross-validation to tune our hyperparameters. Instead, we will simply reuse the same training and testing set for each possible combination of hyperparameter values. The final list of parameters we test are listed below:

- f - the number of user and item factors
- λ - regularization penality
- α - tuning parameter of c_{ui}

V. Results

As stated in the previous section, computation time was a serious concern for estimating these models. In order to keep overall time down, we tried keeping our parameter grid relatively small. In fact, we choose not to test the model using the logged confidence function. This allows us to eliminate an additional two parameters, keeping it at (just) three. Furthermore, we didn't feel like waiting a week for a single set of results so for each hyperparameter we only tested three possible values, giving us 27 total combinations. Because this search was rather small we tried to capture both ends of the spectrum for each hyperparameter - few/many factors, weak/strong regularization, and slow/fast rate of increase in confidence.

We chose to evaluate each model using the MAP@10 and NDCG@10 metrics described earlier, where $K = 10$. Both metrics measure roughly the same thing so we keep the discussion to only MAP scores for ease of interpretation. So, for each user in our "historic" test set we recommended 10 artists they had not listened to before and we evaluated the model on whether or not the user listened to those recommendations in the "future" time period. Results were highly variable - some were dreadfully low and some performed remarkably well. On the low end, there were a handful of models with scores around .10, roughly equating to getting every 10th prediction correct. In contrast, our best performing model recieved a score of .79. When all was said and done, that model had 128 factors ($f = 128$), strong regularization ($\lambda = 100$), and a slow rate of increase in confidence.

These 'optimal' hyperparameter values are relatively intuitive - the higher the rank of the two decomposed matrices, the more they approximate the original user-item interaction matrix R. However, the closer we approximate the original matrix, the more likely our model is to overfit the data, hence the large regularization penalty. Finally, recall that our data was only on user-artist plays. It's not unimaginable for a single user to rack up a high play count for a single artist and still remain only somewhat interested in the artist. It could be the case that they simply wanted to listen to an entire album before making a decision, for instance. On the other hand, if we had observed that a user listened to a particular song many times we would probably have a lot more confidence in guessing that they like the song. In short, the model resulted

in better performance when we raised the bar for what we considered actual user desire.

VI. Discussion

This paper was meant to be a simple exploration into music recommendation systems with the commonly used W-ALS technique. This was not meant to be a paper comparing efficacy of various recommendation system methodologies, but rather to see how well W-ALS could work on a unique data set. There were several problems we ran into during the process, some of which were easily resolved while some had to be left untouched. Specifically, the fact that this data set does not contain any sort of time dimension was problematic because a validation process had to be constructed on the fly. This choice may have introduced some unwanted bias in our results, skewing our interpretation of how well the model actually performs on test data. However, seeing as there was no alternative and the size of the dataset was still extremely large we do not believe this was significant enough of a problem to completely invalidate the results. A second issue was the amount of time each model took to train. We tried decreasing the tolerance for convergence as well as setting a maximum amount of iterations, but still each model would run for at least 20 minutes before convergence. This made hyperparameter tuning extremely difficult as any parameter grid we were interested in testing would take at least several hours. In the end, we chose to use only one of the confidence functions that could have been used, saving us a significant amount of time.

Overall, the most obvious thing to improve these models is to do more robust testing. We missed out entirely on the additional measure of confidence and we barely scratched the surface of the potential hyperparameter space. In any case, our models still perform well and we are content with their performance.

References

[1] Ben Frederickson. www.benfrederickson.com/.

[2] Decision Fatigue. Wikipedia, Wikimedia Foundation, 28 Apr. 2018, en.wikipedia.org/wiki/Decision_fatigue.

[3] He, Xiangnan, et al. Fast Matrix Factorization for Online Recommendation with Implicit Feedback. Proceedings of the 39th International ACM SIGIR Conference on Research and Development in Information Retrieval - SIGIR 16, 2016, doi:10.1145/2911451.2911489.

[4] Hidasi, Balzs, and Domonkos Tikk. Enhancing Matrix Factorization through Initialization for Implicit Feedback Databases. Proceedings of the 2nd Workshop on Context-Awareness in Retrieval and Recommendation - CaRR 12, 2012, doi:10.1145/2162102.2162104.

[5] Hu, Yifan, et al. Collaborative Filtering for Implicit Feedback Datasets. 2008 Eighth IEEE International Conference on Data Mining, 2008, doi:10.1109/icdm.2008.22.

[6] Johnson, Christopher C.. Logistic Matrix Factorization for Implicit Feedback Data. (2014).

[7] Kirwin, Will. Implicit Recommender Systems: Biased Matrix Factorization. Activision Game Science, activisiongamescience.github.io/2016/01/11/Implicit-Recommender-Systems-Biased-Matrix-Factorization/.

[8] Koren, Yehuda. The BellKor Solution to the Netflix Grand Prize. Yahoo! Research Labs, Aug. 2009, www.netflixprize.com/assets/GrandPrize2009_BPC_BellKor.pdf.

[9] Salakhutdinov, Ruslan, and Andriy Mnih. Bayesian Probabilistic Matrix Factorization Using Markov Chain Monte Carlo. Proceedings of the 25th International Conference on Machine Learning - ICML 08, 2008, doi:10.1145/1390156.1390267.

[10] Selivanov, Dmitriy. Rsparse Package. GitHub, github.com/dselivanov/rsparse/.

[11] Takcs, Gbor, et al. Applications of the Conjugate Gradient Method for Implicit Feedback Collaborative Filtering. Proceedings of the Fifth ACM Conference on Recommender Systems - RecSys 11, 2011, doi:10.1145/2043932.2043987.

[12] Zhou, Yunhong, et al. Large-Scale Parallel Collaborative Filtering for the Netflix Prize. Algorithmic Aspects in Information and Management Lecture Notes in Computer Science, 2008, doi:10.1007/978-3-540-68880-8_32.

Weighted Alternating Least Squares with the LastFM Dataset

William Morgan

04 May, 2018

Load data

```
# Load data and add names
raw_data <-  fread("Data/lastfm-dataset-360K/usersha1-artmbid-artname-plays.tsv", showProgress = F)

names(raw_data) <- c("user_id", "artist_id", "artist_name", "number_plays")

raw_data <- raw_data %>%
  filter(str_length(artist_id) > 10)
```

Tidy data

```
# Use integer-valued ids for users and items
user_encoding <-  raw_data %>%
  distinct(user_id) %>%
  mutate(uid = row_number())

item_encoding <- raw_data %>%
  distinct(artist_id, artist_name) %>%
  mutate(iid = row_number())

data <-  raw_data %>%
  inner_join(user_encoding, by = 'user_id') %>%
  inner_join(item_encoding, by = 'artist_id')

rm(raw_data)
```

Split data

- We use 30K listeners in our test set
- For each listener in the testing set, we need to randomly split their listens into history and future listens (this is done for testing)

```
# Define our model matrix
X = sparseMatrix(i = data$uid, j = data$iid, x = data$number_plays,
                 dimnames = list(user_encoding$user_id, item_encoding$artist_name))

n_test <- 30000L
test_uid <- sample(nrow(user_encoding), n_test)
```

```
X_train <-  X[-test_uid, ]
X_test <-  X[test_uid, ]

# Split our test set into "history" or "future"
temp = as(X_test, "TsparseMatrix")
temp = data.table(i = temp@i, j = temp@j, x = temp@x)

temp <- temp %>%
  group_by(i) %>%                          # group by user
  mutate(ct = length(j),                   # number of artists each user has
         history =
           sample(c(TRUE, FALSE), ct, replace = TRUE, prob = c(.5, .5))) %>%
  select(-ct)

X_test_history <- temp %>% filter(history == TRUE)
X_test_future <- temp %>% filter(history == FALSE)

rm(temp)

# Convert them back to sparse matrices
X_test_history <- sparseMatrix(i = X_test_history$i,
                               j = X_test_history$j,
                               x = X_test_history$x,
                               dims = dim(X_test),
                               dimnames = dimnames(X_test),
                               index1 = FALSE)

X_test_future <- sparseMatrix(i = X_test_future$i,
                              j = X_test_future$j,
                              x = X_test_future$x,
                              dims = dim(X_test),
                              dimnames = dimnames(X_test),
                              index1 = FALSE)

rm(user_encoding, item_encoding, n_test, test_uid, data)
```

Confidence Measures

Recall our loss function:

$$L = \sum_u \sum_i c_{ui}(p_{ui} - x_u^T y_i)^2 + \lambda(\|X\|^2 + \|Y\|^2)$$

We need to define two functions that will allow us to create the confidence matrix:

$$c_{ui} = 1 + \alpha log(1 + \frac{r_{ui}}{\epsilon})$$
$$c_{ui} = 1 + \alpha log(1 + r_{ui})$$

α and ϵ are hyperparameters that should be tuned using CV

```
# Define confidence functions and create matrices
log_conf <-  function(x, alpha, epsilon){
  x_confidence <-  x
  stopifnot(inherits(x, "sparseMatrix"))
  x_confidence@x = 1 + alpha * log(1 + (x@x / epsilon))
  return(x_confidence)
}

lin_conf <- function(x, alpha) {
  x_confidence <- x
  stopifnot(inherits(x, "sparseMatrix"))
  x_confidence@x = 1 + alpha * x@x
  return(x_confidence)
}
```

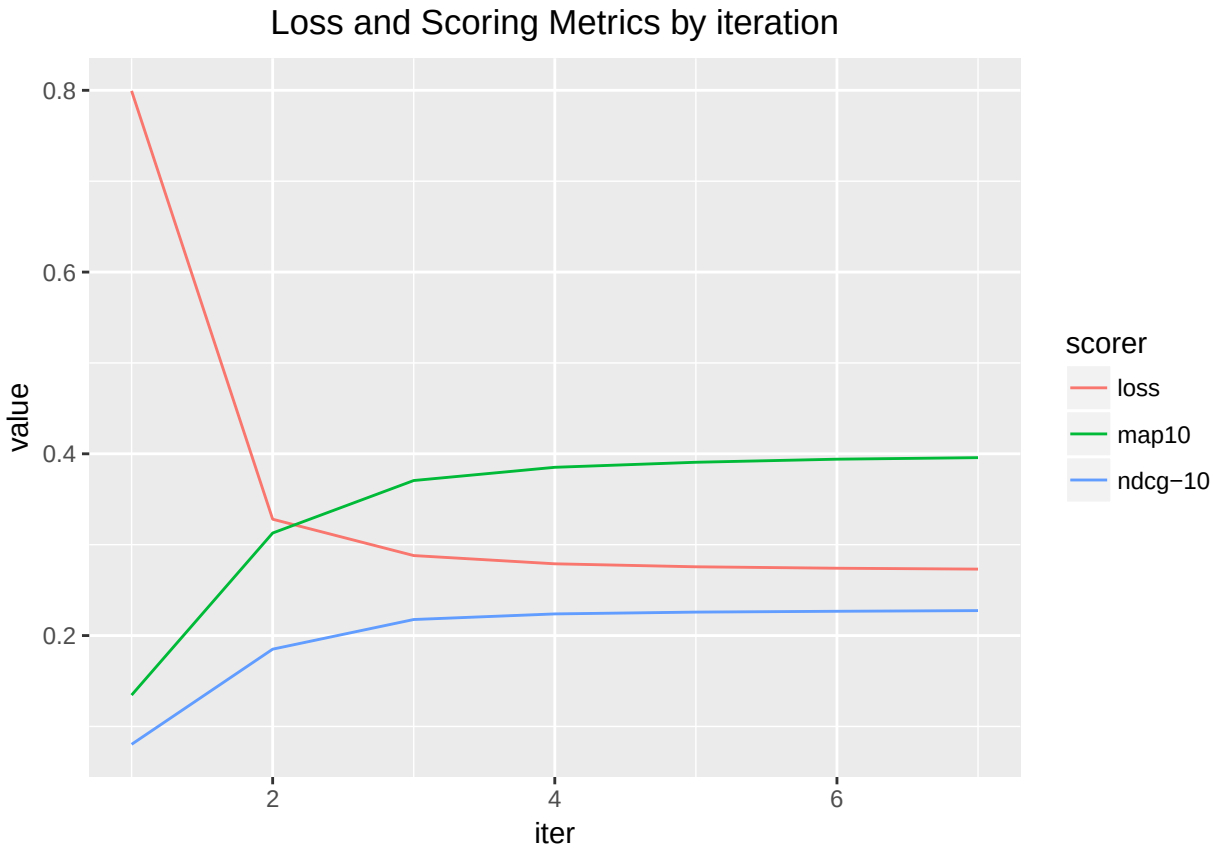
Practice Model

```
# Define hyperparameters
alpha <- .1
lambda <- 10
components <- 10L

# Create Confidence Matrices
X_train_conf <- lin_conf(X_train, alpha)
X_test_history_conf <- lin_conf(X_test_history, alpha)

# Initialize a model
model <- WRMF$new(rank = components,
                  lambda = lambda,
                  feedback = 'implicit',
                  solver = 'conjugate_gradient')

# Add scoring metrics
model$add_scorers(x_train = X_test_history_conf, x_cv = X_test_future,
                  list("map10" = "map@10", "ndcg-10" = "ndcg@10"))

# Calculate user factors
train_user_factors <- model$fit_transform(X_train_conf)
```

```
## INFO [2018-05-04 16:23:28] starting factorization with 12 threads
## INFO [2018-05-04 16:23:59] iter 1 loss = 0.7994  score ndcg-10 = 0.080304 score map10 = 0.134484
## INFO [2018-05-04 16:24:30] iter 2 loss = 0.3280  score ndcg-10 = 0.185036 score map10 = 0.312825
## INFO [2018-05-04 16:25:00] iter 3 loss = 0.2880  score ndcg-10 = 0.217662 score map10 = 0.370606
## INFO [2018-05-04 16:25:31] iter 4 loss = 0.2790  score ndcg-10 = 0.223829 score map10 = 0.385161
## INFO [2018-05-04 16:26:01] iter 5 loss = 0.2757  score ndcg-10 = 0.225811 score map10 = 0.390712
## INFO [2018-05-04 16:26:32] iter 6 loss = 0.2741  score ndcg-10 = 0.226737 score map10 = 0.394074
## INFO [2018-05-04 16:27:02] iter 7 loss = 0.2731  score ndcg-10 = 0.227460 score map10 = 0.395774
## INFO [2018-05-04 16:27:02] Converged after 7 iterations
```

```
# Make ALS step given fixed items matrix, and then predict for those users top K items
test_predictions <- model$predict(X_test_history_conf, k = 10)

trace = attr(train_user_factors, "trace")
```

```
ggplot(trace) +
  geom_line(aes(x = iter, y = value, col = scorer)) +
  labs(title = "Loss and Scoring Metrics by iteration") +
  theme(plot.title = element_text(hjust = .5))
```



Tune Model - Linear Confidence

```
# Convergence parameters
n_iter_max = 10L
convergence_tol = .01

# Hyperparameters to test
grid = expand.grid(alpha = c(.01, 1),
                   rank = c(16, 64, 128),
                   lambda = c(100))

# Empty vector to throw results into
lin_scores <-  vector("list", nrow(grid))

for (k in seq_len(nrow(grid))){
  # Define parameters
  alpha = grid$alpha[[k]]
  rank = grid$rank[[k]]
  lambda = grid$lambda[[k]]
```

```
  # Initialize
  model <- WRMF$new(rank = rank,
                    lambda = lambda,
                    feedback = 'implicit',
                    solver = 'conjugate_gradient')

  # Conf. matrices
  X_train_conf<- lin_conf(X_train, alpha)
  X_test_history_conf <- lin_conf(X_test_history, alpha)


  # Scoring metrics
  model$add_scorers(x_train = X_test_history_conf,
                    x_cv = X_test_future,
                    list("map10" = "map@10", "ndcg-10" = "ndcg@10"))

  # Fit
  fit <-  model$fit_transform(X_train_conf, n_iter = n_iter_max,
                              convergence_tol = convergence_tol)

  # Extract score
  score <-  attr(fit, "trace")

  score$alpha = alpha
  score$lambda = lambda
  score$rank = rank

  # Add to list
  lin_scores[[k]] <-  score

  # Clean up
  rm(alpha, rank, lambda, model, score)
}

lin_results <-  bind_rows(lin_scores) %>%
  group_by(alpha, lambda, rank, scorer) %>%
  arrange(iter) %>%
  filter(row_number() == n()) %>%
  select(-iter) %>%
  ungroup()

fwrite(lin_results, "Data/linear confidence results.csv")
```

```
lin_results <- fread("Data/linear confidence results.csv")

print("Best Performing Models by MAP@10")
```

```
## [1] "Best Performing Models by MAP@10"
```

```
lin_results %>%
  filter(scorer == "map10") %>%
  arrange(desc(value))
```

```
##   scorer     value alpha lambda rank
```

```
## 1  map10 0.7867390  0.01    100  128
## 2  map10 0.6978280  0.01    100   64
## 3  map10 0.5081508  1.00    100  128
## 4  map10 0.4799475  0.01    100   16
## 5  map10 0.4580219  1.00    100   64
## 6  map10 0.3538391  1.00    100   16
```

```
print("Best Performing Models by NDCG@10")
```

```
## [1] "Best Performing Models by NDCG@10"
```

```
lin_results %>%
  filter(scorer == "ndcg-10") %>%
  arrange(desc(value))
```

```
##    scorer     value alpha lambda rank
## 1 ndcg-10 0.4295828  0.01    100  128
## 2 ndcg-10 0.3738932  0.01    100   64
## 3 ndcg-10 0.3189858  1.00    100  128
## 4 ndcg-10 0.2780850  1.00    100   64
## 5 ndcg-10 0.2516068  0.01    100   16
## 6 ndcg-10 0.2029855  1.00    100   16
```

```
# Convergence parameters
n_iter_max = 10L
convergence_tol = .01

# Hyperparameters to test
grid = expand.grid(alpha = c(.01, 1, 100),
                   rank = c(16, 64, 128),
                   lambda = c(.01, 1, 100),
                   epsilon = c(.01, 1, 100))

# Empty vector to throw results into
log_scores <-  vector("list", nrow(grid))

for (k in seq_len(nrow(grid))){
  # Define parameters
  alpha = grid$alpha[[k]]
  rank = grid$rank[[k]]
  lambda = grid$lambda[[k]]
  epsilon = grid$epsilon[[k]]


  # Initialize
  model <- WRMF$new(rank = rank,
                    lambda = lambda,
                    feedback = 'implicit',
                    solver = 'conjugate_gradient')

  # Conf. matrices
  X_train_conf <- log_conf(X_train, alpha, epsilon)
  X_test_history_conf <- log_conf(X_test_history, alpha, epsilon)

  # Scoring metrics
  model$add_scorers(x_train = X_test_history_conf,
```

```
                    x_cv = X_test_future,
                    list("map10" = "map@10", "ndcg-10" = "ndcg@10"))

  # Fit
  fit <-  model$fit_transform(X_train_conf, n_iter = n_iter_max,
                              convergence_tol = convergence_tol)

  # Extract score
  score <-  attr(fit, "trace")

  score$alpha = alpha
  score$lambda = lambda
  score$rank = rank
  score$epsilon = epsilon

  # Add to list
  scores[[k]] <-  score

  # Clean up
  rm(alpha, rank, lambda, model, score, epsilon)
}
```