Project

Mark Cunningham, Ethan Saloom, Branson Stickney

07 July, 2022

Contents

Data Preprocessing

Library Imports

Dataframe Imports

```
anime_df <- read.csv("anime.csv")
ratings_df <- read.csv("rating_complete.csv")
```

Filtering out explicit content and Subsetting Irrelevant Columns

```
# Filtering out explicit anime from anime_df
# Filtering out anime with no English names, or less than 1000 complete watches
anime_df <- anime_df %>% filter(!(grepl("Hentai",Genres)|Score=="Unknown"|English.name=="Unknown") & ani

# Updating ratings dataframe with anime datframe
ratings_df <- ratings_df[(ratings_df$anime_id %in% anime_df$MAL_ID),]

# Remove unused columns
anime_df <- subset(anime_df, select=-c(Japanese.name,Producers,Licensors,Favorites,Watching,On.Hold,Plan
```

Previewing the Data

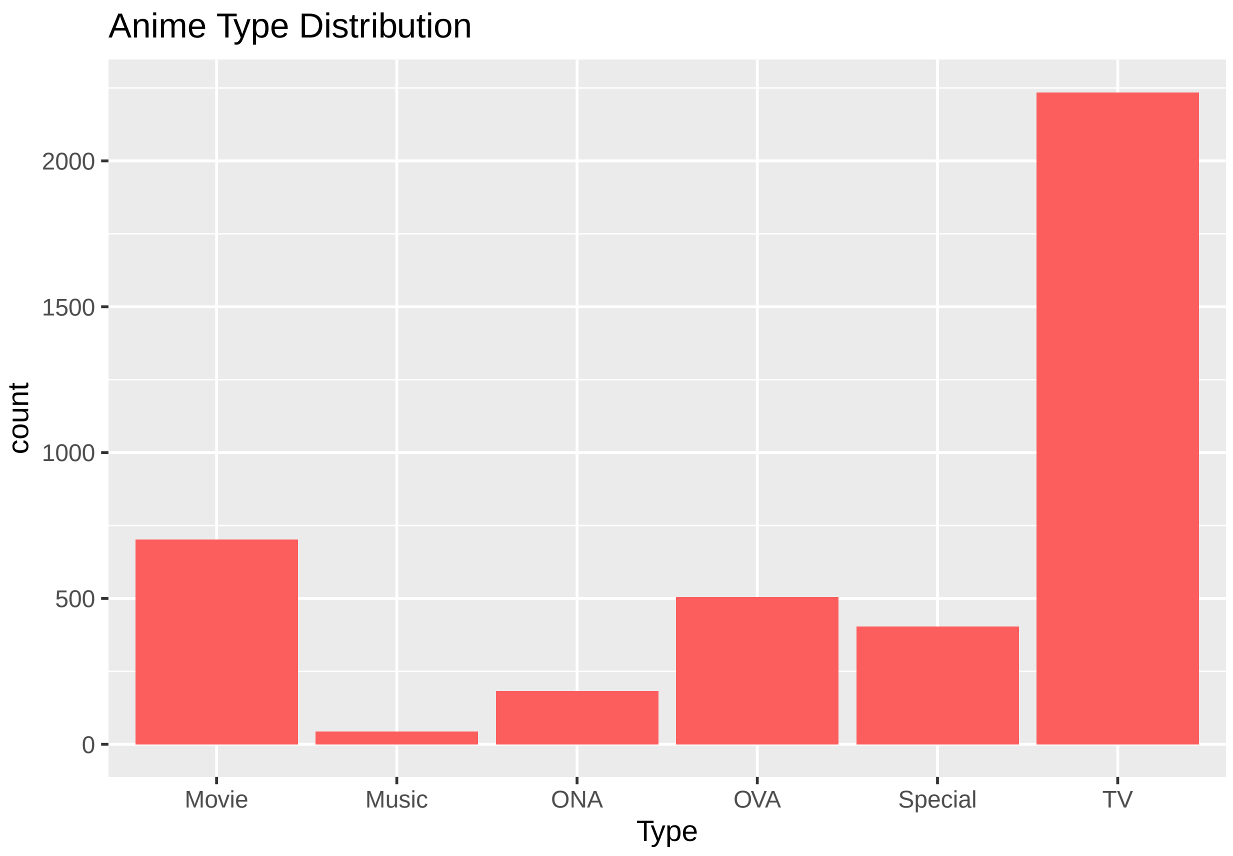
```
glimpse(head(anime_df))
```

```
## Rows: 6
## Columns: 18
## $ MAL_ID       <int> 1, 5, 6, 7, 8, 16
## $ Name         <chr> "Cowboy Bebop", "Cowboy Bebop: Tengoku no Tobira", "Trigu~
## $ Score        <chr> "8.78", "8.39", "8.24", "7.27", "6.98", "8.06"
## $ Genres       <chr> "Action, Adventure, Comedy, Drama, Sci-Fi, Space", "Actio~
## $ English.name <chr> "Cowboy Bebop", "Cowboy Bebop:The Movie", "Trigun", "Witc~
## $ Type         <chr> "TV", "Movie", "TV", "TV", "TV", "TV"
## $ Episodes     <chr> "26", "1", "26", "26", "52", "24"
## $ Aired        <chr> "Apr 3, 1998 to Apr 24, 1999", "Sep 1, 2001", "Apr 1, 199~
## $ Premiered    <chr> "Spring 1998", "Unknown", "Spring 1998", "Summer 2002", "~
## $ Studios      <chr> "Sunrise", "Bones", "Madhouse", "Sunrise", "Toei Animatio~
## $ Source       <chr> "Original", "Original", "Manga", "Original", "Manga", "Ma~
## $ Duration     <chr> "24 min. per ep.", "1 hr. 55 min.", "24 min. per ep.", "2~
## $ Rating       <chr> "R - 17+ (violence & profanity)", "R - 17+ (violence & pr~
## $ Ranked       <chr> "28.0", "159.0", "266.0", "2481.0", "3710.0", "468.0"
## $ Popularity   <int> 39, 518, 201, 1467, 4369, 687
## $ Members      <int> 1251960, 273145, 558913, 94683, 13224, 214499
## $ Completed    <int> 718161, 208333, 343492, 46165, 7314, 81145
## $ Dropped      <int> 26678, 770, 13925, 5378, 1108, 11026
```

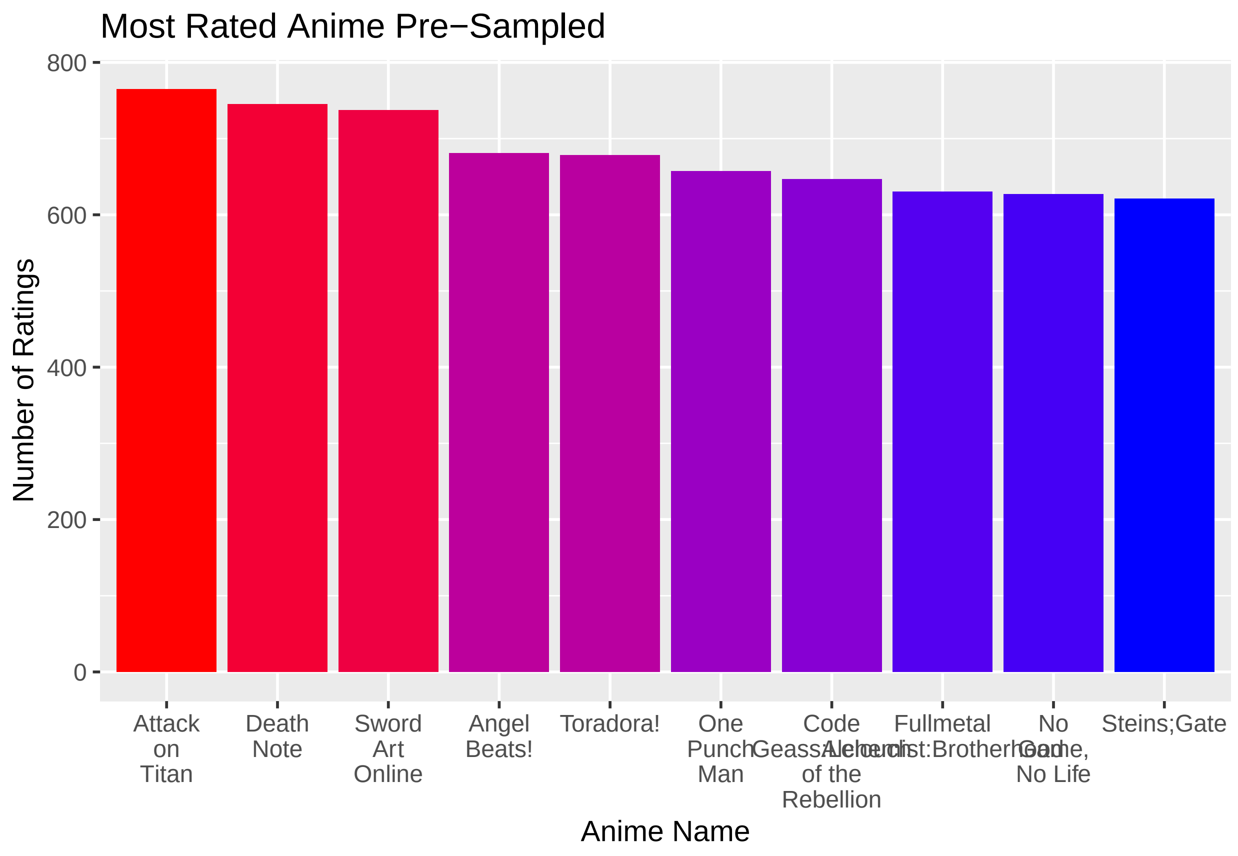
```
glimpse(head(ratings_df))
```

```
## Rows: 6
## Columns: 3
## $ user_id  <int> 0, 0, 0, 0, 0, 0
## $ anime_id <int> 430, 1004, 3010, 570, 431, 578
## $ rating   <int> 9, 5, 7, 7, 8, 10
```

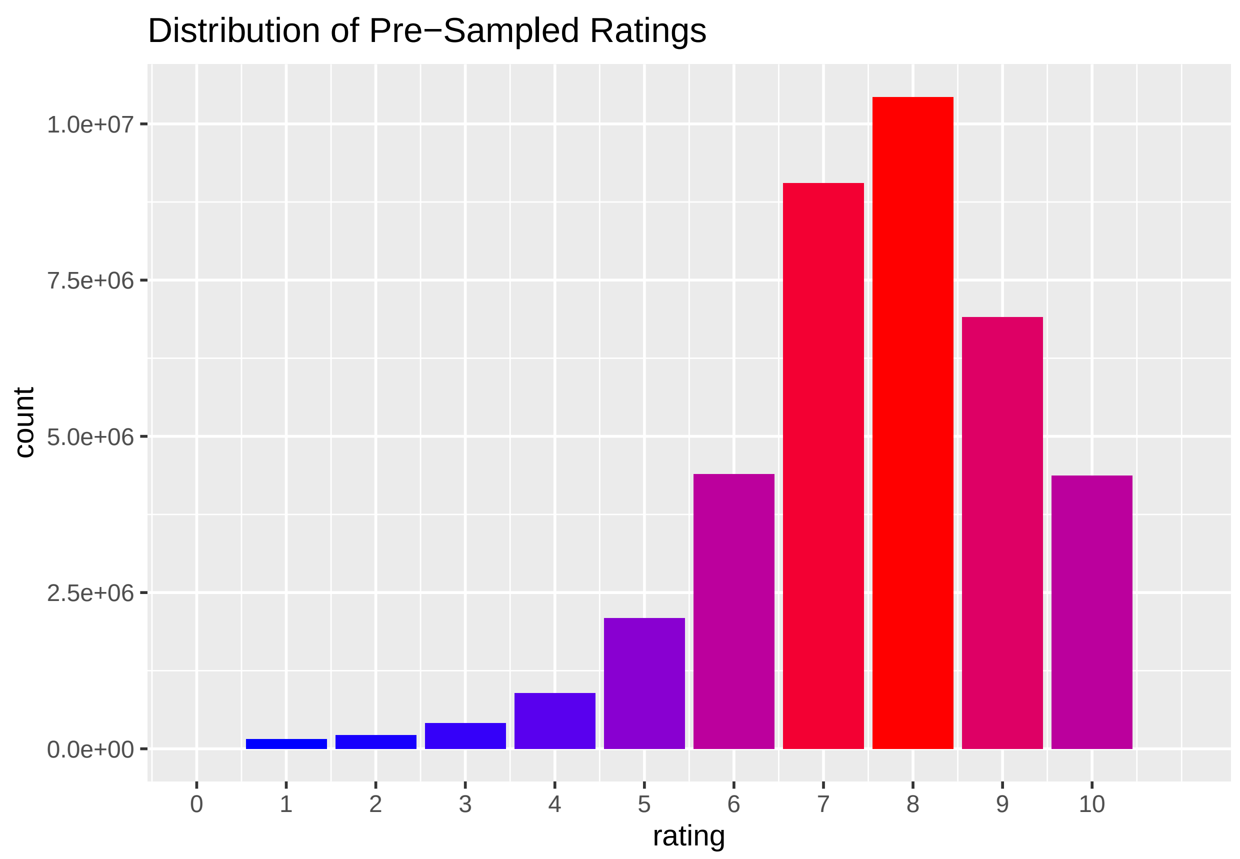
```
anime_df %>%
ggplot(aes(x=Type)) +
geom_bar(stat = "count", fill = "#fc5e5e" ) +
ggtitle("Anime Type Distribution")
```



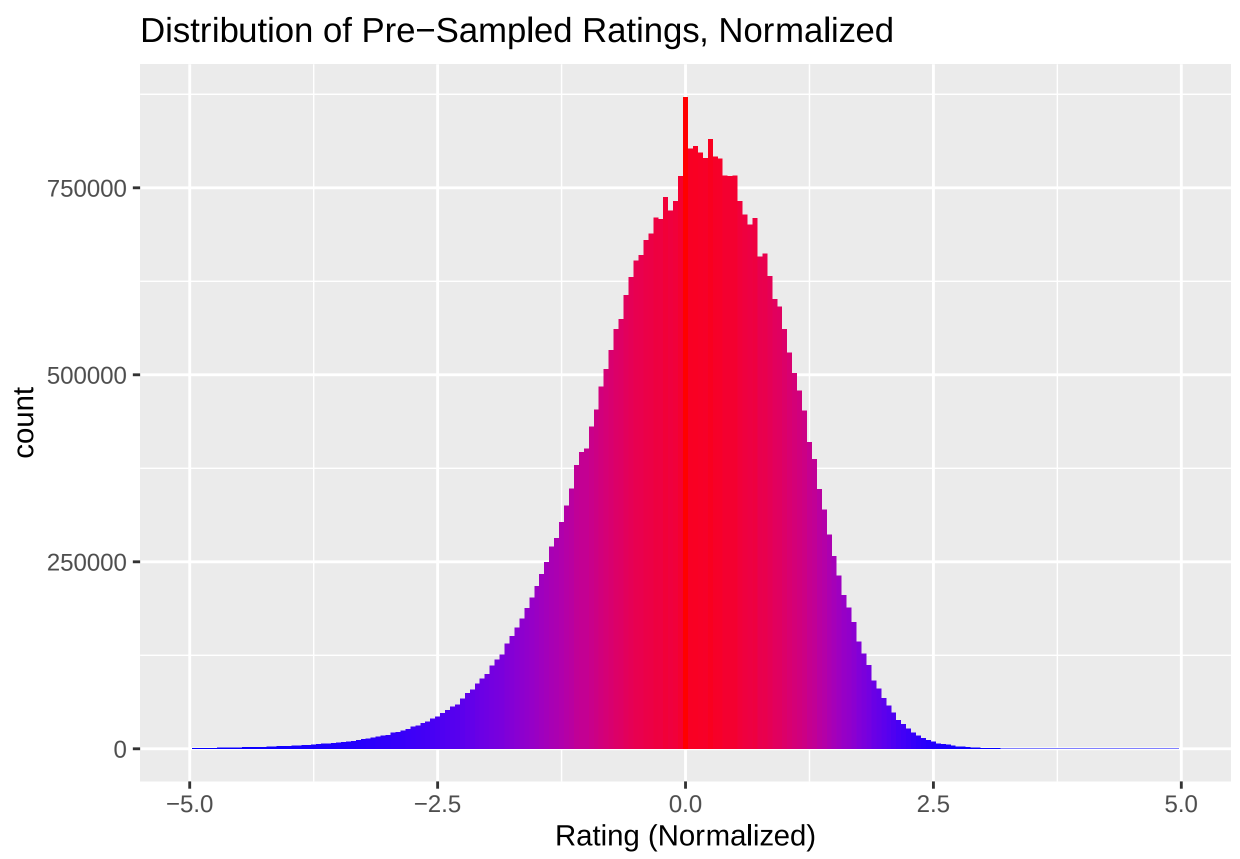
```r
load("pre_sampled_df.rda")
rating_og_counts[1:10,] %>%
  ggplot(aes(x = reorder(Anime_Name, -Num_Ratings),y=Num_Ratings,fill=Num_Ratings)) +
  scale_fill_continuous(low="blue", high="red") +
  scale_x_discrete(labels = scales::wrap_format(8))+
  geom_bar(stat='identity') +
  xlab("Anime Name" )+
  ylab("Number of Ratings" )+
  ggtitle("Most Rated Anime Pre-Sampled") +
  theme(legend.position="none")
```



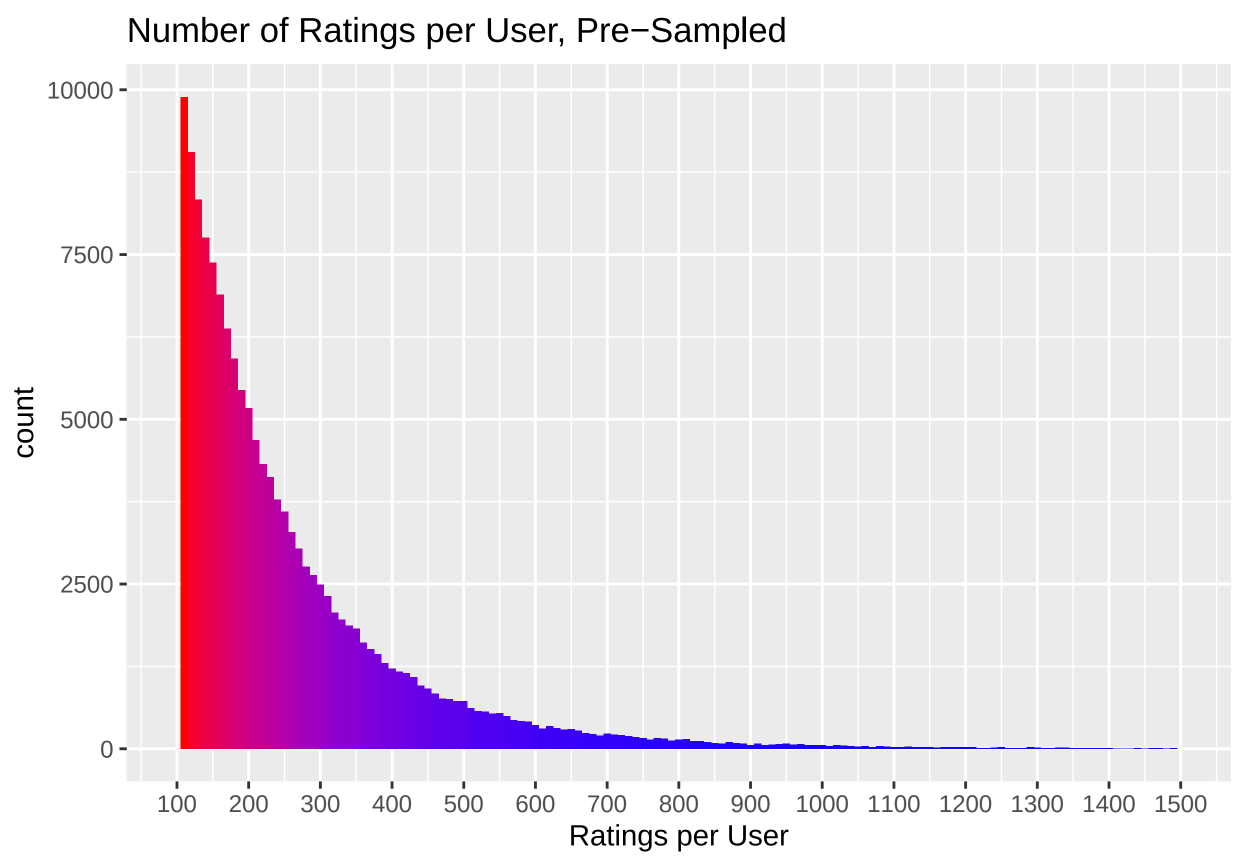
```r
ggplot(data=ratingmat_og_df,aes(x=rating,y=..count..,fill=..count..)) +
  geom_bar() +
  scale_fill_continuous(low="blue", high="red") +
  scale_x_continuous(limits = c(0,11),breaks = seq(-1,10,1)) +
  ggtitle("Distribution of Pre-Sampled Ratings") +
  theme(legend.position="none")
```



```
ggplot(data=ratingmat_df_normalized,aes(x=`getRatings(normalize(ratingmat, method = "Z-score"))`,y=..cou
  geom_histogram(binwidth = 0.05) +
  scale_fill_continuous(low="blue", high="red") +
  xlim(-5,5)+
  xlab("Rating (Normalized)" )+
  ggtitle("Distribution of Pre-Sampled Ratings, Normalized") +
  theme(legend.position="none")
```



```
ggplot(data=ratingmat_og_rowcounts,aes(x=`rowCounts(ratingmat)`,y=..count..,fill=..count..)) +
  geom_histogram(binwidth = 10) +
  scale_fill_continuous(low="blue", high="red") +
  scale_x_continuous(limits = c(100,1500),breaks = seq(0,1500,100)) +
  xlab("Ratings per User") +
  ggtitle("Number of Ratings per User, Pre-Sampled") +
  theme(legend.position="none")
```



Sampling

```
#Unlist data so it can be transformed into realRatingMatrix
ratings_df$user_id <- as.numeric(unlist(ratings_df$user_id))
ratings_df$anime_id <- as.numeric(unlist(ratings_df$anime_id))
ratings_df$rating <- as.numeric(unlist(ratings_df$rating))
ratings_df<-as.data.frame(ratings_df)

set.seed(1234)
#Convert rating matrix into a recommenderlab sparse matrix
ratingmat <- as(ratings_df, "realRatingMatrix")
ratingmat

#Filter for anime that has 1000 or more ratings, and users who have greater than 100 anime reviews
ratingmat <- ratingmat[rowCounts(ratingmat) > 100, colCounts(ratingmat) > 1000]
ratingmat
object.size(ratingmat)

ratingmat_filtered <- as(ratingmat,"data.frame")
```

The size of the rating matrix is still too big for use with Recommenderlab functions, so we will randomly sample 5000 users. This will only compromise very unpopular anime, which is a sacrifice worth making.

```
set.seed(123)
#Sample of rating matrix
ratingmat_sample <- sample(ratingmat, 1000)
ratingmat_sample
```

```
## 1000 x 3140 rating matrix of class 'realRatingMatrix' with 260682 ratings.
```

```
object.size(ratingmat_sample)
```

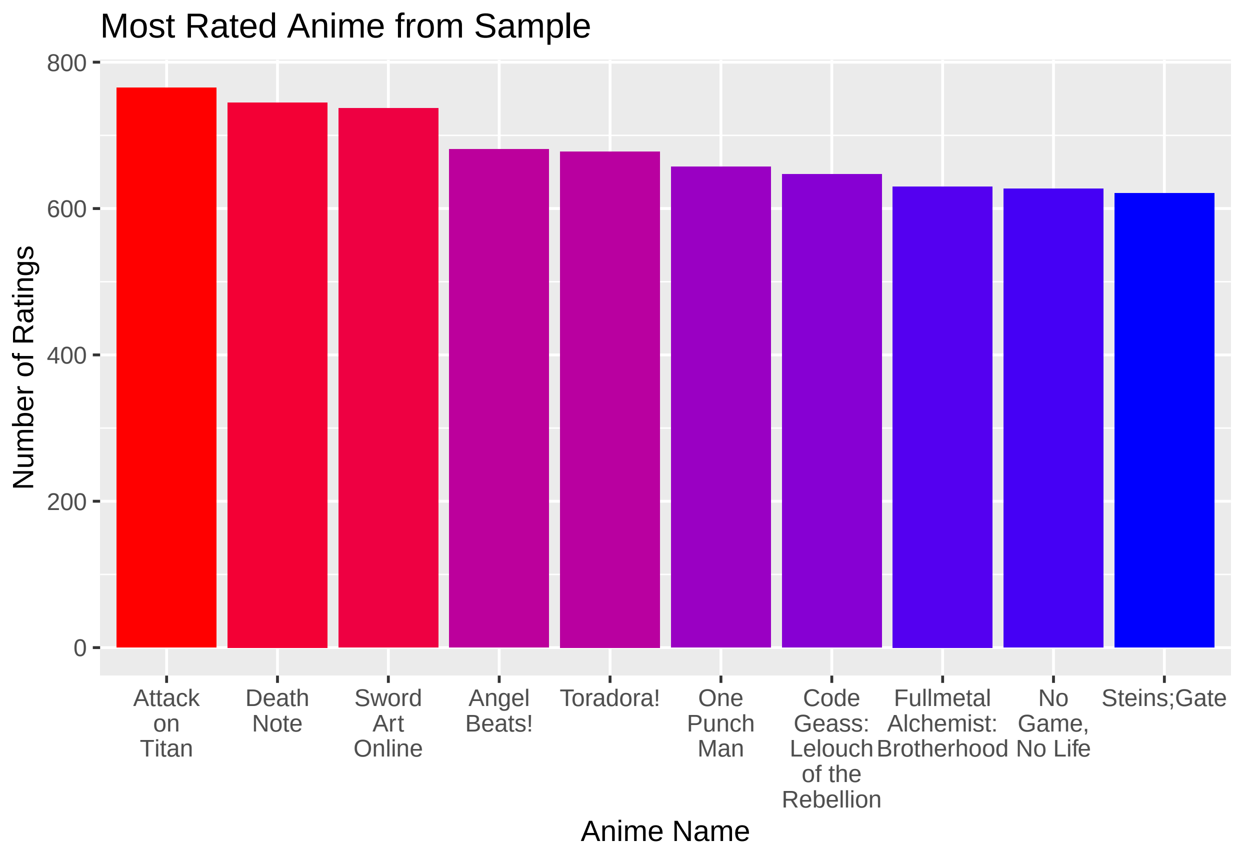
```
## 3408120 bytes
```

```
# Removing anime that aren't part of the sample
ratingmat_sample_filtered <- as(ratingmat_sample,"data.frame")
anime_id_sample_filtered <- ratingmat_sample_filtered[,2]
anime_id_sample_filtered <- anime_id_sample_filtered[!duplicated(anime_id_sample_filtered)]
anime_id_sample_filtered <- as.data.frame(anime_id_sample_filtered)
colnames(anime_id_sample_filtered)[1] <- "item"
anime_id_sample_filtered$item <- as.character(unlist(anime_id_sample_filtered$item))
anime_df_sample <- anime_df[(anime_df$MAL_ID %in% anime_id_sample_filtered$item),]
```

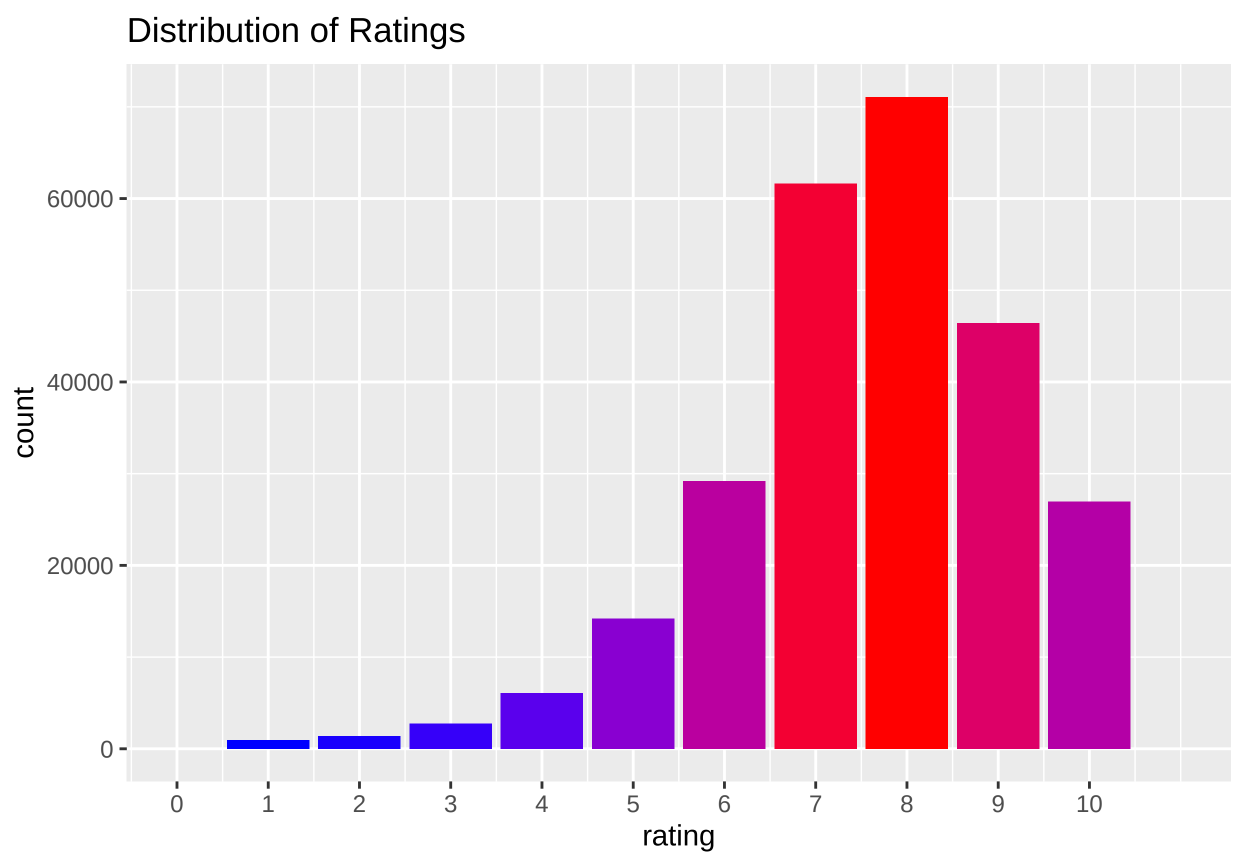
Preview Sampled Rating Distribution

```
ratingmat_sample_df <- as(ratingmat_sample,"data.frame")
ratingmat_sample_df_normalized <- as.data.frame(getRatings(normalize(ratingmat_sample,method="Z-score")
ratingmat_colcounts <- as.data.frame(colCounts(ratingmat_sample))
ratingmat_colcounts[ "MAL_ID" ] <- rownames(ratingmat_colcounts)
ratingmat_colcounts <- ratingmat_colcounts[,c(2,1)]
anime_id_ratingmat_colcounts <- ratingmat_colcounts[,1]
anime_id_ratingmat_colcounts<- as.data.frame(anime_id_ratingmat_colcounts)
ratingmat_rowcounts <- as.data.frame(rowCounts(ratingmat_sample))

getAnimeNames <- function(df) {
  # function that takes as input a list of MAL_ID,
  # and returns a data frame containing the corresponding English names
  out = as.data.frame(matrix(ncol=1,nrow=3140))
  names(out)[1] <- 'Anime Name'
  for(i in 1:nrow(df)) {
    value <- df[[i,1]]
    row <- which(anime_df[["MAL_ID"]]==value)
    out[i,1] <- anime_df[row,5]
  }
  return(out)
}

rating_counts <- getAnimeNames(anime_id_ratingmat_colcounts)
rating_counts <- cbind(rating_counts,ratingmat_colcounts)
names(rating_counts)[3] <- "Num_Ratings"
names(rating_counts)[1] <- "Anime_Name"
rating_counts <- rating_counts %>% arrange(rating_counts$Num_Ratings, decreasing = TRUE)
rating_counts[8,1] <- "Fullmetal Alchemist: Brotherhood"
```

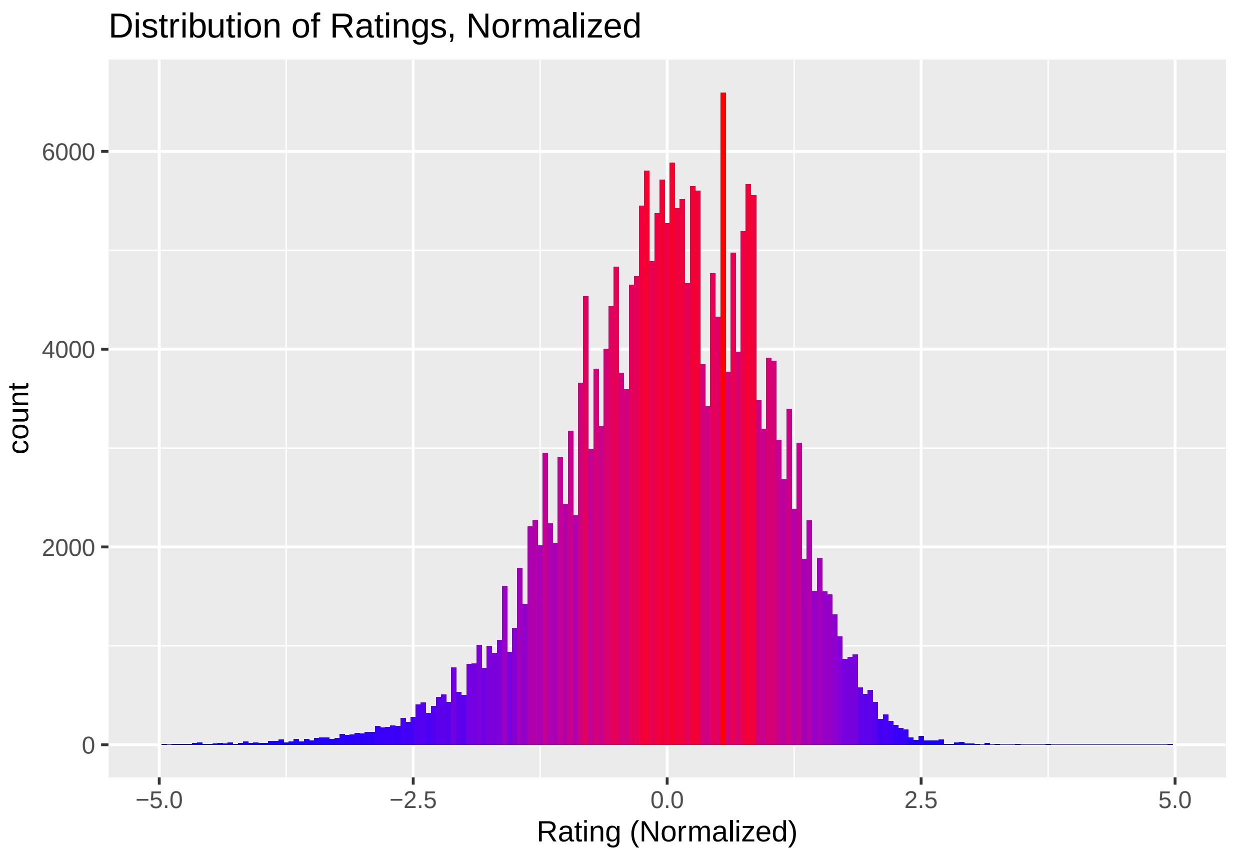
```r
rating_counts[7,1] <- "Code Geass: Lelouch of the Rebellion"

rating_counts[1:10,] %>%
  ggplot(aes(x = reorder(Anime_Name, -Num_Ratings),y=Num_Ratings,fill=Num_Ratings)) +
  scale_fill_continuous(low="blue", high="red") +
  scale_x_discrete(labels = scales::wrap_format(8))+
  geom_bar(stat='identity') +
  xlab("Anime Name" )+
  ylab("Number of Ratings" )+
  ggtitle("Most Rated Anime from Sample") +
  theme(legend.position="none")
```



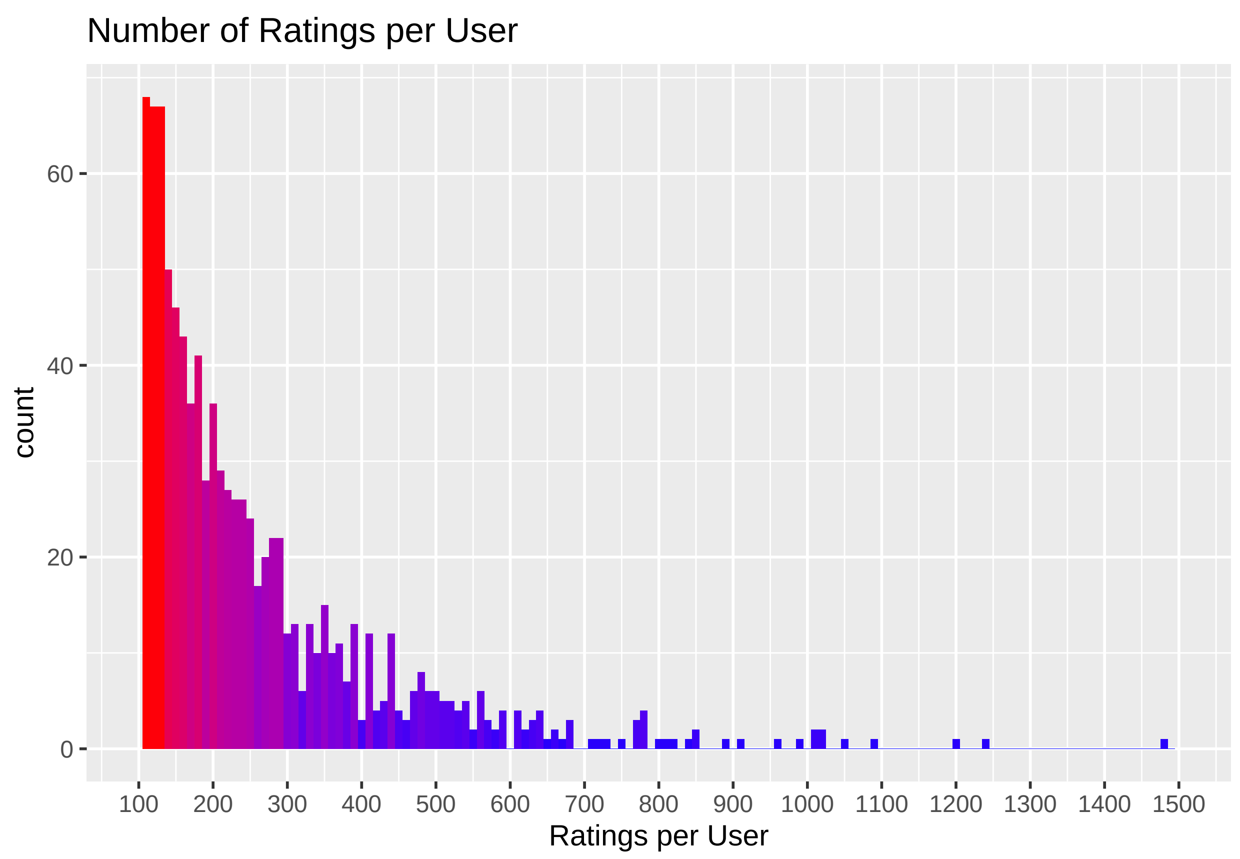
```r
ggplot(data=ratingmat_sample_df,aes(x=rating,y=..count..,fill=..count..)) +
  geom_bar() +
  scale_fill_continuous(low="blue", high="red") +
  scale_x_continuous(limits = c(0,11),breaks = seq(-1,10,1)) +
  ggtitle("Distribution of Ratings") +
  theme(legend.position="none")
```



```
ggplot(data=ratingmat_sample_df_normalized,aes(x=`getRatings(normalize(ratingmat_sample, method = "Z-sc
  geom_histogram(binwidth = 0.05) +
  scale_fill_continuous(low="blue", high="red") +
  xlim(-5,5)+
  xlab("Rating (Normalized)" )+
  ggtitle("Distribution of Ratings, Normalized") +
  theme(legend.position="none")
```

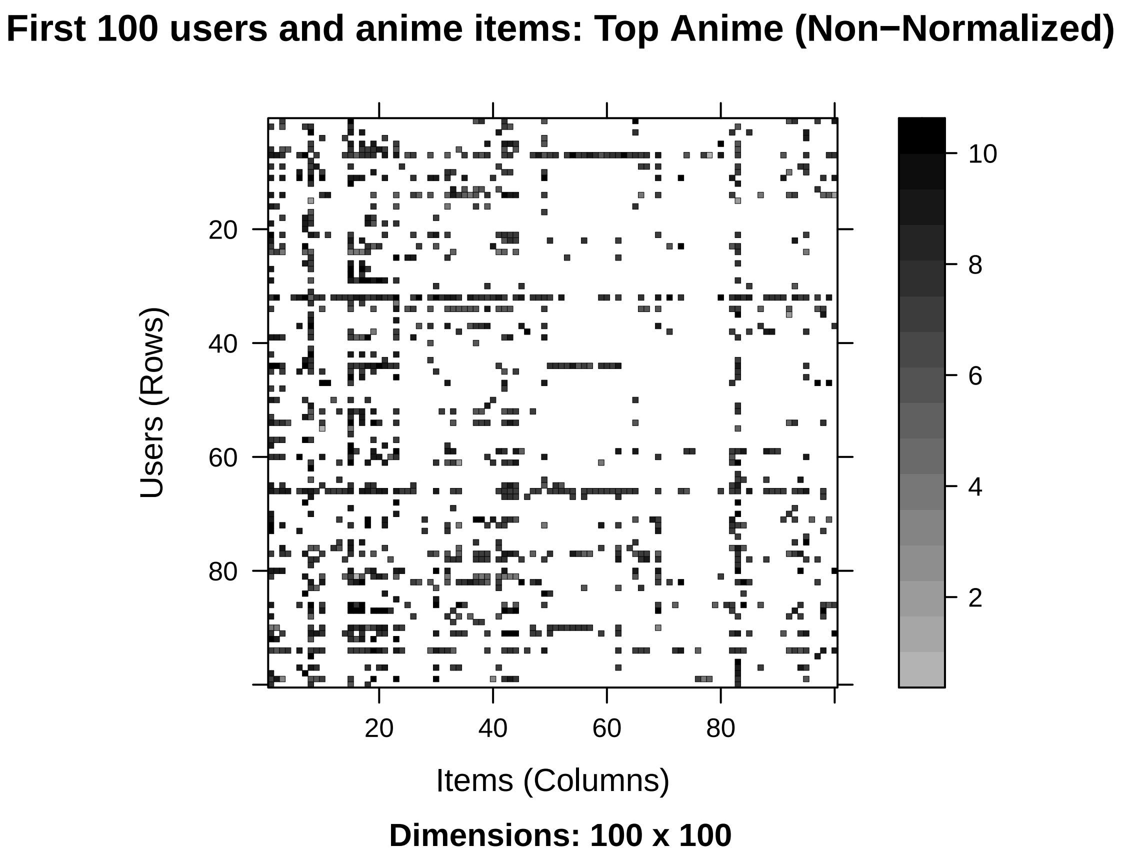


```
ggplot(data=ratingmat_rowcounts,aes(x=`rowCounts(ratingmat_sample)`,y=..count..,fill=..count..)) +
  geom_histogram(binwidth = 10) +
  scale_fill_continuous(low="blue", high="red") +
  scale_x_continuous(limits = c(100,1500),breaks = seq(0,1500,100)) +
  xlab("Ratings per User") +
  ggtitle("Number of Ratings per User") +
  theme(legend.position="none")
```

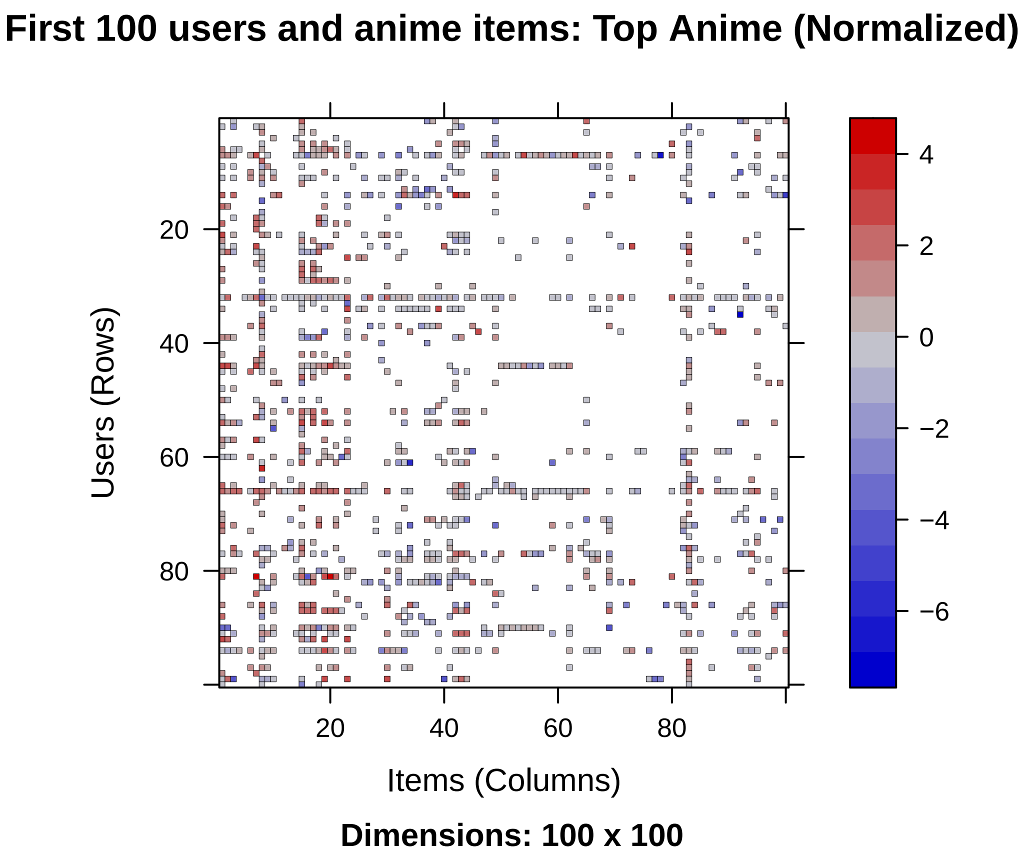


Similarity

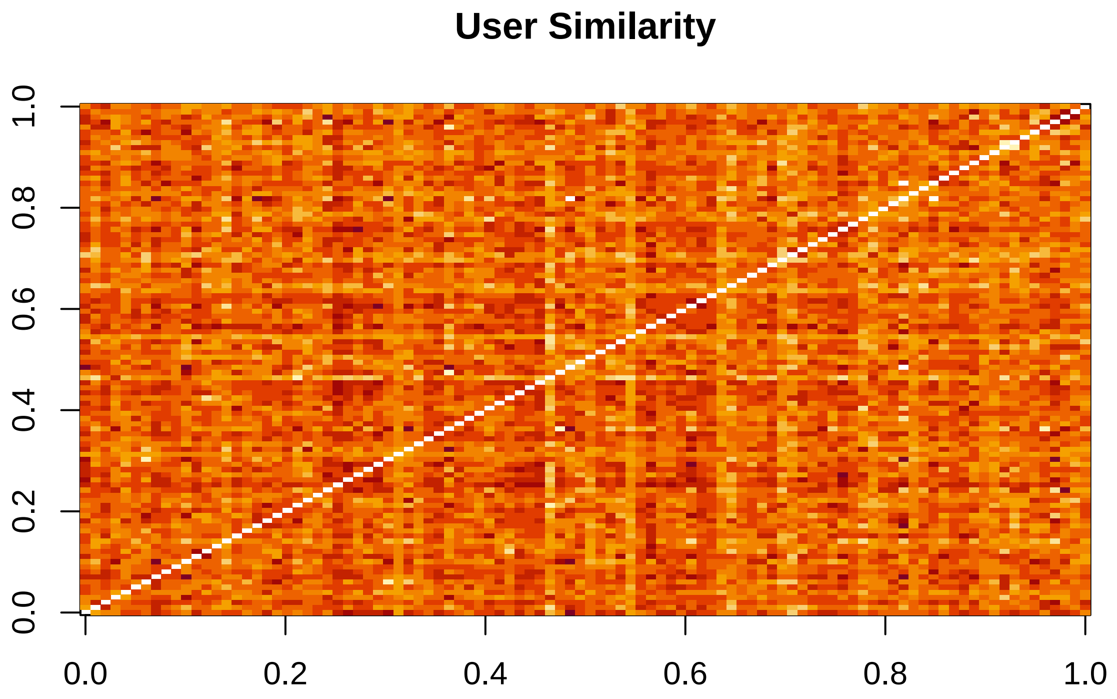
```
image(ratingmat_sample[1:100, 1:100], main = "First 100 users and anime items: Top Anime (Non-Normalize
```



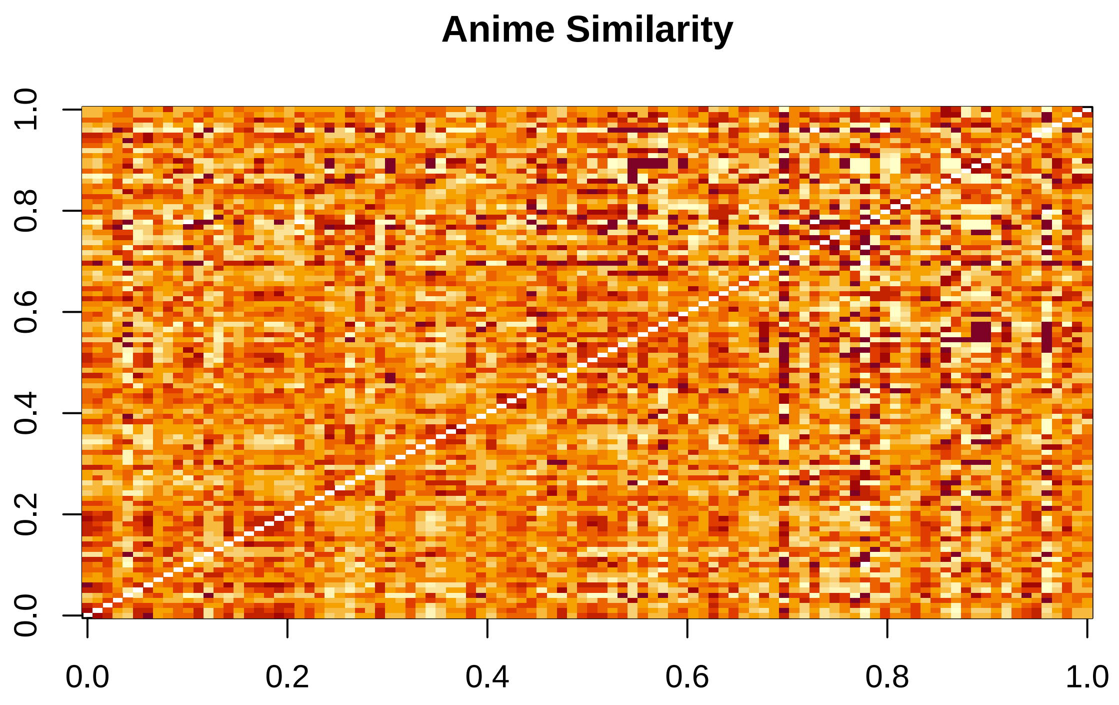
```r
ratingmat_norm <- normalize(ratingmat_sample)
avg <- round(rowMeans(ratingmat_norm), 5)

image(ratingmat_norm[1:100, 1:100], main = "First 100 users and anime items: Top Anime (Normalized)")
```



```
similarity_users <- similarity(ratingmat_norm[1:100,],
                               method = "cosine",
                               which = "users")
image(as.matrix(similarity_users), main = "User Similarity")
```



```
similarity_anime <- similarity(ratingmat_norm[,1:100],
                               method = "cosine",
                               which = "items")
image(as.matrix(similarity_anime), main = "Anime Similarity")
```



Using similarity, we can see that users have more in common than do items, but this is only a preview of the first 100 users and items.

Creating the Algorithms

Creating Evualuation Schemes

This Value represents the minimum value we can choose for the "given" parameter of the evaluationScheme function, which is the number of ratings given per user. We can also choose to implement an all-but-x scheme, where given takes in all ratings except for a user specified number of them. Example: given =-5 excludes 5 ratings from all users to be used in the test set.

In order to evaluate our algorithms, RecommenderLab uses the evaluationScheme function to split the data into train/test sets. There are a few methods available to use, most notably the split and cross methods.

The split method which splits the users into two groups based on the value of the train parameter and uses the first half of the split to train the model, and the second half to evaluate the model. If using an all-but-x scheme, then the scheme uses only the second half of the split to exclude ratings from. Example: method="split", train = 0.9, given = -5 excludes 5 ratings from 10% of users to be used in the test set.

The cross method uses k-fold cross validation and uses the user specified k-value as the number of folds.

To speed up loading, we have already created evaluation schemes and saved them as .rda files, which we load now.

```
anime_eval_split_min <- evaluationScheme(data = ratingmat_sample,
                                         method = "split",
                                         train = 0.8,
                                         given = min(rowCounts(ratingmat_sample)),
                                         goodRating = 5)
anime_eval_split_allbut5 <- evaluationScheme(data = ratingmat_sample,
                                         method = "split",
                                         train = 0.8,
                                         given = -5,
                                         goodRating = 5)
anime_eval_cross_allbut5 <- evaluationScheme(data = ratingmat_sample,
                                         method = "cross",
                                         k=4,
                                         given = -5,
                                         goodRating = 5)
anime_eval_cross_min <- evaluationScheme(data = ratingmat_sample,
                                         method = "cross",
                                         k=4,
                                         given = min(rowCounts(ratingmat_sample)),
                                         goodRating = 5)
```

Which models are available in recommenderlab?

```
recommenderRegistry$get_entry_names()
```

```
##  [1] "HYBRID_realRatingMatrix"          "HYBRID_binaryRatingMatrix"
##  [3] "ALS_realRatingMatrix"             "ALS_implicit_realRatingMatrix"
##  [5] "ALS_implicit_binaryRatingMatrix" "AR_binaryRatingMatrix"
##  [7] "IBCF_binaryRatingMatrix"          "IBCF_realRatingMatrix"
##  [9] "LIBMF_realRatingMatrix"           "POPULAR_binaryRatingMatrix"
## [11] "POPULAR_realRatingMatrix"         "RANDOM_realRatingMatrix"
## [13] "RANDOM_binaryRatingMatrix"        "RERECOMMEND_realRatingMatrix"
## [15] "RERECOMMEND_binaryRatingMatrix"  "SVD_realRatingMatrix"
## [17] "SVDF_realRatingMatrix"            "UBCF_binaryRatingMatrix"
## [19] "UBCF_realRatingMatrix"
```

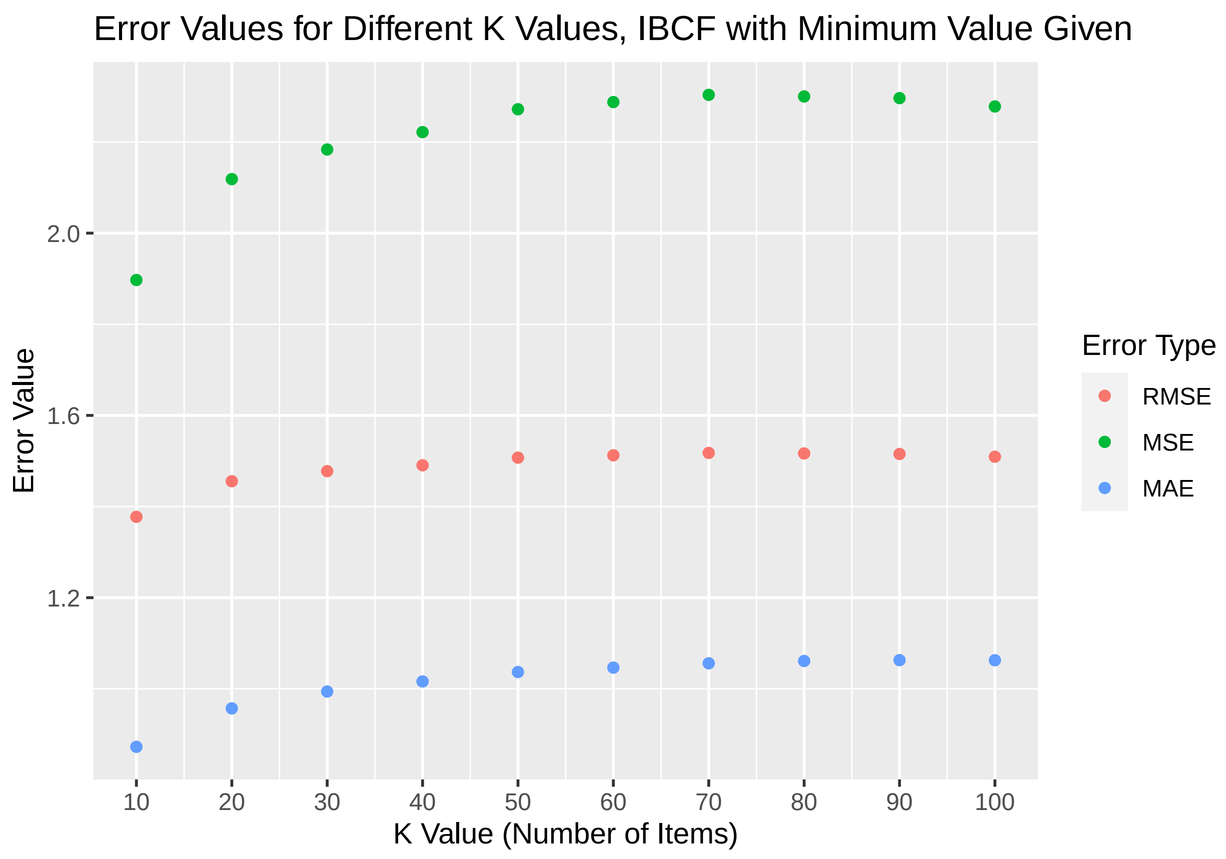
Item-Based Collaborative Filtering Model

Which value of k (number of closest anime to store for each item) to choose?

We have created data structures that contain the results of running time intensive error calculations for 10 different k-values, for both of the following evaluation schemes:

K-Value Error IBCF Split using Minimum number ratings as Given

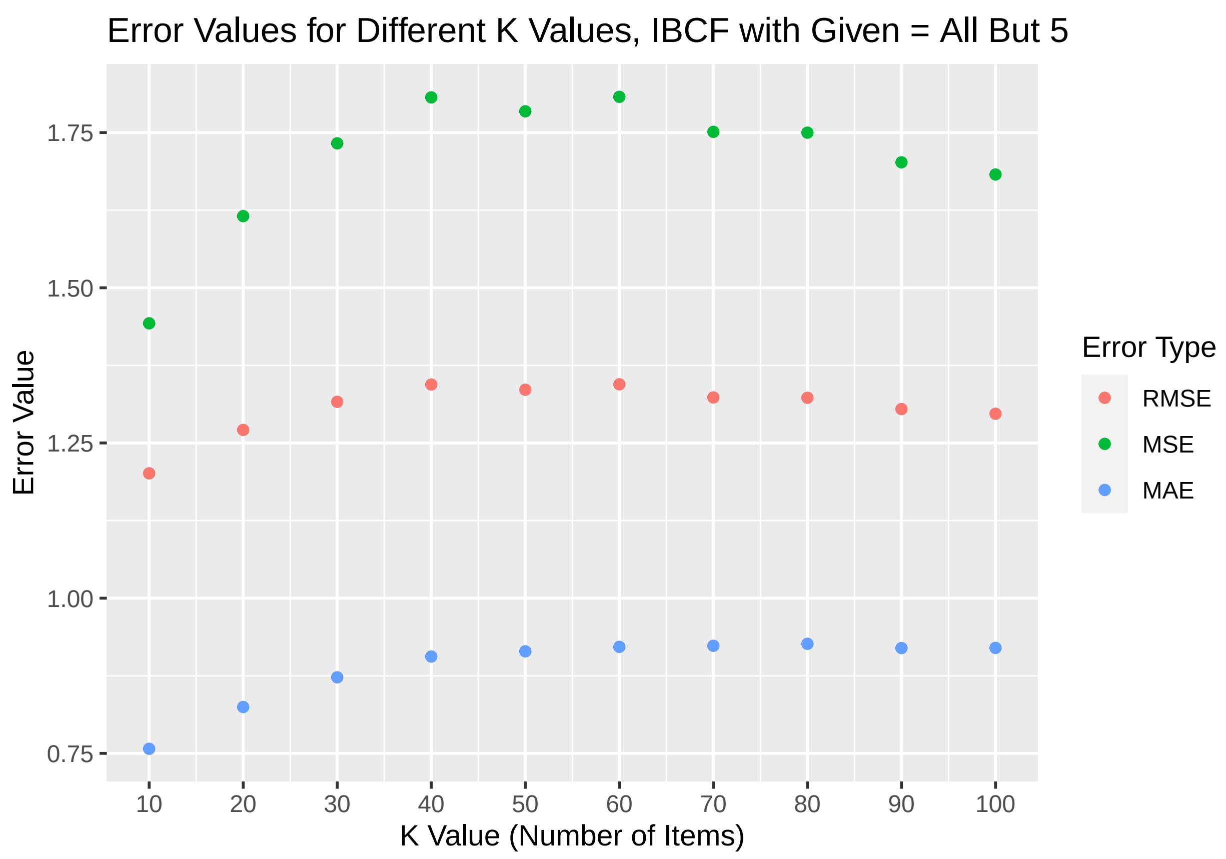
```
load(file="kvalue_nn_errors.rda")
ggplot(data=kvalue_error_IBCF_split_min, aes(k_value,value,col=variable)) +
  geom_point() +
  labs(x="K Value (Number of Items)") +
  labs(y="Error Value") +
  labs(col="Error Type") +
  scale_x_continuous(limits = c(10,100),breaks = seq(0,100,10)) +
  ggtitle("Error Values for Different K Values, IBCF with Minimum Value Given")
```



The error seems to be lowest at small values of K.

K-Value Error IBCF Split using all-but-5 ratings as Given

```
ggplot(data=kvalue_error_IBCF_split_allbut5, aes(k_value,value,col=variable)) +
  geom_point() +
  labs(x="K Value (Number of Items)") +
  labs(y="Error Value") +
  labs(col="Error Type") +
  scale_x_continuous(limits = c(10,100),breaks = seq(0,100,10)) +
  ggtitle("Error Values for Different K Values, IBCF with Given = All But 5")
```



The values seems to be lowest at low values of K.

IBCF Models

```
# ibcf_model <- Recommender(getData(anime_eval_split_min,"train"), method = "IBCF", param = list(normal
# ibcf_model2 <- Recommender(getData(anime_eval_split_allbut5,"train"), method = "IBCF", param = list(n
# ibcf_model3 <- Recommender(getData(anime_eval_cross_allbut5,"train"), method = "IBCF", param = list(n
# ibcf_model4 <- Recommender(getData(anime_eval_cross_min,"train"), method = "IBCF", param = list(norma
# save(ibcf_model,ibcf_model2,ibcf_model3,ibcf_model4, file="ibcf_models.rda")

# Loading model objects to save time
load("ibcf_models.rda")
```

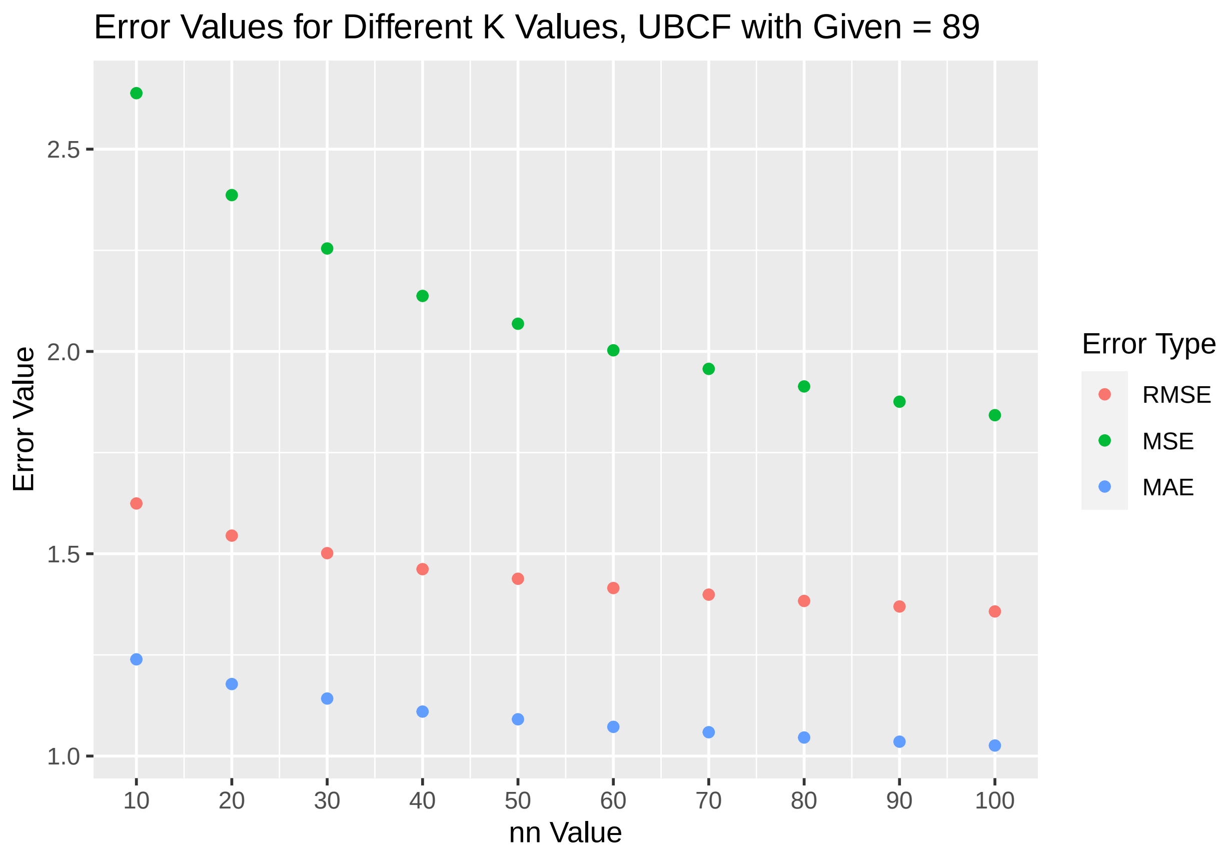
User-Based Collaborative Filtering Model

Which value of nn (number of nearest neighbors) to choose?

We have created data structures that contain the results of running time intensive error calculations for 10 different nn-values, for both of the following evaluation schemes:

NN-Value Error UBCF Split using Minimum number ratings as Given

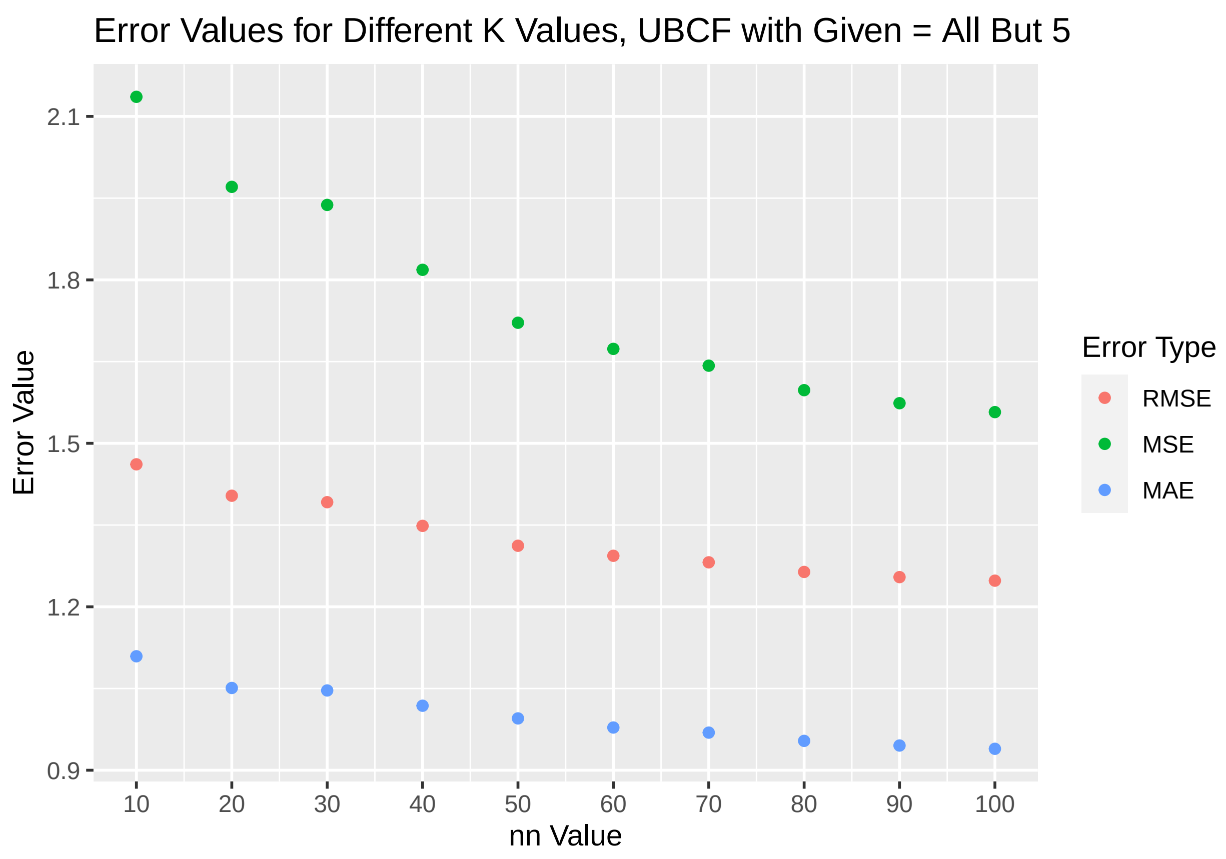
```
ggplot(data=nn_error_UBCF_split_min) +
  geom_point(mapping = aes(nn_number,value,col=variable)) +
  labs(x="nn Value") +
  labs(y="Error Value") +
  labs(col="Error Type") +
  scale_x_continuous(limits = c(10,100),breaks = seq(0,100,10)) +
  ggtitle("Error Values for Different K Values, UBCF with Given = 89")
```



The larger the nn Value, the lower the error.

NN-Value Error UBCF Split using all-but-5 as Given

```
ggplot(data=nn_error_UBCF_split_allbut5) +
  geom_point(mapping = aes(nn_number,value,col=variable)) +
  labs(x="nn Value") +
  labs(y="Error Value") +
  labs(col="Error Type") +
  scale_x_continuous(limits = c(10,100),breaks = seq(0,100,10)) +
  ggtitle("Error Values for Different K Values, UBCF with Given = All But 5")
```



We can see that increasing the number of nearest neighbors decreases all types of errors, but it would also increase computation time. We choose nn=50 as it is a good compromise of accuracy and computation time.

UBCF Models

```
# ubcf_model <- Recommender(getData(anime_eval_split_min,"train"),method = "UBCF",param = list(normaliz
# ubcf_model2 <- Recommender(getData(anime_eval_split_allbut5,"train"),method = "UBCF",param = list(nor
# ubcf_model3 <- Recommender(getData(anime_eval_cross_allbut5,"train"),method = "UBCF",param = list(nor
# ubcf_model4 <- Recommender(getData(anime_eval_cross_min,"train"),method = "UBCF",param = list(normali
# save(ubcf_model,ubcf_model2,ubcf_model3,ubcf_model4, file="ubcf_models.rda")

# Loading model objects to save time
load("ubcf_models.rda")
```

Alternating Least Squares Model

```
# als_model <- Recommender(getData(anime_eval_split_min,"train"), method = "ALS", param =list(n_factors
# als_model2 <- Recommender(getData(anime_eval_split_allbut5,"train"), method = "ALS", param =list(n_fa
# als_model3 <- Recommender(getData(anime_eval_cross_allbut5,"train"), method = "ALS", param =list(n_fa
# als_model4 <- Recommender(getData(anime_eval_cross_min,"train"), method = "ALS", param =list(n_factor
# save(als_model,als_model2,als_model3,als_model4, file = "als_models.rda")
```

```r
# Loading model objects to save time
load("als_models.rda")
```

Singular Value Decomposition Model

```r
# svd_model <- Recommender(getData(anime_eval_split_min,"train"), method = "SVD")
# svd_model2 <- Recommender(getData(anime_eval_split_allbut5,"train"), method = "SVD")
# svd_model3 <- Recommender(getData(anime_eval_cross_allbut5,"train"), method = "SVD")
# svd_model4 <- Recommender(getData(anime_eval_cross_min,"train"), method = "SVD")
# save(svd_model,svd_model2,svd_model3,svd_model4, file = "svd_models.rda")

# Loading model objects to save time
load("svd_models.rda")
```

Hybrid Model

```r
# hybrid_model <- HybridRecommender(
#   Recommender(data = getData(anime_eval_split_min, "train"), method = "IBCF", param = list(normalize
#   Recommender(data = getData(anime_eval_split_min, "train"), method = "UBCF", param = list(normalize
#   Recommender(data = getData(anime_eval_split_min, "train"), method = "SVD"),
#   Recommender(data = getData(anime_eval_split_min, "train"), method = "POPULAR"),
#   weights = c(0.4, 0.2, 0.2, 0.2))
# hybrid_model2 <- HybridRecommender(
#   Recommender(data = getData(anime_eval_split_allbut5, "train"), method = "IBCF", param = list(normal
#   Recommender(data = getData(anime_eval_split_allbut5, "train"), method = "UBCF", param = list(normal
#   Recommender(data = getData(anime_eval_split_allbut5, "train"), method = "SVD"),
#   Recommender(data = getData(anime_eval_split_allbut5, "train"), method = "POPULAR"),
#   weights = c(0.4, 0.2, 0.2, 0.2))
# hybrid_model3 <- HybridRecommender(
#   Recommender(data = getData(anime_eval_cross_allbut5, "train"), method = "IBCF", param = list(normal
#   Recommender(data = getData(anime_eval_cross_allbut5, "train"), method = "UBCF", param = list(normal
#   Recommender(data = getData(anime_eval_cross_allbut5, "train"), method = "SVD"),
#   Recommender(data = getData(anime_eval_cross_allbut5, "train"), method = "POPULAR"),
#   weights = c(0.4, 0.2, 0.2, 0.2))
# hybrid_model4 <- HybridRecommender(
#   Recommender(data = getData(anime_eval_cross_min, "train"), method = "IBCF", param = list(normalize
#   Recommender(data = getData(anime_eval_cross_min, "train"), method = "UBCF", param = list(normalize
#   Recommender(data = getData(anime_eval_cross_min, "train"), method = "SVD"),
#   Recommender(data = getData(anime_eval_cross_min, "train"), method = "POPULAR"),
#   weights = c(0.4, 0.2, 0.2, 0.2))
# save(hybrid_model,hybrid_model2,hybrid_model3,hybrid_model4, file = "hybrid_models.rda")

# Loading model objects to save time
load("hybrid_models.rda")
```

Model Evaluation

Error

We computed errors for ratings generated by the models using three different measurements: RMSE, MSE, and MAE.

RMSE - root mean square error, which penalizes larger errors stronger than AME MAE - mean average error, which is the mean error of all predicted ratings vs actual ratings MSE - similar to MAE, but it uses the squared value of the differences

Compute predicted ratings for the known part of the test data

```r
# ibcf_predict <- predict(ibcf_model, getData(anime_eval_split_min,"known"), type="ratings")
# ibcf_predict2 <- predict(ibcf_model2, getData(anime_eval_split_allbut5,"known"), type="ratings")
# ibcf_predict3 <- predict(ibcf_model3, getData(anime_eval_cross_allbut5,"known"), type="ratings")
# ibcf_predict4 <- predict(ibcf_model4, getData(anime_eval_cross_min,"known"), type="ratings")
#
# ubcf_predict <- predict(ubcf_model, getData(anime_eval_split_min,"known"), type="ratings")
# ubcf_predict2 <- predict(ubcf_model2, getData(anime_eval_split_allbut5,"known"), type="ratings")
# ubcf_predict3 <- predict(ubcf_model3, getData(anime_eval_cross_allbut5,"known"), type="ratings")
# ubcf_predict4 <- predict(ubcf_model4, getData(anime_eval_cross_min,"known"), type="ratings")
#
# als_predict <- predict(als_model, getData(anime_eval_split_min,"known"), type="ratings")
# als_predict2 <- predict(als_model2, getData(anime_eval_split_allbut5,"known"), type="ratings")
# als_predict3 <- predict(als_model3, getData(anime_eval_cross_allbut5,"known"), type="ratings")
# als_predict4 <- predict(als_model4, getData(anime_eval_cross_min,"known"), type="ratings")
#
# svd_predict <- predict(svd_model, getData(anime_eval_split_min,"known"), type="ratings")
# svd_predict2 <- predict(svd_model2, getData(anime_eval_split_allbut5,"known"), type="ratings")
# svd_predict3 <- predict(svd_model3, getData(anime_eval_cross_allbut5,"known"), type="ratings")
# svd_predict4 <- predict(svd_model4, getData(anime_eval_cross_min,"known"), type="ratings")
#
# hybrid_predict <- predict(hybrid_model, getData(anime_eval_split_min,"known"), type="ratings")
# hybrid_predict2 <- predict(hybrid_model2, getData(anime_eval_split_allbut5,"known"), type="ratings")
# hybrid_predict3 <- predict(hybrid_model3, getData(anime_eval_cross_allbut5,"known"), type="ratings")
# hybrid_predict4 <- predict(hybrid_model4, getData(anime_eval_cross_min,"known"), type="ratings")
#
# save(ibcf_predict, ibcf_predict2, ibcf_predict3, ibcf_predict4, ubcf_predict, ubcf_predict2, ubcf_pre

# Loading prediction objects to save time
load("prediction_models.rda")
```

Calculating the error between the prediction and the unknown part of the data

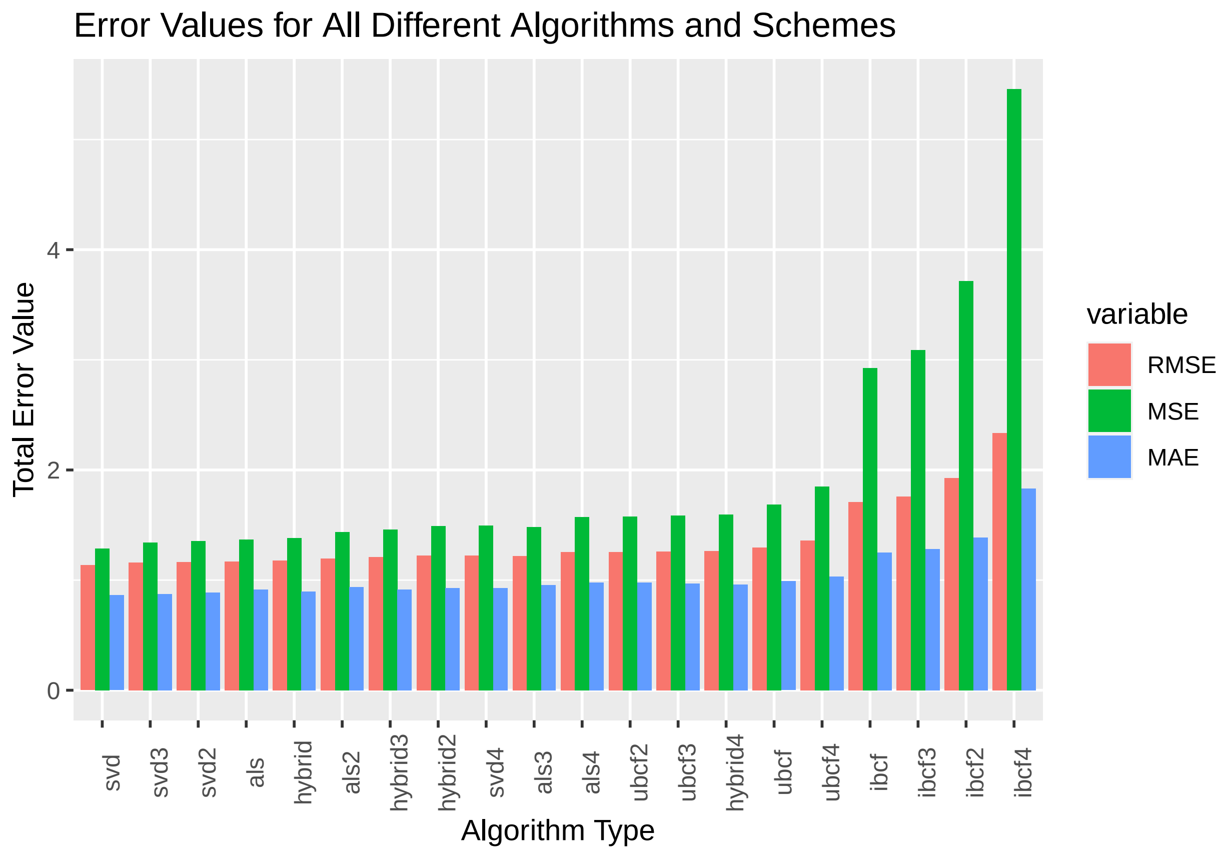
```r
error <- rbind(
ibcf = calcPredictionAccuracy(ibcf_predict, getData(anime_eval_split_min, "unknown")),
ibcf2 = calcPredictionAccuracy(ibcf_predict2, getData(anime_eval_split_allbut5, "unknown")),
ibcf3 = calcPredictionAccuracy(ibcf_predict3, getData(anime_eval_cross_allbut5, "unknown")),
ibcf4 = calcPredictionAccuracy(ibcf_predict3, getData(anime_eval_cross_min, "unknown")))
```

```
error2 <- rbind(
ubcf = calcPredictionAccuracy(ubcf_predict, getData(anime_eval_split_min, "unknown")),
ubcf2 = calcPredictionAccuracy(ubcf_predict2, getData(anime_eval_split_allbut5, "unknown")),
ubcf3 = calcPredictionAccuracy(ubcf_predict3, getData(anime_eval_cross_allbut5, "unknown")),
ubcf4 = calcPredictionAccuracy(ubcf_predict4, getData(anime_eval_cross_min, "unknown")))

error3 <- rbind(
als = calcPredictionAccuracy(als_predict, getData(anime_eval_split_min, "unknown")),
als2 = calcPredictionAccuracy(als_predict2, getData(anime_eval_split_allbut5, "unknown")),
als3 = calcPredictionAccuracy(als_predict3, getData(anime_eval_cross_allbut5, "unknown")),
als4= calcPredictionAccuracy(als_predict4, getData(anime_eval_cross_min, "unknown")))

error4 <- rbind(
svd = calcPredictionAccuracy(svd_predict, getData(anime_eval_split_min, "unknown")),
svd2 = calcPredictionAccuracy(svd_predict2, getData(anime_eval_split_allbut5, "unknown")),
svd3 = calcPredictionAccuracy(svd_predict3, getData(anime_eval_cross_allbut5, "unknown")),
svd4= calcPredictionAccuracy(svd_predict4, getData(anime_eval_cross_min, "unknown")))

error5 <- rbind(
hybrid = calcPredictionAccuracy(hybrid_predict, getData(anime_eval_split_min, "unknown")),
hybrid2 = calcPredictionAccuracy(hybrid_predict2, getData(anime_eval_split_allbut5, "unknown")),
hybrid3 = calcPredictionAccuracy(hybrid_predict3, getData(anime_eval_cross_allbut5, "unknown")),
hybrid4 = calcPredictionAccuracy(hybrid_predict4, getData(anime_eval_cross_min, "unknown")))

error_total <- rbind(as.data.frame(error),as.data.frame(error2),as.data.frame(error3),as.data.frame(erro
error_total
```

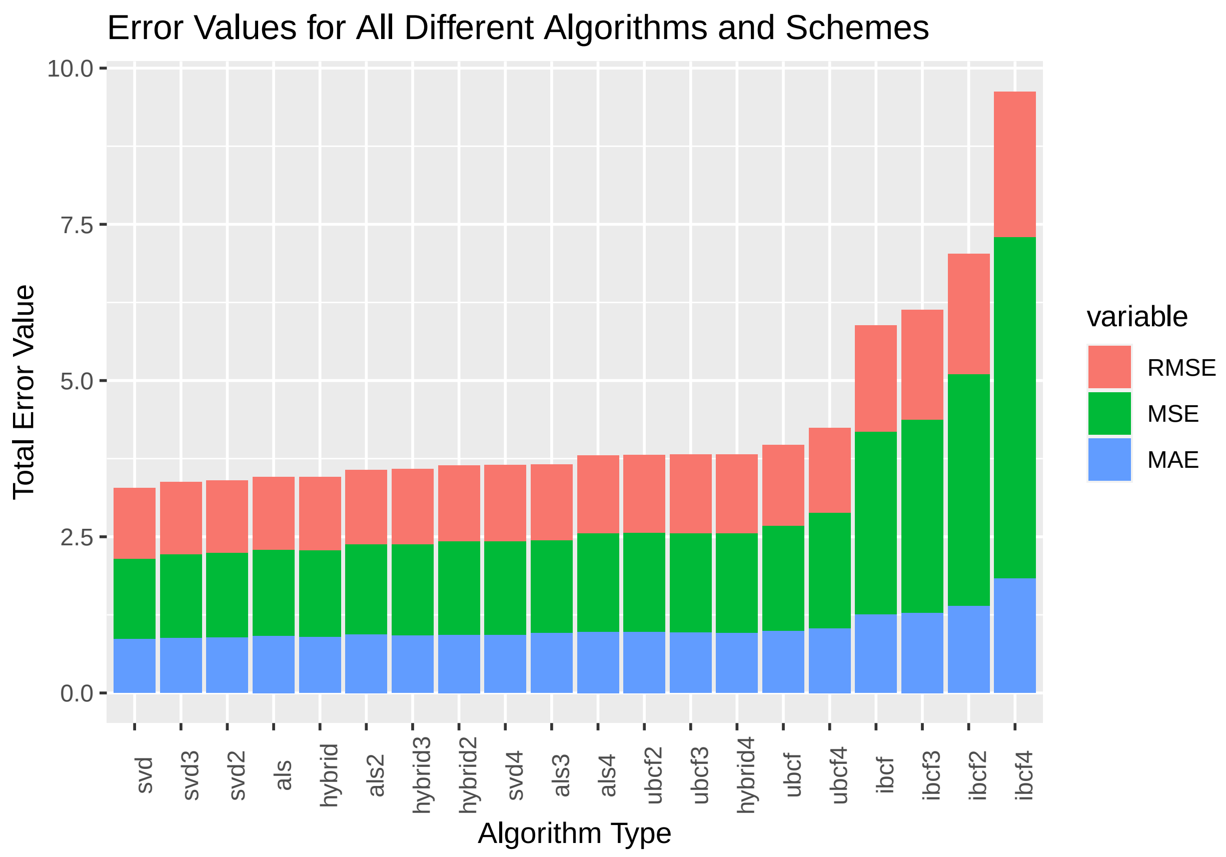
```
##              RMSE      MSE       MAE
## ibcf    1.709994 2.924080 1.2513874
## ibcf2   1.927469 3.715136 1.3888328
## ibcf3   1.757725 3.089597 1.2821422
## ibcf4   2.336386 5.458698 1.8321631
## ubcf    1.298101 1.685067 0.9894274
## ubcf2   1.256568 1.578963 0.9769684
## ubcf3   1.260093 1.587833 0.9672050
## ubcf4   1.360280 1.850361 1.0328353
## als     1.170515 1.370106 0.9134030
## als2    1.197928 1.435032 0.9379296
## als3    1.218107 1.483786 0.9573560
## als4    1.254722 1.574326 0.9769839
## svd     1.134805 1.287783 0.8624381
## svd2    1.163191 1.353012 0.8874547
## svd3    1.158914 1.343083 0.8756684
## svd4    1.223662 1.497347 0.9271904
## hybrid  1.176606 1.384403 0.8962583
## hybrid2 1.221871 1.492968 0.9298499
## hybrid3 1.208936 1.461527 0.9165502
## hybrid4 1.262488 1.593877 0.9599486
```

Interpretation of Errors

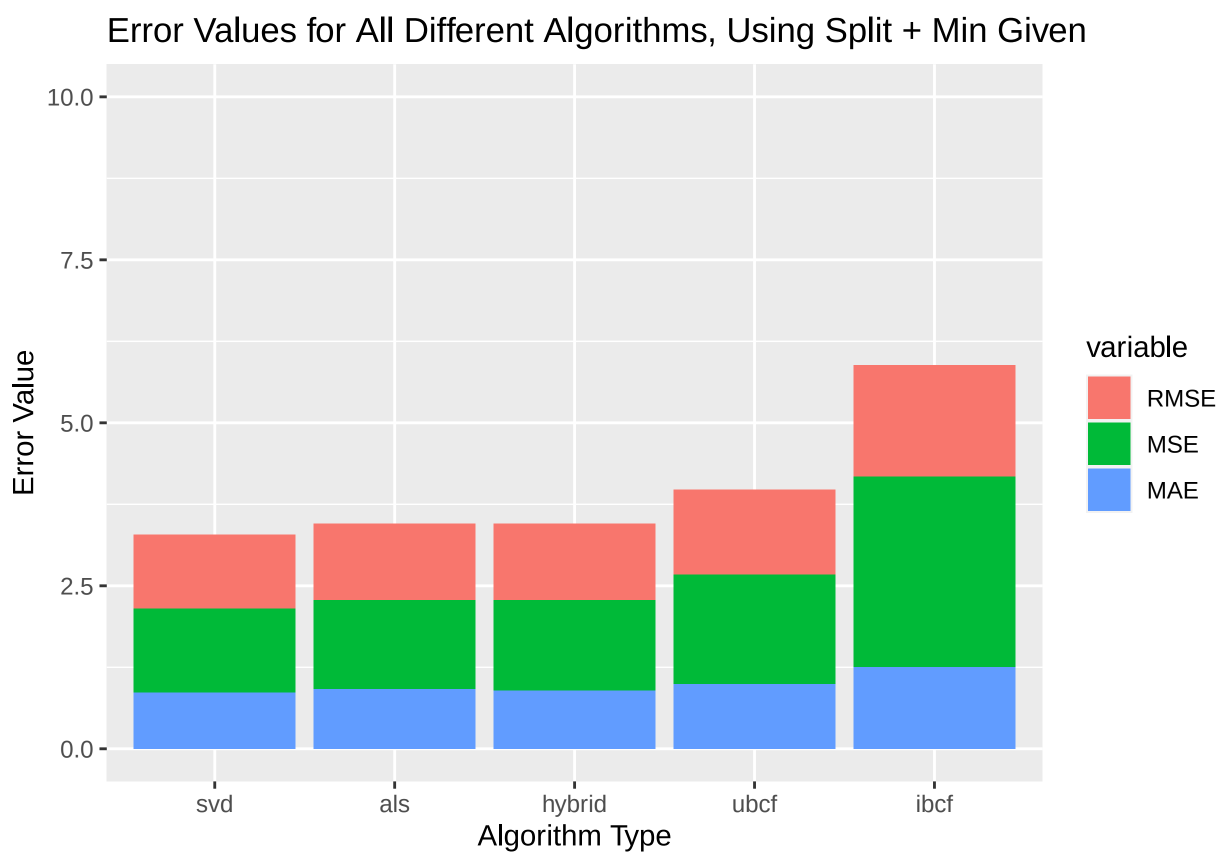
```
error_total[ "algo_type" ] <- rownames(error_total)
error_total_melted <- melt(error_total, id.vars = "algo_type")

error_total_split_min <- error_total[c("ibcf","ubcf","als","svd","hybrid"),]
error_total_split_allbut5 <- error_total[c("ibcf2","ubcf2","als2","svd2","hybrid2"),]
error_total_cross_allbut5 <- error_total[c("ibcf3","ubcf3","als3","svd3","hybrid3"),]
error_total_cross_min <- error_total[c("ibcf4","ubcf4","als4","svd4","hybrid4"),]

error_total_split_min_melted <- melt(error_total_split_min, id.vars = "algo_type")
error_total_split_allbut5_melted <- melt(error_total_split_allbut5, id.vars = "algo_type")
error_total_cross_allbut5_melted <- melt(error_total_cross_allbut5, id.vars = "algo_type")
error_total_cross_min_melted <- melt(error_total_cross_min, id.vars = "algo_type")

ggplot(data = error_total_melted,aes(x=reorder(algo_type,value),y=value,fill=variable)) +
  geom_bar(stat = 'identity', position = 'dodge') +
  labs(x="Algorithm Type") +
  labs(y="Total Error Value") +
  labs(col="Error Type") +
  ggtitle("Error Values for All Different Algorithms and Schemes")+
  theme(axis.text.x = element_text(angle = 90))
```



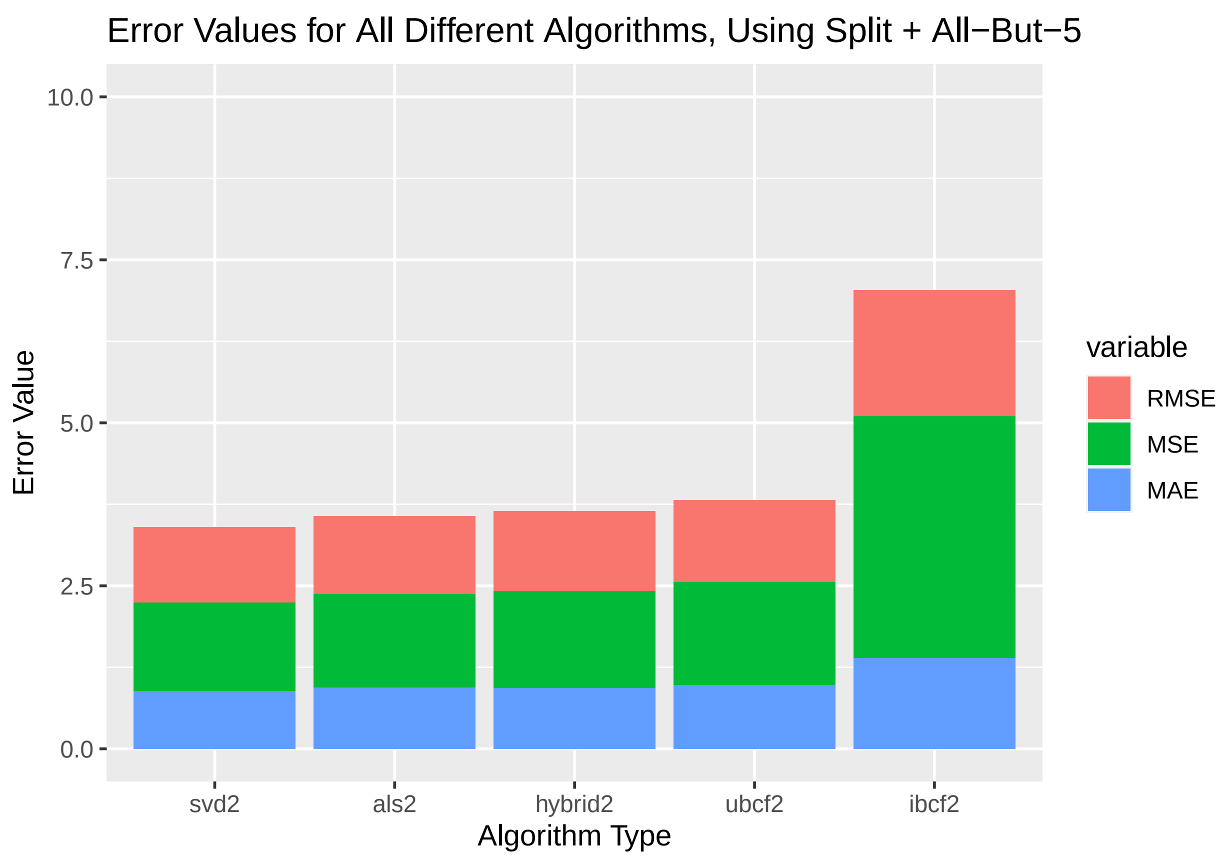
```r
ggplot(data = error_total_melted,aes(x=reorder(algo_type,value),y=value,fill=variable)) +
  geom_bar(stat = 'identity') +
  labs(x="Algorithm Type") +
  labs(y="Total Error Value") +
  labs(col="Error Type") +
  ggtitle("Error Values for All Different Algorithms and Schemes")+
  theme(axis.text.x = element_text(angle = 90))
```



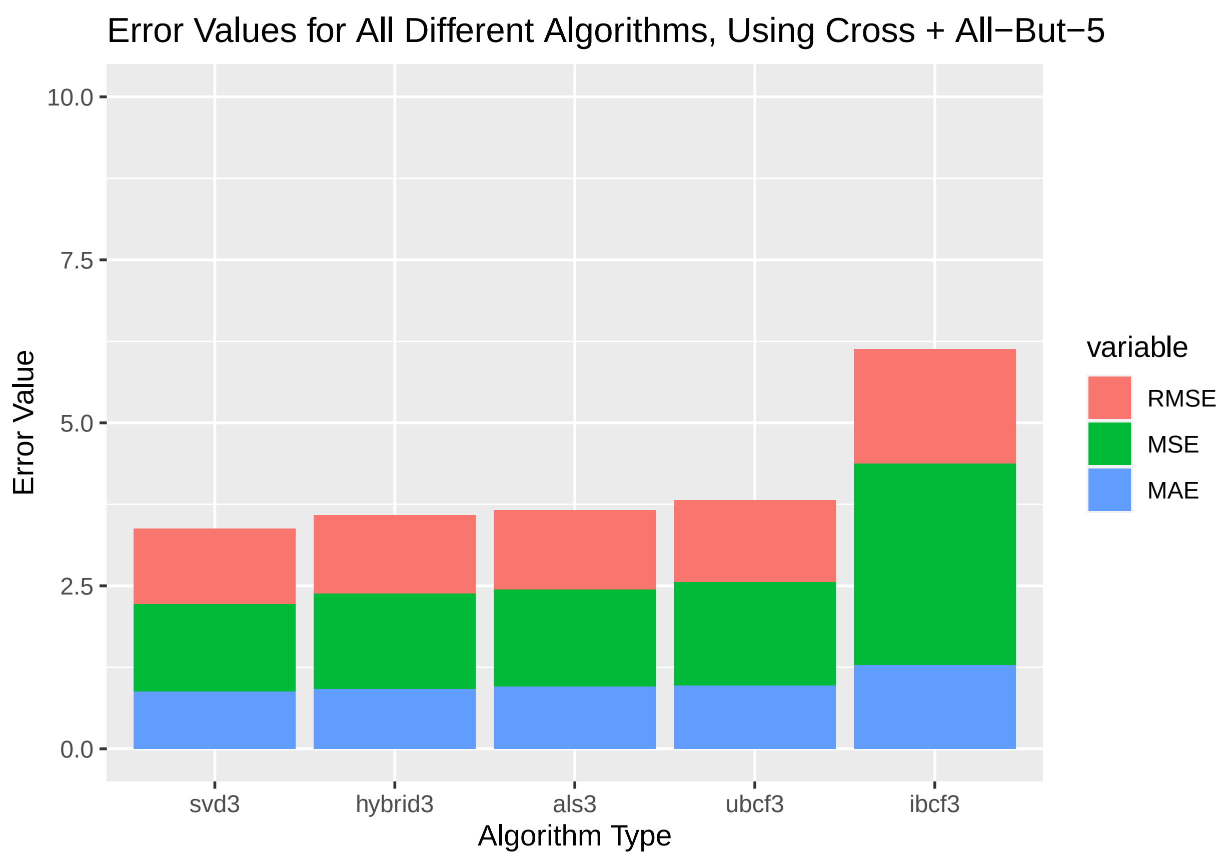
```r
ggplot(data = error_total_split_min_melted,aes(x=reorder(algo_type,value),y=value,fill=variable)) +
  geom_bar(stat = 'identity') +
  labs(x="Algorithm Type") +
  labs(y="Error Value") +
  labs(col="Error Type") +
  ylim(0,10)+
  ggtitle("Error Values for All Different Algorithms, Using Split + Min Given")
```



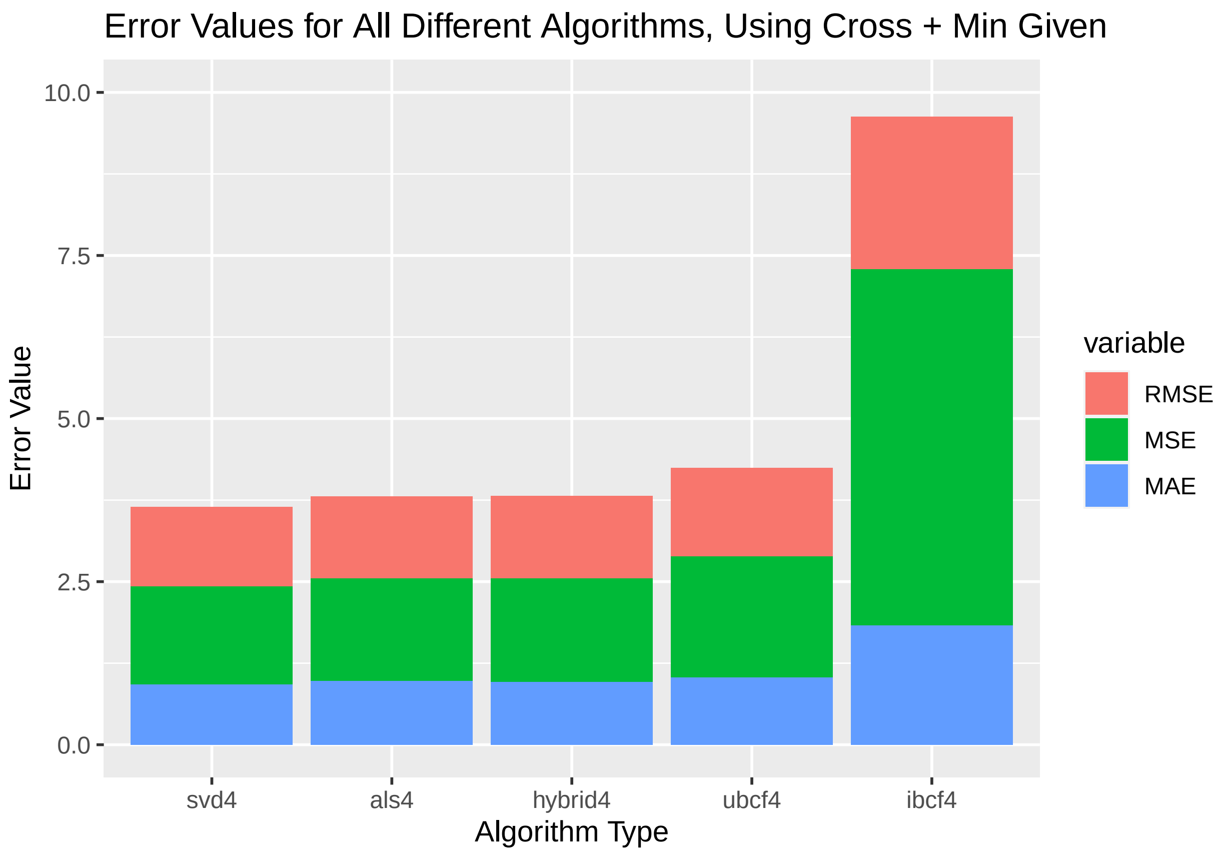
```
ggplot(data = error_total_split_allbut5_melted,aes(x=reorder(algo_type,value),y=value,fill=variable)) +
  geom_bar(stat = 'identity') +
  labs(x="Algorithm Type") +
  labs(y="Error Value") +
  labs(col="Error Type") +
  ylim(0,10)+
  ggtitle("Error Values for All Different Algorithms, Using Split + All-But-5")
```



```
ggplot(data = error_total_cross_allbut5_melted,aes(x=reorder(algo_type,value),y=value,fill=variable)) +
  geom_bar(stat = 'identity') +
  labs(x="Algorithm Type") +
  labs(y="Error Value") +
  labs(col="Error Type") +
  ylim(0,10)+
  ggtitle("Error Values for All Different Algorithms, Using Cross + All-But-5")
```



```
ggplot(data = error_total_cross_min_melted,aes(x=reorder(algo_type,value),y=value,fill=variable)) +
  geom_bar(stat = 'identity') +
  labs(x="Algorithm Type") +
  labs(y="Error Value") +
  labs(col="Error Type") +
  ylim(0,10)+
  ggtitle("Error Values for All Different Algorithms, Using Cross + Min Given")
```



In general, using all-but-5 as the "given" parameter within Recommenderlab generates results with lower error values. It is more computationally expensive, but since we are already sampling our data, it doesn't add much to computation time. Using all-but-5 makes it so that the models are able to use more data for training instead of being limited to the minimum value of 89 (in our case), the models can make use of users who have ratings much greater than 89.

We calculated the errors using both the 80/20 Split Method and using 4-fold Cross-Fold Validation. We were happy to see that the errors from both methods weren't significantly different from each other, but we used both just to make sure our errors were double checked. 4-fold Cross-Validation is more computationally expensive, which is probably why it was able to detect a slightly higher level of error.

For the above reasons, we will be using the All-But-5 "Given" parameter for our anime recommender application for all algorithms.

Accuracy

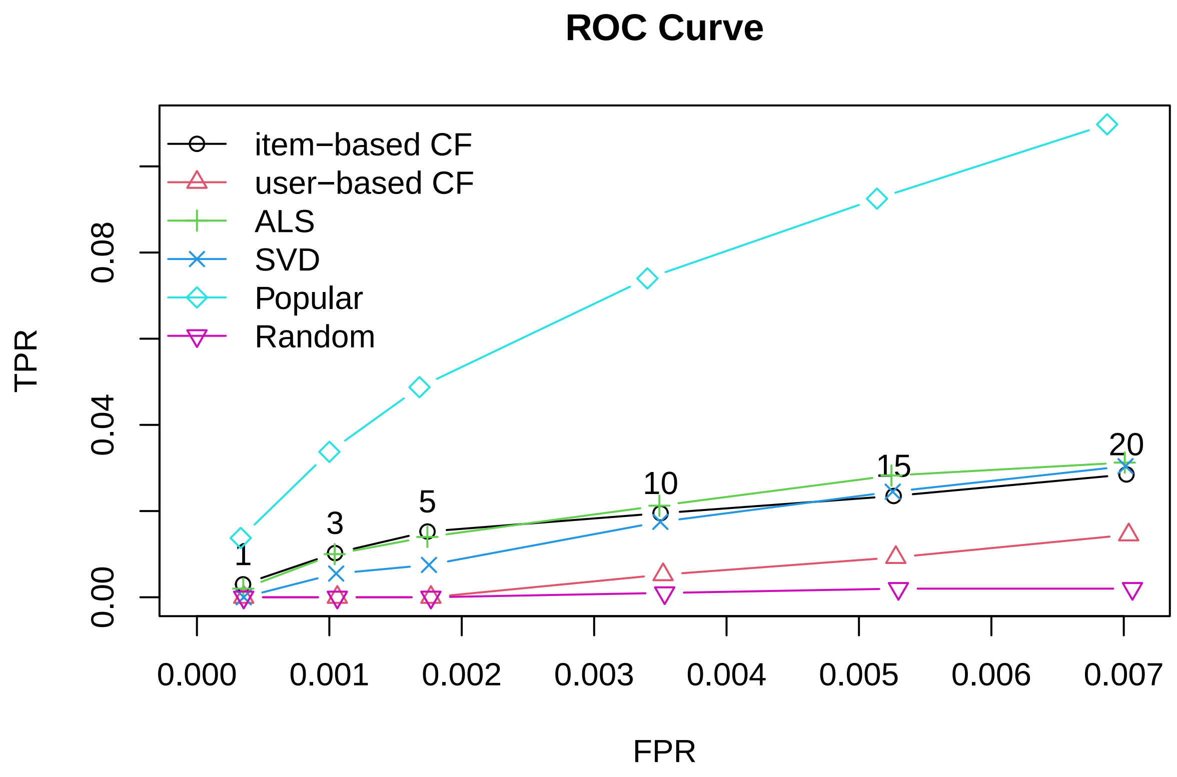
To compute accuracy for a recommender system, we have to set a rating value that seperates ratings into correct or incorrect ratings. We have chosen the value of "7" since it is the rounded average of the ratings in our data set. An ROC curve plots the true positive rate against the false positive rate, and is linear with a slope of one with a random classifier. The larger the area under the curve, the better the classifier performance.

ROC Curve

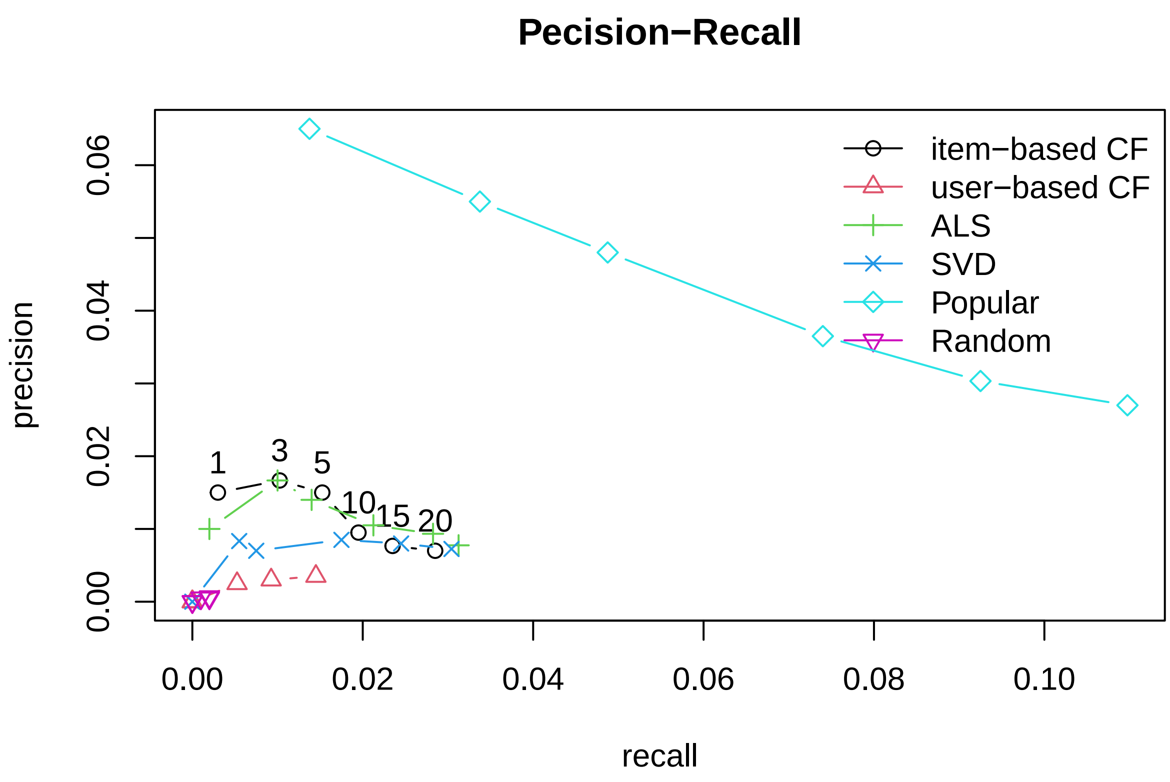
```r
algorithms <- list(
  "item-based CF" = list(name="IBCF", param = list(normalize = "center", method="Cosine", k=50)),
  "user-based CF" = list(name="UBCF", param = list(normalize = "center",method="Cosine",nn=50)),
  "ALS" = list(name="ALS", param = list(n_iterations = 5)),
  "SVD" = list(name="SVD"),
  "Popular"= list(name="POPULAR"),
  "Random"= list(name="Random")
  )
results <- evaluate(anime_eval_split_allbut5, algorithms,type="topNList", n = c(1, 3, 5, 10,15, 20))
```

```
## IBCF run fold/sample [model time/prediction time]
##   1  [11.1sec/0.19sec]
## UBCF run fold/sample [model time/prediction time]
##   1  [0.02sec/2.25sec]
## ALS run fold/sample [model time/prediction time]
##   1  [0sec/32.87sec]
## SVD run fold/sample [model time/prediction time]
##   1  [0.27sec/0.2sec]
## POPULAR run fold/sample [model time/prediction time]
##   1  [0.02sec/0.59sec]
## Random run fold/sample [model time/prediction time]
##   1  [0sec/0.13sec]
```

```r
plot(results, annotate=T, legend="topleft")
title("ROC Curve")
```



```r
plot(results, "prec/rec", annotate = T, legend = "topright")
title("Pecision-Recall")
```

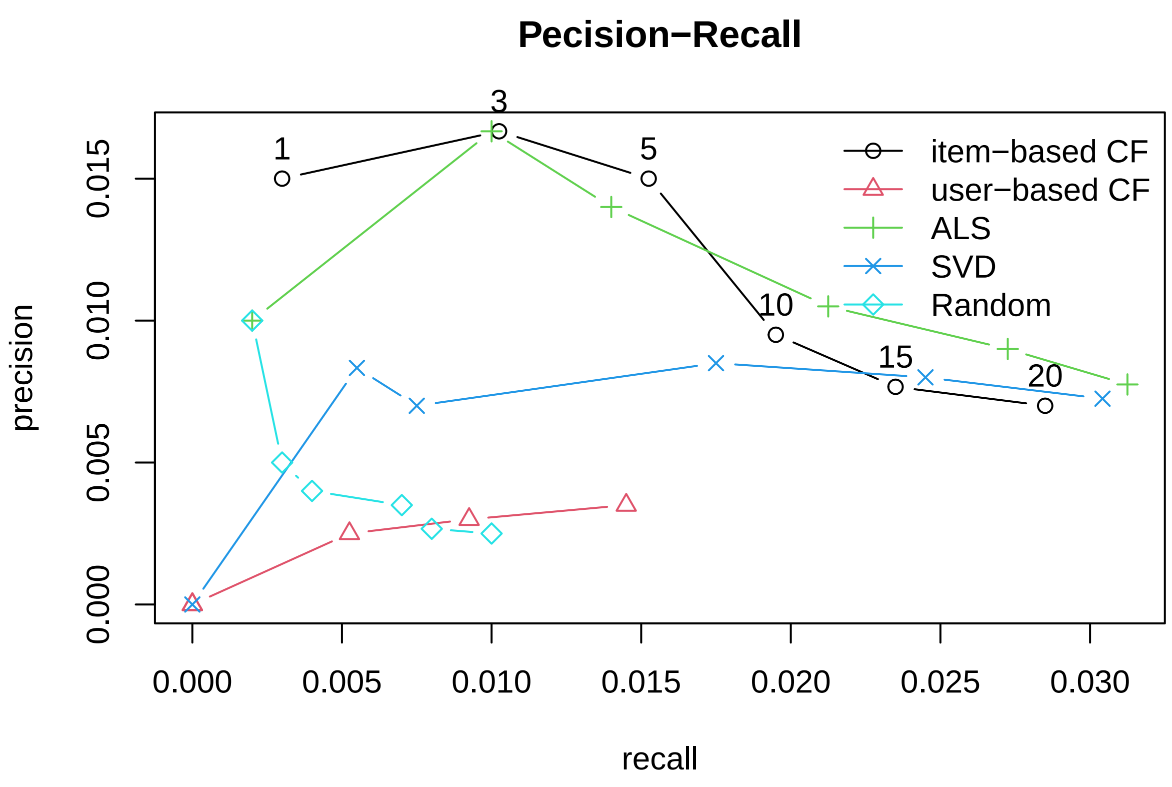


Precision_recall Curve is being dominated by the Popular algorithm, so lets look at it without this algorithm.

```r
algorithms <- list(
  "item-based CF" = list(name="IBCF", param = list(normalize = "center", method="Cosine", k=50)),
  "user-based CF" = list(name="UBCF", param = list(normalize = "center",method="Cosine",nn=50)),
  "ALS" = list(name="ALS", param = list(n_iterations = 5)),
  "SVD" = list(name="SVD"),
  "Random"= list(name="Random")
  )
results <- evaluate(anime_eval_split_allbut5, algorithms,type="topNList", n = c(1, 3, 5, 10,15, 20))
```

```
## IBCF run fold/sample [model time/prediction time]
##   1  [12.88sec/0.14sec]
## UBCF run fold/sample [model time/prediction time]
##   1  [0.03sec/2.55sec]
## ALS run fold/sample [model time/prediction time]
##   1  [0sec/32.63sec]
## SVD run fold/sample [model time/prediction time]
##   1  [0.28sec/0.18sec]
## Random run fold/sample [model time/prediction time]
##   1  [0sec/0.14sec]
```

```
plot(results, "prec/rec", annotate = T, legend = "topright")
title("Pecision-Recall")
```



Which Models Performed the Best?

IBCF: Relatively slow time to build model, but nearly instant predictions. Doesn't need to be recreated for a new user.

UBCF: Instant time to build model, slightly slow predictions. Does need to be recreated for a new user.

ALS: Instant time to build model, slow predictions. Does need to be recreated for a new user.

SVD: Quick time to build model, quick predictions. Does need to be recreated for a new user.

Example Recommendations for One User

RecommenderLab creates predictions using predict() to create an objects of various classes like a TopNList.

```
getAnimeNames <- function(df) {
  # function that takes as input a list of MAL_ID,
  # and returns a data frame containing the corresponding English names
  out = as.data.frame(matrix(ncol=1,nrow=5))
  names(out)[1] <- 'Anime Name'
  for(i in 1:nrow(df)) {
    value <- df[[i,1]]
    row <- which(anime_df[["MAL_ID"]]==value)
    out[i,1] <- anime_df[row,5]
  }
  return(out)
}

ibcf_recom<-predict(ibcf_model2, ratingmat_sample[1],n=5)
ubcf_recom<-predict(ubcf_model2, ratingmat_sample[1],n=5)
als_recom<-predict(als_model2, ratingmat_sample[1],n=5)
svd_recom<-predict(svd_model2, ratingmat_sample[1],n=5)

getAnimeNames(as.data.frame(as(ibcf_recom,"list")))
```

```
##                      Anime Name
## 1                       Berserk
## 2                   Dragon Ball
## 3         Great Teacher Onizuka
## 4 Daphne in the Brilliant Blue
## 5          Casshan:Robot Hunter
```

```
getAnimeNames(as.data.frame(as(ubcf_recom,"list")))
```

```
##                                     Anime Name
## 1 The Girl From the Other Side:SiÃºil, a RÃºn
## 2                         Aria the Origination
## 3                                         Erin
## 4                            Millennium Actress
## 5                  Legend of the Galactic Heroes
```

```
getAnimeNames(as.data.frame(as(als_recom,"list")))
```

```
##                              Anime Name
## 1         Legend of the Galactic Heroes
## 2                               Monster
## 3 March Comes In Like A Lion 2nd Season
## 4                         Hunter x Hunter
## 5                                   Erin
```

```
getAnimeNames(as.data.frame(as(svd_recom,"list")))
```

```
##                              Anime Name
## 1         Legend of the Galactic Heroes
## 2                 Clannad ~After Story~
## 3                               Monster
## 4                         Hunter x Hunter
## 5 March Comes In Like A Lion 2nd Season
```