Recommender Systems

Tulasi Ramarao

7/6/2017

DATA643 Project 5

Description

In this project, a recommender was implemented on a distrubuted system. The Performance of this recommender was then compared with the recommender that was created on Apache Spark using ALS.

Dataset

The dataset from MovieLens(Ref#:5) listed under Recommended for Education and Research was used. This dataset contains 100,000 ratings for 9,000 movies by 700 users. The ratings.csv and movies.csv were used to build the recommendation system.

Installations:

The first attempt was to build this recommender on databricks.com that runs on Amazon Web Services (AWS)[Fig.1]. The Spark connection was successfully established and the data was loaded succesfully. The queries worked until the calls called sdf_copy_to() and sdf_import() were made. The error messages that appeared were not easy to debug even with the help of google search.

So, this project on databricks was abandoned due to time restrictions and an alternative approach was chosen, which was to install the recommender on Apache Spark on a single node (Ref#:4).

Sparklyr was recommended by Ref# 1 because it is an easier environment to work with, when compared to SparkR. The data manipulation in Sparklyr uses the same verbage as dplyr, so the learning curve is said to be easier for R programmers. Also, Sparkly is faster than R and help documentation are easily available in R (?function_name).

```
##
## The downloaded binary packages are in
##  /var/folders/y0/r9w_xkcs48z_sdkj9pd8g5p80000gn/T//Rtmp7ULOeT/downloaded_packages
```

The returned spark connection(sc) below provides a remote dplyr data source to the Spark cluster

Loaded the smaller dataset (1M) for simplicty

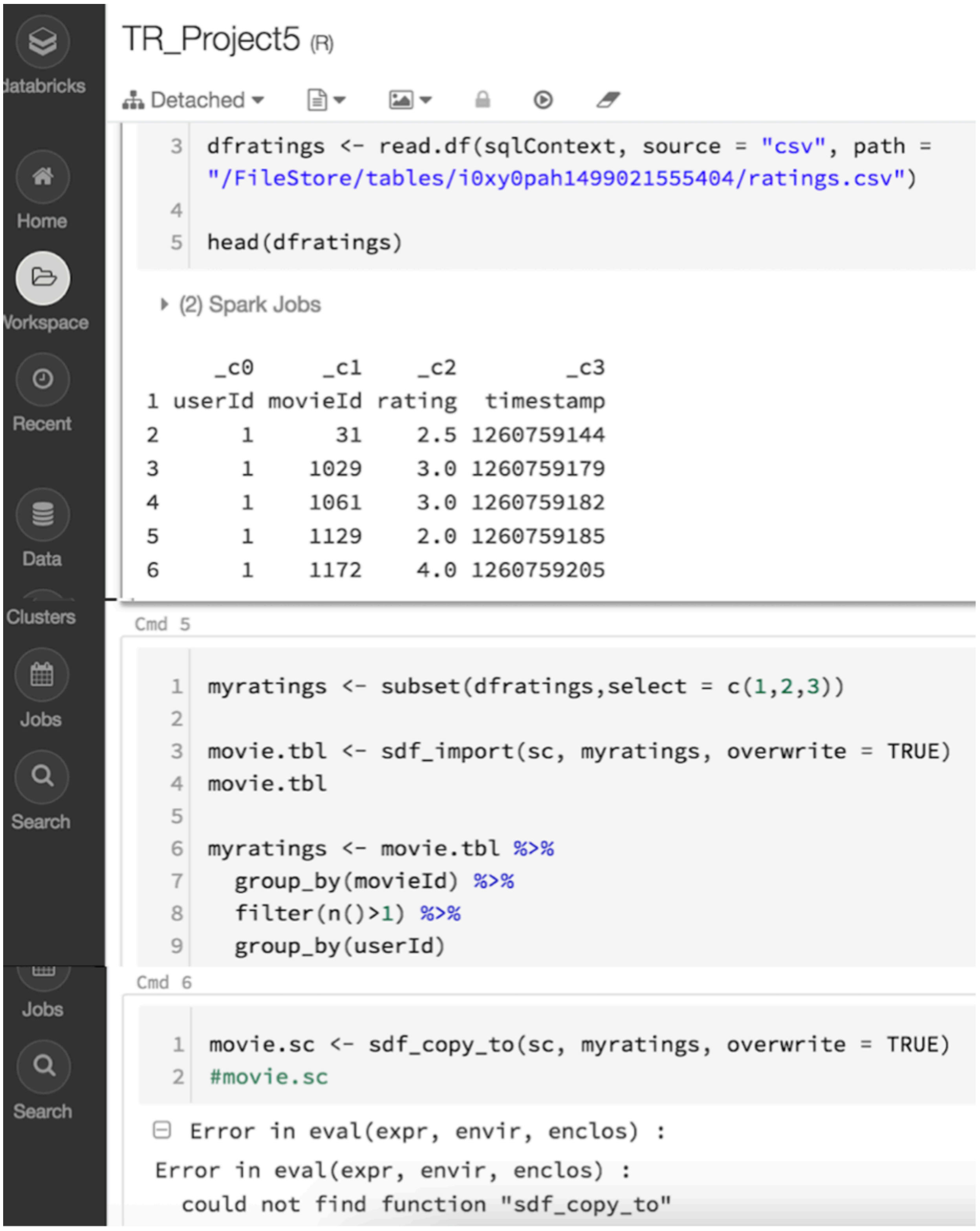
```
dfratings <- read.csv("MovieRatingsData/ml-latest-small/ratings.csv", header = TRUE, sep =",",
                    stringsAsFactors = FALSE)
colnames(dfratings)
```

```
## [1] "userId"    "movieId"   "rating"    "timestamp"
```

```
dfmovies <- read.csv("MovieRatingsData/ml-latest-small/movies.csv", header = TRUE, sep =",",
                    stringsAsFactors = FALSE)
colnames(dfmovies)
```

```
## [1] "movieId" "title"   "genres"
```

Now, the two datasets are merged - to get a movie name for each movieId.



```
dfratings <- read.df(sqlContext, source = "csv", path =
"/FileStore/tables/i0xy0pah1499021555404/ratings.csv")

head(dfratings)
```

```
      _c0     _c1    _c2        _c3
1 userId movieId rating  timestamp
2      1      31    2.5 1260759144
3      1    1029    3.0 1260759179
4      1    1061    3.0 1260759182
5      1    1129    2.0 1260759185
6      1    1172    4.0 1260759205
```

```
myratings <- subset(dfratings,select = c(1,2,3))

movie.tbl <- sdf_import(sc, myratings, overwrite = TRUE)
movie.tbl

myratings <- movie.tbl %>%
  group_by(movieId) %>%
  filter(n()>1) %>%
  group_by(userId)
```

```
movie.sc <- sdf_copy_to(sc, myratings, overwrite = TRUE)
#movie.sc
```

```
Error in eval(expr, envir, enclos) :
Error in eval(expr, envir, enclos) :
  could not find function "sdf_copy_to"
```

Fig. 1: Screenshot from a workspace on Databricks

Figure 1:

```r
combinedData <- merge(dfratings,dfmovies, by=c("movieId"))
colnames(combinedData)
```

```
## [1] "movieId"   "userId"    "rating"    "timestamp" "title"     "genres"
```

```r
#Select the relevant columns - skip timestamp and genres, movieID
dfratings <- subset(combinedData,select = c(1,2,3,5))
colnames(dfratings)
```

```
## [1] "movieId" "userId"  "rating"  "title"
```

```r
myratings <- as.data.frame(dfratings)
```

```r
kable(summary(dfratings))
```

movieId	userId	rating	title
Min. : 1	Min. : 1	Min. :0.500	Length:100004
1st Qu.: 1028	1st Qu.:182	1st Qu.:3.000	Class :character
Median : 2406	Median :367	Median :4.000	Mode :character
Mean : 12549	Mean :347	Mean :3.544	NA
3rd Qu.: 5418	3rd Qu.:520	3rd Qu.:4.000	NA
Max. :163949	Max. :671	Max. :5.000	NA

The ratings run from 0.5 to 5.

```r
# Filter by minimum movies watched and minimum users per movie
myratings <- myratings %>%
  group_by(movieId) %>%
  filter(n()>100) %>%
  group_by(userId) %>%
  filter(n()>50)

# Generating the user-item matrix for the predictor
ratingdcast <- dcast(myratings, userId~movieId,
                     value.var = "rating", fill=0, fun.aggregate = mean)

# Filling in rownames
rownames(ratingdcast) = ratingdcast$userId

# Removing the first column
ratingdcast <- as.matrix(ratingdcast[,-1])
# Converting to a matrix
ratingdcast <- as.matrix(ratingdcast)
```

Copy myratings into Spark and return an R object wrapping the copied object(a spark dataframe)

```r
colnames(myratings)
```

```
## [1] "movieId" "userId"  "rating"  "title"
```

```r
#which_train <- sample(x=c(TRUE,FALSE), size = nrow(myratings), #replace=TRUE,prob=c(0.8,0.2))
##trainData <- myratings[which_train,]
#testData <- myratings[!which_train,]
#head(trainData)
```

```r
## seelct 3 columns here
# copy the table to Spark
movie.tbl <- sdf_copy_to(sc, myratings, overwrite = TRUE)
#movie.tbl
#movie.tbl <- sdf_copy_to(sc, trainData, overwrite = TRUE)


mv <- movie.tbl %>% filter(userId == 428)
kable(head(mv,2))
```

```
## Warning: Translator is missing window functions:
## cor, count, cov, n_distinct, sd

## Warning: Translator is missing window functions:
## cor, count, cov, n_distinct, sd
```

movieId	userId	rating	title
1	428	5	Toy Story (1995)
2	428	3	Jumanji (1995)
SVD was at	tempted t	o impleme	nt, but due to errors, switched to ALS instead.

This SVD command gave a sparkException for the Java.lang.OutofMemoryError in heap space. "Error: org.apache.spark.SparkException: Job aborted due to stage failure: Task 0 in stage 18.0 failed 1 times, most recent failure: Lost task 0.0 in stage 18.0 (TID 34, localhost, executor driver): java.lang.OutOfMemoryError: Java heap space at java.nio.HeapByteBuffer.(HeapByteBuffer.java:57)"

So chose Alternating Least Squares(ALS) to perform matrix factorization on a Spark Dataframe.

Create an ALS model:

```r
# https://github.com/rstudio/sparklyr/blob/master/man/ml_als_factorization.Rd
# https://rdrr.io/cran/sparklyr/man/ml_als_factorization.html

# Measure time
tic()
MLSmodel <- ml_als_factorization(movie.tbl, user.column = "userId",item.column = "movieId",
                                 rating.column = "rating", iter.max = 7)

exectime <- toc()
```

```
## 2.938 sec elapsed
```

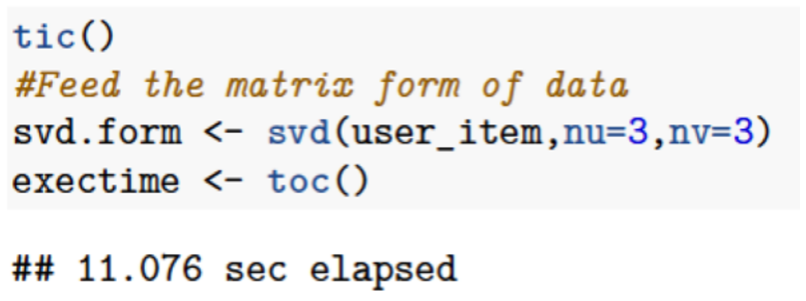
```r
exectimeALS <- exectime$toc - exectime$tic

summary(MLSmodel)
```

```
##                  Length Class      Mode
## item.factors     11     data.frame list
## user.factors     11     data.frame list
## data              2     spark_jobj environment
## ml.options        6     ml_options list
## model.parameters  2     -none-     list
## .call             6     -none-     call
## .model            2     spark_jobj environment
```

It took ~2 seconds in Spark. In R, UBCF needed ~9 seconds and IBCF needed ~89 seconds to evaluate the models.

When comparing SVD, the performance of the ALS model performed 5 times faster than R(2.8 seconds versus 11.076 seconds)



```
tic()
#Feed the matrix form of data
svd.form <- svd(user_item,nu=3,nv=3)
exectime <- toc()

## 11.076 sec elapsed
```

Fig 2: Execution time for SVD calculated in Project 3

Figure 2:

Using documentation from
https://spark.apache.org/docs/latest/ml-collaborative-filtering.html, the following predictions were created.

```
predictions <- MLSmodel$.model %>%
  invoke("transform", spark_dataframe(movie.tbl)) %>%
  collect()

#predictions[predictions$userId == 8,]
#dim(predictions)

pred_RMSE <- sqrt(mean(with(predictions, prediction-rating)^2))
pred_RMSE
```

```
## [1] 0.6364164
```

The RMSE for the ALS model is calculated and it is : 0.6093105

Comparing the ALS RMSE to the RMSE calculated from R irlba/svd in project 3, it can be seen that there is a great improvement in RMSE for ALS (0.6093105 vs. 3.43567)[Fig.2]

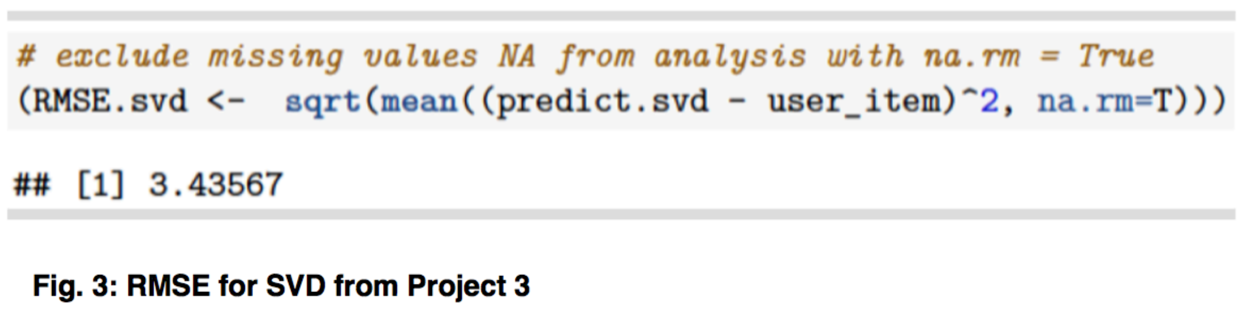
The executime time is very fast when running the recommender in Spark. R recommender took several minutes to run.

ALS is a method where the entire loss function is minimized at once, changing half the parameters at a time[Ref#:6}. So, half the parameters are fixed and the other half is recomputed and the process is repeated. ALS uncovers latent features.

Next, the matrix for the predictions from ALS is calculated. First two dataframes are created - one for userid and one for movieID w/title.

```
usernames <- myratings %>%
  distinct(userId) %>%
  arrange(userId)

userratings <- myratings %>%
  distinct(userId,rating) %>%
```



```
# exclude missing values NA from analysis with na.rm = True
(RMSE.svd <-  sqrt(mean((predict.svd - user_item)^2, na.rm=T)))

## [1] 3.43567
```

Fig. 3: RMSE for SVD from Project 3

Figure 3:

```
  arrange(userId,rating)

movienames <- myratings %>%
  group_by(title) %>%
  distinct(title)

u.df <- MLSmodel$user.factors[,-1]
m.df <- MLSmodel$item.factors[,-1]
u.matrix <- as.matrix(u.df)
m.matrix <- as.matrix(m.df)

# now predict
predict.ALS <- u.matrix %*% t(m.matrix)

rownames(predict.ALS) = usernames$userId
colnames(predict.ALS) = movienames$title

kable(predict.ALS[1:5, 1:5])
```

	Toy Story (1995)	Jumanji (1995)	Heat (1995)	GoldenEye (1995)	Leaving Las Vegas (1995)
4	4.611692	3.918420	4.250472	4.081249	4.050529
8	3.870543	3.187401	3.896780	3.353761	3.933417
15	2.831116	2.071788	4.040609	2.408394	2.906512
17	3.196486	2.395453	3.989754	2.751173	4.027206
19	3.608474	3.089160	3.551563	3.118113	3.430007

Performance testing:

```
predict.ALS.df <- as.data.frame(predict.ALS)
predict.ALS.df[5,1]
```

```
## [1] 3.608474
```

```
kable(predict.ALS.df[1:6, 1:6])
```

	Toy Story (1995)	Jumanji (1995)	Heat (1995)	GoldenEye (1995)	Leaving Las Vegas (1995)	Twelve Monkeys (
4	4.611692	3.918420	4.250472	4.081249	4.050529	
8	3.870543	3.187401	3.896780	3.353761	3.933417	
15	2.831116	2.071788	4.040609	2.408394	2.906512	
17	3.196486	2.395453	3.989754	2.751173	4.027206	
19	3.608474	3.089160	3.551563	3.118113	3.430007	
21	3.832396	2.952647	3.314175	2.826197	3.357965	

```
# remove dates and numbers from the movie name (easy to query without dates and numbers)
```

```
predict.ALS.df["15", "Toy Story "]
```

```
## [1] 2.831117
```

A confusion matrix was created to visualize the performance of the algorithm.
Choosing a threshold value of 3, the predictions and the ratings are partitioned.

```
# choose threshold value of 3
confusionMat <- predictions %>%
  mutate(actual = if_else(rating >= 3, 1, 0),
  predicted = if_else(prediction >= 3, 1, 0))

confusionMat <- subset(confusionMat,select = c(6,7))
#colnames(confusionMat)

confusionTable<- table(confusionMat)
confusionTable
```

```
##       predicted
## actual    0    1
##      0  874  446
##      1  638 9257
```

The diagonal represents the cases where the ratings are correctly predicted. Now the precision, recall and Fscores are calculated.
Precision is the ability of the classifier to not label as positive when its actually negative. In the formula $Precision : tp/(tp + fp)$ tp is the number of true positives and fp is the number of true negatives.
Recall is the ability of the classifier to find all the positive samples.
FScore is the weighted harmonic mean of the precision and recall, where the best score is at 1 and worse at 0.

```
# Precision: tp/(tp+fp):
(precision <- confusionTable[1,1]/sum(confusionTable[1,1:2]))
```

```
## [1] 0.6621212
```

```
# Recall: tp/(tp + fn):
(recall <- confusionTable[1,1]/sum(confusionTable[1:2,1]))
```

```
## [1] 0.5780423
```

```
# F-Score: 2 * precision * recall /(precision + recall):
(F_Score <- 2 * precision * recall / (precision + recall))
```

```
## [1] 0.6172316
```

To improve the FScore, a different threshold value (2) is chosen.

```
confusionMatLow <- predictions %>%
  mutate(actual = if_else(rating >= 2, 1, 0),
  predicted = if_else(prediction >= 2, 1, 0))



confusionMatLow <- subset(confusionMatLow,select = c(6,7))
colnames(confusionMatLow)
```

```
## [1] "actual"    "predicted"
```

```
confusionTableLow<- table(confusionMatLow)
confusionTableLow
```

```
##        predicted
## actual     0     1
##      0    97   300
##      1    28 10790
```

```
# Precision: tp/(tp+fp):
(precisionL <- confusionTableLow[1,1]/sum(confusionTableLow[1,1:2]))
```

```
## [1] 0.2443325
```

```
# Recall: tp/(tp + fn):
(recallL <- confusionTableLow[1,1]/sum(confusionTableLow[1:2,1]))
```

```
## [1] 0.776
```

```
# F-Score: 2 * precision * recall /(precision + recall):
(F_ScoreL <- 2 * precisionL * recallL / (precisionL + recallL))
```

```
## [1] 0.3716475
```

The F-Score got worse, so the accuracy of this recommender is not too great.

Now disconnect from spark gracefully.

```
 # disconnect from Spark
spark_disconnect(sc)
```

The results are tabulated and it can be seen that Spark is much faster, eventhough the RMSE didn't improve much.

```
ubcf <- c('0.938', '9')
ibcf <-c('1.067', '85')
svd <- c('3.43','9')
als <- c('3.12','2')

myresults.df <- data.frame(ubcf,ibcf,svd,als)
str(myresults.df)
```

```
## 'data.frame':    2 obs. of  4 variables:
##  $ ubcf: Factor w/ 2 levels "0.938","9": 1 2
##  $ ibcf: Factor w/ 2 levels "1.067","85": 1 2
##  $ svd : Factor w/ 2 levels "3.43","9": 1 2
##  $ als : Factor w/ 2 levels "2","3.12": 2 1
```

```
colnames(myresults.df) <- c("UBCF","IBCF","SVD","ALS")
rownames(myresults.df) <- c("RMSE-Distance","Executime Time(secs)")

kable(myresults.df,  type = "html",caption="Results - R vs Spark")
```

Table 5: Results - R vs Spark

	UBCF	IBCF	SVD	ALS
RMSE-Distance	0.938	1.067	3.43	3.12
Executime Time(secs)	9	85	9	2

```
x <- c("R is a very familiar language","has negligible learning curve", "Most datasets fit in memory","
y <- c("Spark is unfamiliar to programmers","has a teep learning curve","is scalable to handle big data
tradeoffs.df <- data.frame(x,y)
str(tradeoffs.df)
```

```
## 'data.frame':    7 obs. of  2 variables:
##  $ x: Factor w/ 7 levels "Abundance of help in google search",..: 5 3 4 1 7 2 6
##  $ y: Factor w/ 7 levels "Accuracy is not good",..: 7 2 3 4 5 1 6
```

```
rownames(tradeoffs.df) <- c("1","2","3","4","5","6","7")
colnames(tradeoffs.df) <- c("R","Spark")

kable(tradeoffs.df,  type = "html",caption="Comparison: R vs. Spark" )
```

Table 6: Comparison: R vs. Spark

R	Spark
R is a very familiar language	Spark is unfamiliar to programmers
has negligible learning curve	has a teep learning curve
Most datasets fit in memory	is scalable to handle big data
Abundance of help in google search	Not much help on google search
Takes a lot of time to process large samples in the dataset.	runs on multi threads, so its faster even for bigger datasets
Accuracy in prediction is good	Accuracy is not good
Slow executime time for large datasets	Spark has a super fast execution time for massive datasets.

Conclusion:

R is single threaded and comparing its performance with Spark is not a fair one. R does better with smaller samples of dataset. With its enormous statistical computation and visual libraries, R has a lot to offer. And its easier to test and verify results. But in the past few projects, the biggest frustration was with the time it took for the R program to run for 1M+ datasets. It was several hours each time to run some of the commands. However, it can be argued that using a small sample of dataset is not representative of the whole population in real world scenarios. So R may compromise accuracy. Massive datasets are ideal for Spark mllib and with R tools in Spark, it can be used to explore datasets on distributed systems. In the end, increased productivity is a huge deal for businesses with massive datasets and utilizing Spark will help in that area. So, if the dataset is below 1M rows, then R is a better choice with its abundance of tools.

This project was mostly comprised of installation of distributed systems on a single node, working on Databricks on top of Amazon AWS and researching several errors when making the recommender to work on Spark. The mllib in Spark supports collaborative filtering, where users and the movies are described by a small set of latent factors(used to predict the missing entries). Spark's mllib uses ALS with its parameter, lambda, scaled in solving each least squares problem and (Ref#.8) so lambda is less dependent on the dataset scale. So a smilar performance can be expected in a large dataset. It is worth the learning time to get comfortable with Spark in a distrubuted environment, so that the data science skills fit the business requirements.

References:

Ref#1: http://www.lyzander.com/r/spark/2016/11/26/spark_and_r
Ref#2: https://blog.rstudio.org/author/javierrstudiocom/
Ref#3: http://spark.rstudio.com/h2o.html
Ref#4: https://github.com/rstudio/sparklyr/blob/master/README.md
Ref#5: http://grouplens.org/datasets/movielens Ref#6: https://www.quora.com/What-is-the-Alternating-Least-Squares-m
Ref#7: http://spark.rstudio.com/h2o.html

Ref#8: https://spark.apache.org/docs/latest/mllib-collaborative-filtering.html
Ref#9: https://github.com/rstudio/sparklyr/blob/master/man/ml_als_factorization.Rd
Ref#10: https://rdrr.io/cran/sparklyr/man/ml_als_factorization.html