Developer Network

Developing Analytics Solutions with the Data & AI Global Practice

Scaling a recommender system across large data volumes

★★★★★

August 8, 2016 by Yaswant.v // 0 Comments

By Michele Usuelli, Data Scientist Consultant

Building a recommendation engine in presence of large data volume

E-commerce businesses can suggest new products to their customers. How do they choose the products to recommend? The companies collect data about the purchases of their customers. Starting from the purchase history, they can identify items that have been taken by the same customers and customers that bought the same items. Given a new customer, the company can recommend him/her items similar to the ones that he/she purchased already, or items that have been purchased by similar customers. The recommendations can also take account of other data sources like personal information, items descriptions, customer ratings and reviews. Large companies like Amazon and eBay already built sophisticated recommendation engines tailored to their environment.

What if a smaller company wants to build a smaller recommendation engine from scratch? Even if the complexity is not the same as huge companies, the volume of data to process may be large. In this article, we see how to build simple recommender systems that are scalable across large data volumes. To show the concepts avoiding unnecessary complications, in this article we are using the information about purchases only. Due to the complexity of the problem and to the variety of contexts, this article describes it from a generic high-level perspective.

Choosing the environment

In this article, the technology that we use is SQL Server 2016 with R Services. The reason is that we can store large SQL tables and process them using advanced analytics techniques provided by R.

The use-case consists in recommending movies to the users. Given a set of movies and users, the table describes what movies each user has watched. To avoid further complications, the table doesn't include user ratings. Its structure is:

- A row for each movie
- A column for each user
- Cells equal to 1 if the user watched the movie and to 0 otherwise

The data has already been imported into SQL 2016 into a table called *movies*. To analyse the data, we use the R environment and we define a SQL table object called *sql_movies* that allows us to access the data. The function creating the object is *RxSqlServerData* and it needs a connection string for the database and the name of the table. We also define another SQL table object, called *sql_clust_users*, referring to a table that doesn't exist yet. We'll create and use this table from R.

```
connection_string <- "Driver=SQL
Server;Server=.;Database=movies;Uid=usernam;password"

sql_movies <-
RxSqlServerData(connectionString = connection_string,
table ="movies")

sql_clust_users <-
RxSqlServerData(connectionString = connection_string,
table ="sql_clust_users")
```

Let's have a quick look at the data.

```
dim(sql_movies)

## [1] 11503 865

rxGetVarInfo(sql_movies)[1:4]

## $MovieId
## Type: integer, Low/High: (91, 3630218)
##
## $U9928
## Type: numeric, Low/High: (0.0000, 1.0000)
##
## $U15152
## Type: numeric, Low/High: (0.0000, 1.0000)
##
## $U7850
## Type: numeric, Low/High: (0.0000, 1.0000)
```

```
head(sql_movies)[, 1:8]

## MovieId U9928 U15152 U7850 U9065 U8166 U5251 U10249
## 1 368688 0 0 1 0 0 0 0
## 2 368709 0 0 0 0 0 0 0
## 3 368794 0 0 0 0 0 0 0
## 4 368891 0 0 0 0 1 0 0
## 5 368909 0 0 1 0 0 0 0
## 6 368933 0 0 0 0 0 0 0
```

The table has 865 columns. The first column contains the Movie ID and each other column corresponds to a user. The values in the *user* columns are equal to 1 if the user watched the movie and to 0 otherwise.

To build a recommendation engine, we can use the R package *recommenderlab*. However, this package needs all the data to be stored into the RAM, so it can deal with relatively small data volumes. Therefore, to process a large dataset we need to reduce its volume. In this example, the data volume is not really huge. However, the solution is designed in such a way that it's applicable on a larger dataset.

Reducing the data volume

To produce a small table, we can either reduce the number of rows or the number of columns. In this chapter we are reducing both.

A common challenge is that there are many users, i.e. columns. A solution consists in identifying small groups of similar users and defining a new table having a column for each group of users. The steps are

1. Define a formula describing what columns we are dealing with. We include all the columns apart from the first
2. Measure the similarity between users using the function rxCor
3. Starting from the similarity, define a matrix containing the distance between the users
4. Starting from the distance matrix, identify groups of similar users applying hierarchical clustering
5. Define a new table which columns correspond to the clusters. The value of each column is equal to one if at least one user of the cluster watched the movie

The new table contains a row for each movie and a column for each group of users.

This is the related code to detect 100 clusters of users.

```
# define the formula
cols_movies <-
names(sql_movies)[-1]
string_users <-
```

```
paste(cols_movies, collapse = " + ")
formula_users <-
formula(paste("~", string_users))

# build the hierachical clustering model
cor_users <-
rxCor(formula_users, sql_movies,
rowsPerRead =
5e02)

## Rows Read: 11503, Total Rows Processed: 11503, Total Chunk Time: 3.424 seconds
## Computation time: 7.211 seconds.

dist_users <-
as.dist(1 - abs(cor_users))
model_hc <-
hclust(dist_users, method = "complete")
which_cluster <-
cutree(model_hc, k =
100)
var_clusters <-
sort(unique(which_cluster))

# group the users into clusters
clusterCols <-
function(list_in) {

# define a field for each cluster
list_out <-
lapply(var_clusters, function(this_cluster) {
these_cols <-
names(which_cluster[which_cluster == this_cluster])
df_cols <-
data.frame(list_in[these_cols])
rowSums(df_cols) >
0
})

names(list_out) <-
paste0("clust", var_clusters)
# for each movie, count the number of clusers having watched it
df_views <-
data.frame(list_out)
list_out$n_views <-
rowSums(df_views)
```

```
# add the movie ID
list_out$MovieId <-
list_in$MovieId

list_out
}

rxDataStep(
inData = sql_movies,
outFile = sql_clust_users,
overwrite =
TRUE,
transformFunc = clusterCols,
transformObjects =
list(var_clusters = var_clusters,
which_cluster = which_cluster))

## Warning in rxLinkTransformComponents(transforms = transforms, transformFunc =
transformFunc, :
## There are 'transformObjects' that have the same name as an
## object in the evaluation environment, which could cause
## misleading results. Try renaming the following objects in the
## 'transformObjects' list: 'var_clusters', 'which_cluster'.

## Rows Read: 11503, Total Rows Processed: 11503, Total Chunk Time: 2.088 seconds

dim(sql_clust_users)

## [1] 11503 102
```

The other approach to decrease the size is to reduce the number of rows. Applying a clustering algorithm, we can identify groups of similar items and define a table with a row for each group of items. For this purpose, we can use the function *rxKmeans*.

The starting point is *sql_clust_users* and the steps are

1. Build a formula defining the variables to include into the clustering model
2. Identify 20 clusters of movies using *rxKmeans*
3. Extract the centers of the clusters. They contain averages that in this context express the percentage of users that watched each movie
4. Starting from the centers, define the table *df_movies* which values are 1 if the percentage is above 10% and 0 otherwise

This is the related code.

```
cols_km <-
names(sql_clust_users)
feat_km <-
paste(cols_km, collapse = " + ")
formula_km <-
formula(paste("~", feat_km))
model_km <-
rxKmeans(formula_km, sql_clust_users,
numClusters =
20,
reportProgress =
0)
df_movies <-
ifelse(model_km$centers[, -1] > 0.1, 1, 0)
df_movies[1:10, 1:6]

## clust1 clust2 clust3 clust4 clust5 clust6
## 1 1 0 0 1 1 1
## 2 0 0 0 0 1 0
## 3 1 0 1 1 1 1
## 4 0 0 0 0 0 0
## 5 0 0 0 0 1 1
## 6 1 0 1 0 1 1
## 7 1 0 0 1 1 1
## 8 0 0 0 1 1 1
## 9 0 0 0 0 1 1
## 10 1 0 0 1 1 1
```

The table *df_movies* is small-sized and it contains a column for each cluster of users and a row for each cluster of movies. The value of each cell is equal to 1 if the related cluster of users watched at least 10% of the related cluster of movies, and 0 otherwise.

Building the engine

In the previous step we produced a small table stored into the R environment. Starting from that, we can build a recommender system using the package *recommenderlab*.

The steps are

1. Starting from *df_movies*, build, test and validate a recommendation engine. There are a few options in the recommenderlab package, so a good approach is to test and evaluate a few of them, and to pick the best-performing
2. Given a new user, measure the distance between each cluster center and he/she. Then, identify the closest center and associate the user to the related cluster
3. Identify the cluster of movies to recommend to the cluster related to the new user

4. Out of the recommended cluster of movies, choose one or more movies to recommend to the user

Conclusions

Using the approach described by this article, it's possible to apply a recommender system on a large data volume.

To extend this solution, it's possible to use a matrix containing user ratings instead of just 0s a 1s. To include some information about the users and/or movies, it's possible to summarise it for each cluster. Other adjustments can follow a similar methodology.

Popular Tags

#ArtificialIntelligence #DataScience AI AML Analytics Apache Spark APS Artificial Intelligence Azure Data Factory Azure Key Vault AzureML Azure SQL Data Warehouse Basket analysis Center of Excellence Centre of Excellence CoE Data Data Science Data Scientist Role Decision forests Event Hubs k-fold Machine Learning Measurability Model Goodness Modelling Power BI PowerShell ML Scoring R Random Projection R Services Scikit SQL Server 2016 SQL Server R Services Stream Analytics Visualizations Whitepaper

Archives

November 2018 (1)
All of 2018 (4)
All of 2017 (3)
All of 2016 (11)
All of 2015 (11)

Tags R Services SQL Server 2016

Join the conversation

Add Comment

Dev centers

Learning resources

Microsoft Virtual Academy