HANDS-ON RECOMMENDER SYSTEM ON PYSPARK

PySpark Collaborative Filtering with ALS

Understand data munging in PySpark while building a recommender system that utilises matrix factorisation technique — Alternating Least Squares (ALS)

Snehal Nair Aug 10, 2020 · 11 min read ★

Photo by Nathan Dumlao (https://unsplash.com/@nate_dumlao)

Recommender System is an information filtering tool that seeks to predict which product a user will like, and based on that, recommends a few products to the users. For example, Amazon can recommend new shopping items to buy, Netflix can recommend new movies to watch, and Google can recommend news that a user might be interested in. The two widely used approaches for building a recommender system are the content-based filtering (CBF) and collaborative filtering (CF).

To understand the concept of recommender systems, let us look at an example. The below table shows the user-item *utility matrix* Y where the value Rui denotes how item i has been rated by user u on a scale of 1–5. The missing entries (shown by **?** in Table) are the items that have not been rated by the respective user.

The objective of the recommender system is to predict the ratings for these items. Then the highest rated items can be recommended to the respective users. In real world problems, the utility matrix is expected to be very sparse, as each user only encounters a small fraction of items among the vast pool of options available. The code for this project can be found here.

Explicit v.s. Implicit ratings

There are two ways to gather user preference data to recommend items, the first method is to ask for **explicit ratings** from a user, typically on a concrete rating scale (such as rating a movie from one to five stars) making it easier to make extrapolations from data to predict future ratings. However, the drawback with explicit data is that it puts the responsibility of data collection on the user, who may not want to take time to enter ratings. On the other hand, **implicit data** is easy to collect in large quantities without any extra effort on the part of the user. Unfortunately, it is much more difficult to work with.

Data Sparsity and Cold Start

In real world problems, the utility matrix is expected to be very sparse, as each user only encounters a small fraction of items among the vast pool of options available. Cold-Start problem can arise during addition of a new user or a new item where both do not have history in terms of ratings. Sparsity can be calculated using the below function.

1. Dataset with Explicit Ratings (MovieLens)

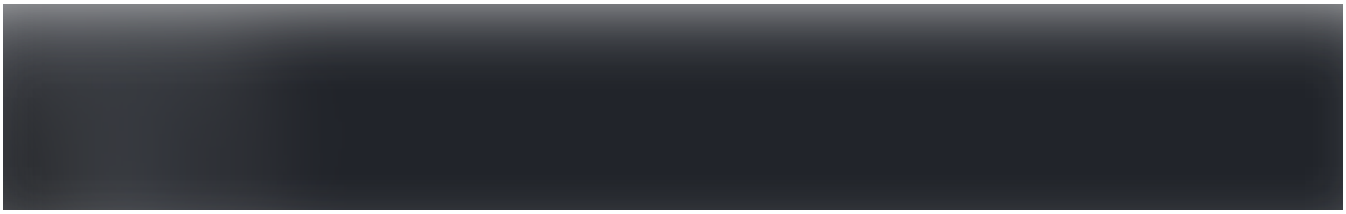
MovieLens is a recommender system and virtual community website that recommends movies for its users to watch, based on their film preferences using collaborative filtering. MovieLens 100M datatset is taken from the MovieLens website, which customizes user recommendation based on the ratings given by the user. To understand the concept of recommendation system better, we will work with this dataset. This dataset can be downloaded from here.

There are 2 tuples, movies and ratings which contains variables such as MovieID::Genre::Title and UserID::MovieID::Rating::Timestamp respectively.

Let's load the data and explore the data. To load the data as a spark dataframe, import pyspark and instantiate a spark session.

```
from pyspark.sql import SparkSession
spark =
SparkSession.builder.appName('Recommendations').getOrCreate()

movies = spark.read.csv("movies.csv",header=True)
ratings = spark.read.csv("ratings.csv",header=True)
ratings.show()
```



```
# Join both the data frames to add movie data into ratings
movie_ratings = ratings.join(movies, ['movieId'], 'left')
movie_ratings.show()
```

Ratings and movie data combined for better visualisation

Let's calculate the data sparsity to understand the sparsity of the data. Please function that we built in the beginning of this article to get the sparsity. The movie lens data is 98.36% sparse.

```
get_mat_sparsity(ratings)
```

Before moving into recommendations, split the dataset into train and test. Please set the seed to reproduce results. We will look at different recommendation techniques in detail in the below sections.

```
# Create test and train set
(train, test) = ratings.randomSplit([0.8, 0.2], seed = 2020)
```

2. Dataset with Binary Ratings (MovieLens)

With some datasets, we don't have the luxury to work with explicit ratings. For those datasets we must infer ratings from the given information. In MovieLens dataset, let us add implicit ratings using explicit ratings by adding 1 for watched and 0 for not watched. We aim the model to give high predictions for movies watched.

```
user_movie.show()
```

based filtering (CBF) and collaborative filtering (CF), of which CBF is the most widely used.

The below figure illustrates the concepts of CF and CBF. The primary difference between these two approaches is that CF looks for similar users to recommend items while CBF looks for similar contents to recommend items.

Content-based Filtering (CBF)

The main idea behind CBF is to recommend items similar to the items previously liked by the user. For example, if the user have rated some items in the past, then these items are used for *user-modeling* where the user's interests are quantified. Traditionally, the item iis represented by a *feature vector* xi, which can be boolean or real valued, and the user is represented by a weight vector ✓u of same dimension. Given a new item x⇤, represented in the same feature vector space, the likeliness, e.g., rating of the item is predicted using the user model.

This can be achieved in two different ways:

• Predicting ratings using parametric models like regression or logistic regression for multiple ratings and binary ratings respectively based on the previous ratings.

CB can be applied even when a strong user-base is not built, as it depends on the item's meta data (features) therefore does not suffer from cold-start problem. However, this also makes it computationally intensive, as similarities between each user and all the items must be computed. Since the recommendations are based on the item similarity to the item that the user already knows about, it leaves no room for serendipity and causes over specialisation. CB also ignores popularity of an item and other users feedbacks.

Collaborative filtering (CF)

Collaborative filtering aggregates the past behaviour of all users. It recommends items to a user based on the items liked by another set of users whose likes (and dislikes) are similar to the user under consideration. This approach is also called the *user-user* based CF.

CF and item-item CF can be achieved by two different ways, **memory-based** (neighbourhood approach) and **model-based** (latent factor model approach).

1. The memory-based approach

Neighbourhood approaches are most effective at detecting very localized relationships (neighbours), ignoring other users. But the downsides are that, first, the data gets sparse which hinders scalability, and second, they perform poorly in terms of reducing the RMSE (root-mean-squared-error) compared to other complex methods. User-based Filtering and Item-based Filtering are the two ways to approach memory-based collaborative filtering.

User-based Filtering: To recommend items to user u1 in the user-user based neighborhood approach first a set of users whose likes and dislikes similar to the useru1 is found using a similarity metrics which captures the intuition that sim(u1, u2) >sim(u1, u3) where user u1 and u2 are similar and user u1 and u3 are dissimilar. similar user is called the neighbourhood of user u1.

Item-based Filtering: To recommend items to user u1 in the item-item based neighborhood approach the similarity between items liked by the user andother items are calculated.

profiles (both of dimension K) through matrix factorization by minimizing the RMSE (Root Mean Square Error) between the available ratings yand their predicted values yˆ. Here each item i is associated with a latent (feature) vector xi, each user u is associated with a latent (profile) vector theta(u), and the rating yˆ(ui) is expressed as

$$\hat{y}_{ui} = \mu + b_u + b_i + \theta_u^\top \mathbf{x}_i$$

Image Source: https://developers.google.com/machine-learning/recommendation/collaborative/matrix

Latent methods deliver prediction accuracy superior to other published CF techniques. It also addresses the sparsity issue faced with other neighbourhood models in CF. The memory efficiency and ease of implementation via gradient based matrix factorization model (SVD) have made this the method of choice within the Netflix Prize competition. However, the latent factor models are only effective at estimating the association between all items at once but fails to identify strong association among a small set of closely related items.

Recommendation using Alternating Least Squares (ALS)

Alternating Least Squares (ALS) matrix factorisation attempts to estimate the ratings matrix R as the product of two lower-rank matrices, X and Y, i.e. X * Yt = R. Typically these approximations are called 'factor' matrices. The general approach is iterative. During each iteration, one of the factor matrices is held constant, while the other is solved for using least squares. The newly-solved factor matrix is then held constant while solving for the other factor matrix.

In the below section we will instantiate an ALS model, run hyperparameter tuning, cross validation and fit the model.

are looking at rating greater than 0. The model also gives an option to select implicit ratings. Since we are working with explicit ratings, set it to '*False*'.

When using simple random splits as in Spark's `CrossValidator` or `TrainValidationSplit`, it is actually very common to encounter users and/or items in the evaluation set that are not in the training set. By default, Spark assigns `NaN` predictions during `ALSModel.transform` when a user and/or item factor is not present in the model.*We set cold start strategy to 'drop' to ensure we don't get NaN evaluation metrics*

```
# Import the required functions
from pyspark.ml.evaluation import RegressionEvaluator
from pyspark.ml.recommendation import ALS
from pyspark.ml.tuning import ParamGridBuilder, CrossValidator

# Create ALS model
als = ALS(
         userCol="userId",
         itemCol="movieId",
         ratingCol="rating",
         nonnegative = True,
         implicitPrefs = False,
         coldStartStrategy="drop"
)
```

2. Hyperparameter tuning and cross validation

```
# Import the requisite packages
from pyspark.ml.tuning import ParamGridBuilder, CrossValidator
from pyspark.ml.evaluation import RegressionEvaluator
```

ParamGridBuilder: We will first define the tuning parameter using param_grid function, please feel free experiment with parameters for the grid. I have only chosen 4 parameters for each grid. This will result in 16 models for training.

```
# Add hyperparameters and their respective values to param_grid
param_grid = ParamGridBuilder() \
            .addGrid(als.rank, [10, 50, 100, 150]) \
```

RegressionEvaluator: Then define the evaluator, select rmse as metricName in evaluator.

```
# Define evaluator as RMSE and print length of evaluator
evaluator = RegressionEvaluator(
           metricName="rmse",
           labelCol="rating",
           predictionCol="prediction")
print ("Num models to be tested: ", len(param_grid))
```

CrossValidator: Now feed both param_grid and evaluator into the crossvalidator including the ALS model. I have chosen number of folds as 5. Feel free to experiement with parameters.

```
# Build cross validation using CrossValidator
cv = CrossValidator(estimator=als, estimatorParamMaps=param_grid,
evaluator=evaluator, numFolds=5)
```

4. Check the best model parameters

Let us check, which parameters out of the 16 parameters fed into the crossvalidator, resulted in the best model.

```
print("**Best Model**")

# Print "Rank"
print("  Rank:", best_model._java_obj.parent().getRank())

# Print "MaxIter"
print("  MaxIter:", best_model._java_obj.parent().getMaxIter())

# Print "RegParam"
print("  RegParam:", best_model._java_obj.parent().getRegParam())
```

```
**Best Model**
  Rank: 16
  MaxIter: 10
  RegParam: 0.16
```

the range of parameters chosen we are testing 16 models, so this might take while.

```
#Fit cross validator to the 'train' dataset
model = cv.fit(train)

#Extract best model from the cv model above
best_model = model.bestModel

# View the predictions
test_predictions = best_model.transform(test)

RMSE = evaluator.evaluate(test_predictions)
print(RMSE)
```

The RMSE for the best model is 0.866 which means that on average the model predicts 0.866 above or below values of the original ratings matrix. Please note, matrix factorisation unravels patterns that humans cannot, therefore you can find ratings for a few users are a bit off in comparison to others.

4. Make Recommendations

Lets go ahead and make recommendations based on our best model. recommendForAllUsers(n) function in als takes n recommedations. Lets go with 5 recommendations for all users.

```
# Generate n Recommendations for all users
recommendations = best_model.recommendForAllUsers(5)
recommendations.show()
```

```
+------+--------------------+
|userId|     recommendations|
+------+--------------------+
|   471|[[3379, 4.816196]...|
|   463|[[3379, 4.994439]...|
|   496|[[3379, 4.5707183...|
|   148|[[33649, 4.504622...|
|   540|[[3379, 5.4063077...|
|   392|[[3379, 4.65679],...|
|   243|[[3379, 5.7590547...|
|    31|[[3379, 5.100943]...|
|   516|[[4429, 4.8041797...|
|   580|[[3379, 4.756224]...|
+------+--------------------+
```

the above the output, the recommendations are saved in an array format with movie id and ratings. To make these recommendations easy to read and compare t check if recommendations make sense, we will want to add more information like movie name and genre, then explode array to get rows with single recommendations.

```
nrecommendations = nrecommendations\
    .withColumn("rec_exp", explode("recommendations"))\
    .select('userId', col("rec_exp.movieId"), col("rec_exp.rating"))

nrecommendations.limit(10).show()
```

```
+------+-------+---------+
|userId|movieId|   rating|
+------+-------+---------+
|   471|   3379| 4.816196|
|   471|   8477|4.6407275|
|   471|  33649|4.5224075|
|   471| 171495|4.4964633|
|   471|  86781| 4.482169|
|   463|   3379| 4.994439|
|   463| 131724|4.7216597|
|   463|  33649| 4.709987|
|   463| 171495| 4.703137|
|   463|  78836|4.6359744|
+------+-------+---------+
```

Do the recommendations make sense?

To check if the recommendations make sense, join movie name and genre to the above table. Lets randomly pick 100th user to check if the recommendations make sense.

100th User's ALS Recommendations:

```
nrecommendations.join(movies, on='movieId').filter('userId =
100').show()
```

```
+-------+------+---------+--------------------+--------------------+
|movieId|userId|   rating|               title|              genres|
+-------+------+---------+--------------------+--------------------+
|  67618|   100|5.1405735|Strictly Sexual (...|Comedy|Drama|Romance|
|   3379|   100| 5.006642| On the Beach (1959)|               Drama|
|  33649|   100|5.0065255|  Saving Face (2004)|Comedy|Drama|Romance|
|  42730|   100|4.9775596|   Glory Road (2006)|               Drama|
|  74282|   100| 4.915519|Anne of Green Gab...|Children|Drama|Ro...|
```

```
|   7071|   100| 4.874221|Woman Under the T...|               Drama|
| 117531|   100| 4.874221|    Watermark (2014)|         Documentary|
+-------+------+---------+--------------------+--------------------+
```

100th User's Actual Preference:

```
ratings.join(movies, on='movieId').filter('userId =
100').sort('rating', ascending=False).limit(10).show()
```

```
+-------+------+------+--------------------+--------------------+
|movieId|userId|rating|               title|              genres|
+-------+------+------+--------------------+--------------------+
|   1101|   100|   5.0|      Top Gun (1986)|      Action|Romance|
|   1958|   100|   5.0|Terms of Endearme...|        Comedy|Drama|
|   2423|   100|   5.0|Christmas Vacatio...|              Comedy|
|   4041|   100|   5.0|Officer and a Gen...|       Drama|Romance|
|   5620|   100|   5.0|Sweet Home Alabam...|      Comedy|Romance|
|    368|   100|   4.5|     Maverick (1994)|Adventure|Comedy|...|
|    934|   100|   4.5|Father of the Bri...|              Comedy|
|    539|   100|   4.5|Sleepless in Seat...|Comedy|Drama|Romance|
|     16|   100|   4.5|       Casino (1995)|         Crime|Drama|
|    553|   100|   4.5|    Tombstone (1993)|Action|Drama|Western|
+-------+------+------+--------------------+--------------------+
```

The movie recommended to the 100th user primarily belongs to comedy, drama, war and romance genres, and the movies preferred by the user as seen in the above table, match very closely with these genres.

I hope you enjoyed reading. Please find the detailed codes in the Github Repository.

References:

http://spark.apache.org/docs/2.2.0/ml-collaborative-filtering.html

ALS Pyspark Collaborative Filtering Recommendation Engine Matrix Factorization