UNIVERSITÀ DEGLI STUDI DI MILANO
FACULTY OF SCIENCE AND TECHNOLOGY

DEPARTMENT OF COMPUTER SCIENCE

Master in
Computer Science

ALGORITHMS FOR MASSIVE DATASETS
FINAL REPORT ABOUT RECOMMENDER SYSTEM

Teacher: Prof. Dario Malchiodi

Final report written by:
Samuele Simone
Matr. Nr. 11910A

Academic year 2022-2023

I/We declare that this material, which I/We now submit for assessment, is entirely my/our own work and has not been taken from the work of others, save and to the extent that such work has been cited and acknowledged within the text of my/our work. I/We understand that plagiarism, collusion, and copying are grave and serious offences in the university and accept the penalties that would be imposed should I engage in plagiarism, collusion or copying. This assignment, or any part of it, has not been previously submitted by me/us or any other person for assessment on this or any other course of study.

Contents

The report is made up of 7 chapters based on this template[1] and adapted for our purpose. In the **Chapter 1** we give at the reader a general view of what is a recommender system and where we can find it. **Chapter 2**, we start to discuss about the development setup for the project. Then, in the **Chapter 3** we focus on the dataset, how is composed and we explore the data. Later, in the **Chapter 4** we explain the recommender system. the different approaches and the mechanisms behind. Then in the **Chapter 5** we enter deeper by showing the code of different approaches and some experiments. Some notion about scalability and complexity are given in the **Chapter 6**. In order to evaluate the project developed we are talking about some metrics like CG,NDCG. Consequently , we summarize the aspects and the results obtained during the various in the **Chapter 7**.

Chapter 1

Introduction

A recommender system is used everywhere nowadays. Indeed all big companies are pushing in these systems because they can increase the sells about their product, e.g., when we are scrolling a product on Amazon, then they show us a list of recommendation based on the item selected.

Recommendation system use a number of different technologies. We can split these system into two broad groups: [2]

- *Content-based systems*: examine properties of the items recommended. For instance, if a Netflix user has watched many cowboy movies, then recommend a movie classified in the database as having the "cowboy" genre.
- *Collaborative filtering* systems recommend items based on similarity measures between users and/or items.

Chapter 2

Environment setup

Before going deeper into the project we must discuss about the entire environment was setup. Indeed I used different libraries in order to create the recommender system. Here the snippet of all libraries used in the project:

```
#importing the required pyspark library
import pyspark
from pyspark import SparkConf
from pyspark.sql import SparkSession
from pyspark.ml.evaluation import RegressionEvaluator
from pyspark.ml.recommendation import ALS
from pyspark.sql.functions import split
from pyspark.sql.functions import array_contains
from pyspark.sql.functions import col
from pyspark.sql.functions import from_json
from pyspark.sql.functions import when
from pyspark.sql import functions as F
from pyspark.ml.feature import VectorAssembler
from pyspark.ml.feature import StringIndexer
from pyspark.ml.feature import MinMaxScaler
from pyspark.ml.feature import Normalizer
from pyspark.ml.linalg import Vectors
from pyspark.sql import Row
from pyspark.sql.window import Window
#import for manage global var
import os
#import for graphics
import matplotlib.pyplot as plt
import pandas as pd
#import for regular expression
import re
import numpy as np
from sklearn.metrics.pairwise import cosine_similarity
from sklearn.model_selection import train_test_split
from sklearn.neighbors import KNeighborsClassifier
```

```
31 from sklearn.model_selection import cross_val_score
32 from sklearn.metrics import accuracy_score
```

Listing 2.1: Loading all the python libraries

2.1 Pyspark setup

PySpark is the Python API for Apache Spark. It enables you to perform real-time, large-scale data processing in a distributed environment using Python. It also provides a PySpark shell for interactively analyzing your data.

PySpark combines Python's learnability and ease of use with the power of Apache Spark to enable processing and analysis of data at any size for everyone familiar with Python.

PySpark supports all of Spark's features such as Spark SQL, DataFrames, Structured Streaming, Machine Learning (MLlib) and Spark Core. [3]

Refering to the Listing 2.1 we can see that there are several libraries about PySpark. I will discuss about the most important:

- **SparkConf**: Configuration for a Spark application. Used to set various Spark parameters as key-value pairs. [4]
- **SparkSession**:The entry point to programming Spark with the Dataset and DataFrame API. [5]
- **pyspark.ml**: DataFrame-based machine learning APIs to let users quickly assemble and configure practical machine learning pipelines. [6]
- **pyspark.sql.functions**:A list of function that allow the user to explore dataframe [7]

In the project I used PySpark Dataframe. They are distributed collections of data that can be run on multiple machines and organize data into named columns. These dataframes can pull from external databases, structured data files or existing resilient distributed datasets (RDDs).

2.2 Kaggle setup

The data were hosted on the Kaggle platform. Kaggle is a subsidiary of Google, it is an online community of data scientists and machine learning engineers.

Kaggle allows users to find datasets they want to use in building AI models, publish datasets, work with other data scientists and machine learning engineers, and enter competitions to solve data science challenges.

Kaggle got its start in 2010 by offering machine learning and data science competitions as well as offering a public data and cloud-based business platform for data science and AI education. [8]

In order to access to the data Kaggle platform gives to every register account an API credential that is necessary to download the datasets.

```
os.environ['KAGGLE_USERNAME'] = 'your_kaggle_username'
os.environ['KAGGLE_KEY'] = 'your_kaggle_key'
```

Listing 2.2: API Credential for accessing the data

2.3 Pandas setup

Pandas is a Python package providing fast, flexible, and expressive data structures designed to make working with "relational" or "labeled" data both easy and intuitive. It aims to be the fundamental high-level building block for doing practical, real-world data analysis in Python. Additionally, it has the broader goal of becoming the most powerful and flexible open source data analysis/manipulation tool available in any language. It is already well on its way toward this goal. [9]

I used it to explore the data, as we will see in the chapter 3, and to build the content-based recommendation system. My goal was to implement the recommendation system with different types of approaches, and Pandas is one of them. However, in the course of the paper I will emphasize the different ways in which I tried my hand at it.

2.4 Scikit-learn setup

Scikit-learn is a library in Python that provides many unsupervised and supervised learning algorithms. It's built upon some of the technology like NumPy, pandas, and Matplotlib. The functionality that scikit-learn provides include [10]:

- Regression, including Linear and Logistic Regression
- Classification, including K-Nearest Neighbors
- Clustering, including K-Means and K-Means++
- Model selection
- Preprocessing, including Min-Max Normalization

Indeed some of these functionality are used in this project such as the K-NN, the second different approach to build the Content based recommender system.

2.5 Numpy setup

NumPy is the fundamental package for scientific computing in Python. It is a Python library that provides a multidimensional array object, various derived objects (such as masked arrays and matrices), and an assortment of routines for fast operations on arrays, including mathematical, logical, shape manipulation, sorting, selecting, I/O, discrete Fourier transforms, basic linear algebra, basic statistical operations, random simulation and much more. [11]

It was useful in constructing the function `cosine_similarity_scratch` to perform multiplication and normalization of vectors.

Chapter 3

Dataset: A look inside

3.1 Data loading

We start to look how the data is loaded inside the project. First of all the data is downloaded from Kaggle with this code:

```
!kaggle datasets download -d yelp-dataset/yelp-dataset
```

Listing 3.1: Download dateset

and proceeded to unzip the dataset obtaining:

```
Archive:  /content/yelp-dataset.zip
  inflating: yelp-dataset/Dataset_User_Agreement.pdf
  inflating: yelp-dataset/yelp_academic_dataset_business.json
  inflating: yelp-dataset/yelp_academic_dataset_checkin.json
  inflating: yelp-dataset/yelp_academic_dataset_review.json
  inflating: yelp-dataset/yelp_academic_dataset_tip.json
  inflating: yelp-dataset/yelp_academic_dataset_user.json
```

Listing 3.2: Unzip dateset

These json files are huge and to manage it we load the data into a Pyspark df with this command:

```
df_review = spark.read.json('/content/yelp-dataset/
    yelp_academic_dataset_review.json')
```

Listing 3.3: Loading json data into Pyspark df

We repeat this operation many time as the number of the json files. So we obtain these dataframe:

- **df_review**: (review_id,user_id,business_id,stars_review). These are the attributes I selected from the df. However, there is one field that I think should be mentioned which is the text field, very useful for doing further

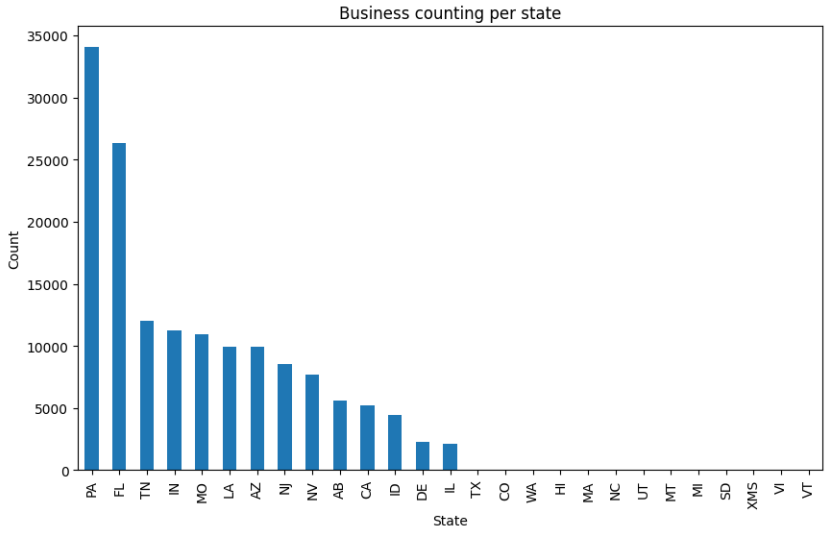
analysis. In my case it was not selected as I believe that the quoted data is sufficient for our purposes.

- **df_users**:(user_id,username,average_stars)
- **df_business**: (business_id,address,attributes,categories,city,is_open,name,stars,state). There are others column that I prefer to remove because is not adding values for our analysis.

3.2 Visualizing the data

To give a main idea of the data I create a chart where on the x-axes there are all the states and on the y-axes the count of the total business for that particular state. From the Figure 1 we can see that in "PA", we refer to Pennsylvania, a

Figure 1: Business counting per state



state located in the northeastern part of the United States. So I decided to conduct my analysis over the "PA" state due the amount of businesses present over there. Furthermore due the high dimension of the df it's difficult to create others type of graphics.

3.3 Data Filtering

As we discussed in the previous section 3.2 we want to targeting our recommender system in a specific area for a specific type of business. So I decided to take *"PA"* as

state and *"Restaurant"* as type of business. Moreover I filtered the new dataframe called `restaurant_df` with another parameter called `is_open` that when is equal to 1 means that this business is actually a running business.

```
# Filtering data
restaurant_df = df_business_pandas[(df_business_pandas['state'] ==
    state) & (df_business_pandas['is_open'] == 1) &
   df_business_pandas['categories'].str.contains(type_business)==
   True].reset_index()
```

Listing 3.4: Filtering Pyspark df

Here an example of `restaurant_df` printed: Note that the attribute and category

index	address	attributes	business_id	categories	city	is_open	name	stars	state
3	935 Race St	(None, None, 'full_bar', {'touristy': False, '...	MTSW4McQd7CbVtyjqoe9mw	Restaurants, Food, Bubble Tea, Coffee & Tea, B...	Philadelphia	1	St Honore Pastries	4.0	"PA"

Table 1: Example of a restaurant_df's row

columns must be preprocessed before being used.

3.4 Data preprocessing

Lets start to analyze the attribute column. As we can see they are in this form:

```
Row(AcceptsInsurance=None, AgesAllowed=None, Alcohol="u'none'",
   Ambience=None, BYOB=None,..)
```

Listing 3.5: Attributes item example

So we need a function that allow us to extract key-value pairs in a format understandable to Python. Here the snippet:

```
# Function for extracting key-value from attributes
def extract_values(row):
    if row is None or row == "{}":
        return {}

    attributes = {}
    pattern = r"(\w+)\s*=\s*([^,]+)"
    matches = re.findall(pattern, row)

    for key, value in matches:
        value = value.strip("'")
        attributes[key] = value

    return attributes
```

Listing 3.6: Extracting key-value from attributes

Table 2: Example of `restaurant_df_attr` elements

BestNights	RestaurantsCounterService	WiFi	...	Smoking	CoatCheck
0	1	0	...	0	0
1	0	1	...	1	1

To achieve the result I used the regular expression. More specifically the line 7 defines a regular expression pattern.The pattern captures two groups:

- the key (composed of one or more word characters \w+)
- the value (composed of one or more characters that are not a comma \[,^]+), separated by an equal sign with optional whitespace around it.

Then we apply the function over all the rows of column attributes. Now that the format seems to be good we proceed to create an entire `restaurant_df_attr` in which the attributes are extracted making a new column of the df and populating it with 0 or 1 based on their original value. An example are in the Table 2.

We apply this reasoning also for categories obtaining the `df_categories_dm`. It will be useful to create the `df_total`, a fundamental component for the recommender system. So in the Chapter 4 we are going to understand how recommendation systems work in theory.

Chapter 4

Recommender system

As we mentioned in the Chapter 1, there are two main basic architectures for a recommendation system. In my project I developed from scratch a *Content-Based* recommendation system. They are based on properties of items and the similarity of these are determined by measuring the similarity in their properties. However I also worked with the ALS as *Collaborative Filtering* using the PySpark library to compare mine with the one developed with the library. In this Chapter we give to the reader the general notion behind recommender system based on the reference book [2].

4.1 Content-Based Recommendations

We are starting to understand what is an item profiles. It's a record or collection of records representing important characteristics of the item. For example if we consider the restaurants as type of business of this project, we can say that the features of a restaurants can be all the attributes such as *GoodForKids,BusinessAcceptsBitcoin,DogsAllowed....* In order to be a recommender system we need also the user. Indeed also the users has their profiles that describes their preferences.

Another fundamental aspect is the utility matrix. It's giving for each user-item pair, a value that represents what is known about the degree of preference of that user for that item. Without a utility matrix, it is almost impossible to recommend items. Usually there are two main approaches:

- We can ask users to evaluate items. We fall in this case because the from the dataset `review` we can access to all user-business pair reviews with the respective rate.

- We can make inferences from user's behavior. Indeed if one user buy something online or just watched a video we can say that he liked it.

Our ultimate goal for content-based recommendation is to create both a item profile consisting of feature-value pairs and a user profile summarizing the preferences of the user,based of their row of the utility matrix.If we consider profiles as vectors we can compute the cosine similarity in the space of the features and understand if the vectors appears closer or not. In this way we can recommend business to the user based on his preferences.

4.2 Collaborative Filtering

When we are referring to Collaborative Filtering we are focusing on the similarity of the user ratings for two items. In place of the item-profile vector for an item, we use its column in the utility matrix. User are similar if their vectors are close according to some notion of distance measures such as cosine distance. The process of identifying similar user and recommending what similar users like is called collaborative filtering.

Chapter 5

Creation of a Content-based Recommender system

Before I talk about the content-based recommendation system that I created from scratch, I want to highlight some of the steps that allowed me to arrive at the solution or at least allowed me to learn more about the data I had at my disposal.

5.1 Business similarity suggestion with Pandas and cosine similarity

First of all I tried to create a suggestion system by looking the business similarity. For achieve this goal I used in this step Pandas. I performed this steps:

1. Extract the reference restaurant index
2. Mapping the rate (1-3 stars will be mapped in 0 and 4-5 in 1)
3. Create the `cosine_similarity_scratch` function

```
def cosine_similarity_scratch(vector1, vector2):
    dot_product = np.dot(vector1, vector2)
    norm_vector1 = np.linalg.norm(vector1)
    norm_vector2 = np.linalg.norm(vector2)

    if norm_vector1 == 0 or norm_vector2 == 0:
        return 0.0

    similarity = dot_product / (norm_vector1 * norm_vector2)
    return similarity
```

Listing 5.1: Cosine similarity function

4. Calculate business similarity respect the one selected. Order the similar restaurants in a descending way and then select n (= 5 in this case) top similar restaurant and print it.

```
restaurant_features = features_df.iloc[restaurant_index,
    6:-2].values  # Reference restaurant feature vector

similarities = []
for i in range(len(features_df)):
    if i != restaurant_index:
        other_restaurant_features = features_df.iloc[i,
    6:-2].values
        similarity = cosine_similarity_scratch(
    restaurant_features, other_restaurant_features)
        similarities.append(similarity)

similarities = np.array(similarities)
similar_restaurants = similarities.argsort()[::-1]  # Similar
    restaurant ordered in a desc way

# Visualize 5 top suggested restaurants with cosine sim
    column
top_similar_restaurants = similar_restaurants[:5]
recommended_restaurants = df_total.iloc[
    top_similar_restaurants][['name', 'stars']]
recommended_restaurants['cosine_similarity'] = similarities[
    top_similar_restaurants]

print(recommended_restaurants)
```

Listing 5.2: Suggestion system in Pandas

So now I will show the results obtained.

5.1.1 Experiment

I choose the `8071` restaurant index called `Adelita Taqueria & Restaurant`. It's a mexican restaurant located in 1108 S 9th St Philadelphia, PA 19147. In the note the link of the restaurant on Yelp.[1] The restaurant counts 39 reviews with an average rating of 4.5 stars.

The similar business that the system recommend are these reported in the Table 3 where there are linked their Yelp page. They are all mexican restaurant so it's perfectly matching our selected restaurant. I tried also to use a cosine similarity library from `from sklearn.metrics.pairwise import cosine_similarity` and the results are quite similar as shown in Table 4.

[1] https://www.yelp.com/biz/adelita-taqueria-and-restaurant-philadelphia?osq=Adelita+Taqueria%26+Restaurant

Table 3: Top similar business recommended in Pandas

	name	stars	cosine_similarity
7109	Los Taquitos de Puebla	4	1.000000
5990	San Antonio Mexican cousine	5	1.000000
3534	Los Cuatro Soles	5	0.942809
3292	El Limon - Norristown	4	0.942809
1525	La Hacienda Mexican Restaurant	4	0.942809

Table 4: Top similar business recommended in Pandas with sklearn cosine similarity

name	cosine similarity	stars
San Antonio Mexican cousine	0.9999999999999999	5
Los Taquitos de Puebla	0.9486832980505138	4
El Charro Negro	0.9486832980505138	4
El Limon - Norristown	0.9128709291752769	4
Plaza Garibaldi	0.8999999999999999	4

5.2 Business similarity suggestion with k-NN algorithm

I continued my search for similar restaurants also based on the k-nn algorithm inspired by a kaggle notebook [12]. In brief it's a a supervised machine learning approach called k-nearest neighbors (k-NN) and it is utilized for both classification and regression problems. It is a non-parametric algorithm that bases its predictions on how closely new input data and training examples resemble each other.

The "k" in the k-NN method denotes how many nearest neighbors are taken into account while producing predictions. The steps done are the following:

- Take features excluding name and stars from the `df_total` as x and stars as y
- Splitting the data into training set and test set
- To figure out which k to choose, I applied 5-fold cross-validation and the best value for k is 20.
- Then I fitted the model and printed the results

  `Score on training set: 0.5624903205823137 ; Score on test set: 0.52198142486068`

- Testing the model and use the last row as the validation test
- Getting the similar restaurants compared with the one taken in the validation test.

In the end, the results obtained are comparable with the other two approaches (cosine similarity from scratch and sklearn).

5.3 Business similarity suggestion with PySpark and cosine similarity

In this section I wanted to focus more on the scalability of the system. Therefore, I converted the approach used in section 5.1 to PySpark, performing all the preprocessing by obtaining `df_final_spark`. Specifically, I summarize the operations:

- Transform the categories into a list of categories for the business
- Perform filtering by state and business type.
- Extract the attributes as columns and then having some columns of nested attributes I re-run the operation
- Transform the values to 0-1 obtaining the `df_business_attr_spark`
- Same for categories creating the `df_categories_dm`
- Finally I get the `df_final_spark` by the join between the categories,attributes and the `restaurant_df_sel_spark` which contains the name and ratings of the various businesses.

Once I had completed all the preprocessing phase and obtained the various df's I could proceed with the recommendation. As seen in the code shown I created the `df_matrix` that contains *business_id,features*. The features were generated using `VectorAssembler`.It is a feature transformer that combines multiple input columns into a single vector column.

```
# Exclude the 'business_id' column from df_final
columns = [c for c in df_final_spark.columns if c != 'business_id']

# Convert binary values to integers
df_numeric = df_final_spark.select(['business_id'] + [col(c).cast('integer') for c in columns])

# Fill NaN values with 0
```

```
df_filled = df_numeric.fillna(0, subset=columns)

# Use VectorAssembler to create a feature vector column
assembler = VectorAssembler(inputCols=columns, outputCol='features')
df_matrix = assembler.transform(df_filled).select('business_id', 'features')
```

Listing 5.3: Suggestion system in PySpark

After that I proceeded to run the same test on the *Adelita Taqueria & Restaurant* using `cosine_similarity_spark` slightly modified to make it work properly with the PySpark pipeline. In this case I wanted to select 15 as the number of recommendations.This is the Table 5 obtained:

Table 5: Top similar business recommended in PySpark

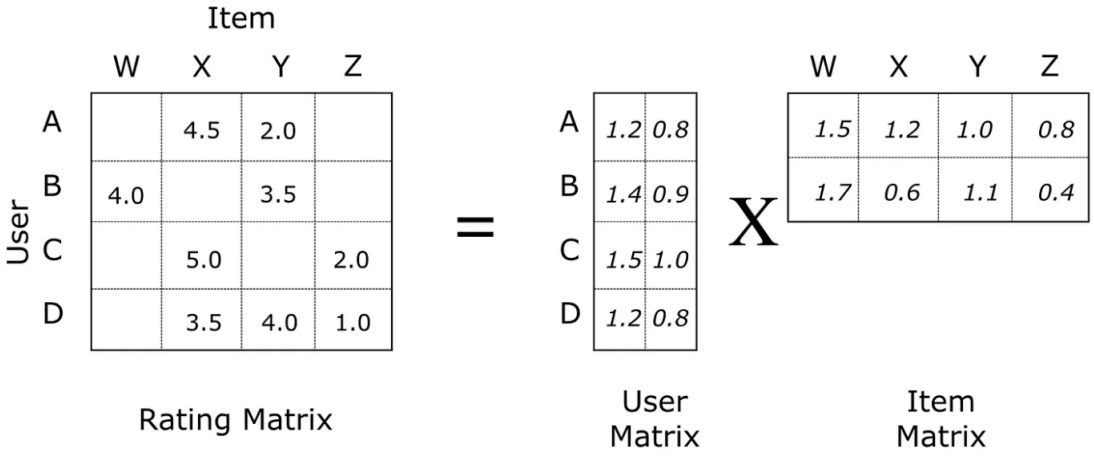
business_id	cosine_similarity	name	stars
F2QwLwzS3vF9os0...	[[1.0]]	San Antonio Mexican cousine	4.5
DwOhLOd9Say...	[[0.9803789354850793]]	Los Taquitos de Puebla	3.5
EoQiJ5D-pyWc...	[[0.9714285714285714]]	El Primo Taqueria	4.5
XQewVfTaosZ3U-4g...	[[0.9714285714285714]]	Los Cuatro Soles	4.5
mKJ_WV7TvrjyDjm...	[[0.9710083124552245]]	El Limon - Bensalem	4.5
SVf23pjKERkedqCdWl6ECA	[[0.9660917830792959]]	Teresa's Mesa	4.0
5ItgryJvadUrKVljjJ8l4g	[[0.9660917830792959]]	Que Chula Es Puebla	4.0
rNg75hKR0UIB5-jX5WiVQg	[[0.9660917830792959]]	Los Mariachis	3.5
1MVMKUvZfWwIqkhxP3rYvQ	[[0.9613406389911041]]	Indian Garden	4.0
sKUs4ISUgn3j6SeLYvSByg	[[0.9613406389911041]]	El Charro Negro	4.0
x1CtK2qnlCr_1DJFNo1_vw	[[0.9583148474999098]]	El Limon - Norristown	4.0
venWfi69QVylbyrvaPt0nQ	[[0.9578414886923188]]	Cafe Ynez	4.5
t0Qyogb4x–K9i5b0AoDCg	[[0.9578414886923188]]	Taco Maya	4.5
fcAaBcYFo1YqetIjs9l8Iw	[[0.9566222088265397]]	Sungate Diner	4.5
uh-387pbyipzjCMcj5Dy6w	[[0.9566222088265397]]	Las Palmas Del Sur	4.5

Also you can see that they are all restaurants that cook Mexican style. But these methods are based solely and exclusively on the profile of the restaurants without taking into account the feedback that the user has given to the various restaurants. For this reason, I decided to continue the project by introducing the user profile component as well.

5.4 Collaborative filtering with ALS ml library

In collaborative filtering, matrix factorization is the state-of-the-art solution for sparse data problem, although it has become widely known since Netflix Prize Challenge. A matrix factorization is a factorization of a matrix into a product of matrices. In the case of collaborative filtering, matrix factorization algorithms work by decomposing the user-item interaction matrix into the product of two lower dimensionality rectangular matrices.One matrix can be seen as the user matrix where rows represent users and columns are latent factors. The other matrix is the item matrix where rows are latent factors and columns represent items [13]. Here's a picture that describe the matrix factorization. The advantages

Figure 2: Example of Matrix Factorization



of using this approach are mainly two:

- Model learns to factorize rating matrix into user and business representations, which allows model to predict better personalized business ratings for users
- Less-known business can have rich latent representations as much as popular business have

More formally, in a sparse user-item interaction matrix the predicted rating user u will give item i is computed as:

$$r_{ui} = \sum_{f=0}^{nfactors} H_{u,f} W_{f,i}$$

Rating of item i given by user u can be expressed as a dot product of the user latent vector and the item latent vector. Latent factors are the features in the lower dimension latent space projected from user-item interaction matrix.

Increasing the number of latent factors will improve personalization, until the number of factors becomes too high, at which point the model starts to overfit. A common strategy to avoid overfitting is to add regularization terms to the objective function.

The objective of matrix factorization is to minimize the error between true rating and predicted rating.

5.4.1 Alternating Least Square (ALS) with Spark ML

Alternating Least Squares (ALS) is a parallel matrix factorization algorithm. It is implemented in Apache Spark ML and specifically designed for large-scale collaborative filtering problems. ALS effectively addresses scalability and sparsity issues commonly encountered with Ratings data. Furthermore, ALS exhibits simplicity and demonstrates excellent scalability even when handling massive datasets. The main points are:

- ALS uses L2 regularization
- ALS minimizes two loss functions alternatively; It first holds user matrix fixed and runs gradient descent with item matrix; then it holds item matrix fixed and runs gradient descent with user matrix
- Its scalability: ALS runs its gradient descent in parallel across multiple partitions

So let's get to the heart of the matter by taking a look at the code. Starting with `df_review_spark_indexed` I go to convert the profile ids to numeric format using the `StringIndexer`. Once this is done we go to split the data into training set and test set as reported by this code:

```
(training, test) = df_review_spark_indexed.randomSplit([0.8, 0.2])
```

Listing 5.4: Splitting dataset in training and test set for ALS

After that, the recommendation model based on the ALS model is created:

```
# Build the recommendation model using ALS on the training data
als = ALS(maxIter=5, regParam=0.01, userCol="user_id_index",
    itemCol="business_id_index", ratingCol="stars_review")
model = als.fit(training)
```

Listing 5.5: Building recommended system model based on the ALS model

There are some parameters that need to be considered such as:

- `maxIter`: is the maximum number of iterations to run
- `regParam`: specifies the regularization parameter in ALS
- `userCol`: as suggested from the name is the column of the user
- `itemCol`: as suggested from the name is the column of the item
- `ratingCol`: as suggested from the name is the column of the rating

This operation requires times according to the type of technical specifications you have. Those used in the project are specified in the section 6.1. Once we have fitted the model with the training data we can proceed to test our model with the test set.

```
predictions = model.transform(test)
#predictions.show()
```

Listing 5.6: Prediction over the test set with the ALS model

Then I proceeded to test the model by filtering for a generic user id which in this case is `user_id 14269.0` by descendingly sorting the prediction column.

To then understand the goodness-of-fit of the model, I calculated the NDCG (see the section 7.1) by manipulating the df columns as can be seen from the code:

```
from pyspark.sql.functions import expr

predictions = predictions.withColumn("rank", expr("row_number() over (partition by user_id_index order by prediction desc)"))
predictions = predictions.withColumn("dcg", expr("1 / (log2(rank + 1))"))
predictions = predictions.withColumn("idcg", expr("1 / (log2(1 + rank))"))
ndcg = (
    predictions
    .groupBy("user_id_index")
    .agg(expr("sum(dcg) as dcg_sum"), expr("sum(idcg) as idcg_sum"))
    .select(expr("avg(dcg_sum / idcg_sum)").alias("ndcg"))
    .collect()[0][0]
)
```

Listing 5.7: Calculating the ndcg over the predictions df obatined in ALS

Printing the metric we get `ndcg = 1.0`. A value of NDCG of 1 indicates that the system has produced perfect recommendations corresponding to user preferences.

5.5 Content based recommendation from scratch in PySpark

Finally we enter in the main section about the project. We introduce how I created a *Content-based* reccomendation system from scratch in PySpark. To do this, I started by constructing the `df_final_content_spark` which contains all the information related to the restaurants such as the categories and attributes coded in 0-1, its rating (*stars*) and its *business_id*. Through the latter it is possible to make a join with `df_review_spark_indexed` so that we also get the information about the user and his explicit rating for a given business through the review.

I decided in order to speed up the computation to reduce the data according to this operation:

```
merged_df = merged_df.sample(fraction=0.001, seed=42)
#merged_df.count()
```

Listing 5.8: Fractioning data

Nevertheless we have a good amount to proceed the with the recommendation system.

To make the system user-centric, I weighted the matrix obtained with the user's ratings in such a way that it is considered in the system. In fact here is the code:

```
# Select all columns and excluding these 'business_id_index', '
    stars_review', 'user_id_index'
columns_to_multiply = [column for column in merged_df.columns if
    column not in ['business_id_index', 'stars_review', '
    user_id_index','stars']]

# Perform the product over each column
for column in columns_to_multiply:
    weighted_merged_df = merged_df.withColumn(column, col(column)
    * col('stars_review'))
# this is the weighted cat matrix
#weighted_merged_df.show()
```

Listing 5.9: Weighted the business matrix with the user review rate

This is useful to me because by grouping by user_id and summing over the columns I get `user_profile`. However I need to do some preprocessing until I get the normalized user profile called in the code `user_profile_normalized_df`.

So the in the same matter we can obtain `business_profile_normalized_df`. Finally with this two profiles we can create the matrix with a crossJoin like in snippet:

```
# Union of business profiles and user profiles
```

```
cross_business_user = business_profile_normalized_df.crossJoin(
    user_profile_normalized_df)
```

Listing 5.10: Cross join between profiles

The structure of `cross_business_user` is shown in the Table 6 As you can see the

Table 6: Cross_business_user show example

business_id_index	user_id_index	user_features	business_features
128.0	14269.0	(289,[3,9,22,23,2..	(292,[0,1,5,11,24...

features have different dimensions. So before calculating the cosine similarity we need to fix this issue. To aim this here the snippet:

```
from pyspark.ml.linalg import Vectors, VectorUDT
from pyspark.sql.functions import udf
from pyspark.sql.types import IntegerType

# Cosine similarity function
def cosine_similarity(v1, v2):
    dot_product = float(v1.dot(v2))
    norm_v1 = float(v1.norm(2))
    norm_v2 = float(v2.norm(2))
    similarity = dot_product / (norm_v1 * norm_v2)
    return similarity

# UDF to truncate or expand feature vectors
def resize_features(features, max_dimension):
    return Vectors.sparse(max_dimension, features.indices,
    features.values)

# UDF to calculate cosine similarity as a column in a DataFrame
cosine_similarity_udf = udf(cosine_similarity)


# Find the maximum size between the two DataFrames
max_dimension = max(cross_business_user.withColumn("size", udf(
    lambda x: x.size, IntegerType())("business_features")).
    selectExpr("max(size)").collect()[0][0],
                    cross_business_user.withColumn("size", udf(
    lambda x: x.size, IntegerType())("user_features")).selectExpr("
    max(size)").collect()[0][0])

# Truncation or expansion of feature vectors
cross_business_user = cross_business_user.withColumn("
    business_features_resized", udf(lambda x: resize_features(x,
    max_dimension), VectorUDT())("business_features")) \
```

```
27                                          .withColumn("
    user_features_resized", udf(lambda x: resize_features(x,
    max_dimension), VectorUDT())("user_features"))
```

Listing 5.11: Cross join between profiles

As you can see we find the maximum dimension and we check if the column should be expanded or restricted based on the max. Now it's possible to apply `cosine_similarity` between the business features and the user features get `similarity_df`. Look at the schema in Listing 5.12:

```
root
 |-- business_id_index: double (nullable = false)
 |-- business_features: vector (nullable = true)
 |-- user_id_index: double (nullable = false)
 |-- user_features: vector (nullable = true)
 |-- business_features_resized: vector (nullable = true)
 |-- user_features_resized: vector (nullable = true)
 |-- similarity: string (nullable = true)
```

Listing 5.12: Similarity df schema

There is the column `similarity` that allows us to perform the recommendation. Lets select our user of interest, in this case, the same one that we have choose in the section 5.4.1. Then we order the similarity in desc way and filter by the user of interest.

Now is time to show the recommended business:

```
#Union with the DataFrame business_profiles_df to obtain
    information about recommended businesses
df_res_sel = df_res_sel.withColumnRenamed('business_id_index','
    df_res_sel_business_id_index')
recommended_businesses = user_similarity.join(df_res_sel,
    user_similarity.business_id_index == df_res_sel.
    df_res_sel_business_id_index, how='inner').drop('
    df_res_sel_business_id_index').select("business_id_index","name
    ","business_features","user_id_index","user_features","
    business_features_resized","user_features_resized","similarity"
    )

#Visualize the recommended business
#recommended_businesses.show(5)
```

Listing 5.13: Recommended businesses calculation

To understand if the recommendation are good enough we need to interpreter the data. Indeed after that, I created a `historical_data_stars` that is useful to our prediction. Indeed `recommended_businesses` is a join of dfs that give us a complete view of the prediction.

We build the prediction df in this way:

```
from pyspark.sql.functions import udf
from pyspark.sql.types import ArrayType, DoubleType

# Define a UDF (User Defined Function) to convert the struct column to an array of double
convert_array_udf = udf(lambda struct_col: struct_col["values"], ArrayType(DoubleType()))

# Apply the UDF to convert the column to the correct type in 'recommended_businesses'
recommended_businesses = recommended_businesses.withColumn("business_features", convert_array_udf("business_features"))

ground_truth = cross_business_user.withColumn("business_features_resized", convert_array_udf("business_features_resized"))


# Perform the join and select the desired columns
predictions = recommended_businesses.join(historical_data_stars, ["user_id_index", "business_id_index"], how="inner") \
                                    .na.fill(0) \
                                    .select("user_id_index", "business_id_index", "business_features", "stars_review")
```

Listing 5.14: Build prediction df

so by joining the historical data with the recommended_business.

I finally create a single joined df that contains all the necessary information on which I can perform the normalized discounted cumulative gain calculation. The result is `NDCG: 0.33141829507855797`.

Chapter 6

Scalability and complexity

So let us discuss the scalability and complexity aspects. Starting from the K-Nearest Neighbors algorithm (section 5.2) used in the code of the project in, the complexity will depend mainly on the size of the data set and the value of K. The computational complexity of the K-NN is O(n * m * d), where:

- n is the number of points in the data set (user or business profiles)
- m is the number of neighbors to be considered (value of K)
- d is the size of the feature vector (number of attributes or profile dimensions)

Thus, if you have a large dataset and a high value of K, the complexity can become significant. However, K-NN can be parallelized and scaled easily on distributed architectures such as Apache Spark, allowing large datasets to be handled.

If, on the other hand, we talk about ALS, its training complexity is dominated by the number of iterations and the number of latent (rank) factors specified. Specifically, the complexity for each iteration of the ALS algorithm is approximately O(m * n * rank), where m is the number of users and n is the number of recommendable items.

On the other hand, as for the content-based recommender system from scratch in PySpark we have to consider that the creation of the weighted matrix for example is O(n) where n is the number of features. If we consider the code provided for the calculation of the cross_bussiness_user matrix and all the above operations in the Listing 5.11 then we can say:

- Calculation of cosine similarity (cosine_similarity): the complexity will depend mainly on the size of the feature vectors passed as input (v1 and v2). Assuming they have size n then it will be O(n)
- UDF for resizing feature vectors (resize_features): assuming the number of nonzero elements is m then it will be O(m)

- Calculating the maximum size between the two DataFrames: assuming they have k rows the complexity will be O(k).
- Resizing feature vectors in DataFrames: . If we assume that the number of rows is k and the number of nonzero elements in the feature vectors is m, then the complexity will be O(k * m).

So in general we can approximate the computations of this important stage of the recommender system as an O(n + m + k + k * m). In general using an approach such as PySpark allows us to exploit some aspects related to distributed computation by dividing the work across several machines and solving problems in parallel. Also if we had an HDFS like Apache Hadoop available we could use partitions in such a way as to balance the load.

6.1 System specification

All the code was executed on *Google Colab*. Date of last execution: `06/06/2023`. The System specification are reported here:

- Disk information

```
!df -h

Filesystem      Size  Used Avail Use% Mounted on
overlay         108G   24G   85G  22% /
tmpfs            64M     0   64M   0% /dev
shm             5.8G     0  5.8G   0% /dev/shm
/dev/root       2.0G 1005M  952M  52% /usr/sbin/docker-init
tmpfs           6.4G  220K  6.4G   1% /var/colab
/dev/sda1        41G   25G   16G  61% /etc/hosts
tmpfs           6.4G     0  6.4G   0% /proc/acpi
tmpfs           6.4G     0  6.4G   0% /proc/scsi
tmpfs           6.4G     0  6.4G   0% /sys/firmware
```

Listing 6.1: Disk information

- CPU specs

```
!cat /proc/cpuinfo

processor : 0
vendor_id : GenuineIntel
cpu family  : 6
model   : 79
model name  : Intel(R) Xeon(R) CPU @ 2.20GHz
stepping  : 0
```

```
microcode : 0xffffffff
cpu MHz   : 2199.998
cache size  : 56320 KB
physical id : 0
siblings  : 2
core id   : 0
cpu cores : 1
apicid    : 0
initial apicid  : 0
fpu   : yes
fpu_exception : yes
cpuid level : 13
wp    : yes
flags   : fpu vme de pse tsc msr pae mce cx8 apic sep mtrr
   pge mca cmov pat pse36 clflush mmx fxsr sse sse2 ss ht
   syscall nx pdpe1gb rdtscp lm constant_tsc rep_good nopl
   xtopology nonstop_tsc cpuid tsc_known_freq pni pclmulqdq
   ssse3 fma cx16 pcid sse4_1 sse4_2 x2apic movbe popcnt aes
   xsave avx f16c rdrand hypervisor lahf_lm abm 3dnowprefetch
    invpcid_single ssbd ibrs ibpb stibp fsgsbase tsc_adjust
   bmi1 hle avx2 smep bmi2 erms invpcid rtm rdseed adx smap
   xsaveopt arat md_clear arch_capabilities
bugs    : cpu_meltdown spectre_v1 spectre_v2
   spec_store_bypass l1tf mds swapgs taa mmio_stale_data
   retbleed
bogomips  : 4399.99
clflush size  : 64
cache_alignment : 64
address sizes : 46 bits physical, 48 bits virtual
power management:

processor : 1
vendor_id : GenuineIntel
cpu family  : 6
model   : 79
model name  : Intel(R) Xeon(R) CPU @ 2.20GHz
stepping  : 0
microcode : 0xffffffff
cpu MHz   : 2199.998
cache size  : 56320 KB
physical id : 0
siblings  : 2
core id   : 0
cpu cores : 1
apicid    : 1
initial apicid  : 1
fpu   : yes
fpu_exception : yes
cpuid level : 13
```

```
wp     : yes
flags   : fpu vme de pse tsc msr pae mce cx8 apic sep mtrr
    pge mca cmov pat pse36 clflush mmx fxsr sse sse2 ss ht
    syscall nx pdpe1gb rdtscp lm constant_tsc rep_good nopl
    xtopology nonstop_tsc cpuid tsc_known_freq pni pclmulqdq
    ssse3 fma cx16 pcid sse4_1 sse4_2 x2apic movbe popcnt aes
    xsave avx f16c rdrand hypervisor lahf_lm abm 3dnowprefetch
     invpcid_single ssbd ibrs ibpb stibp fsgsbase tsc_adjust
    bmi1 hle avx2 smep bmi2 erms invpcid rtm rdseed adx smap
    xsaveopt arat md_clear arch_capabilities
bugs    : cpu_meltdown spectre_v1 spectre_v2
    spec_store_bypass l1tf mds swapgs taa mmio_stale_data
    retbleed
bogomips  : 4399.99
clflush size  : 64
cache_alignment : 64
address sizes : 46 bits physical, 48 bits virtual
power management:
```

Listing 6.2: CPU spec

- Memory info

```
!cat /proc/meminfo

MemTotal:       13294264 kB
MemFree:         9219976 kB
MemAvailable:   12020892 kB
Buffers:           62876 kB
Cached:          2930080 kB
SwapCached:            0 kB
Active:           854108 kB
Inactive:        2969768 kB
Active(anon):       1140 kB
Inactive(anon):   831364 kB
Active(file):     852968 kB
Inactive(file):  2138404 kB
Unevictable:           0 kB
Mlocked:               0 kB
SwapTotal:             0 kB
SwapFree:              0 kB
Dirty:               132 kB
Writeback:             0 kB
AnonPages:        829632 kB
Mapped:           280996 kB
Shmem:              1584 kB
KReclaimable:     100996 kB
Slab:             137044 kB
SReclaimable:     100996 kB
```

```
SUnreclaim:         36048 kB
KernelStack:         4492 kB
PageTables:         18972 kB
NFS_Unstable:           0 kB
Bounce:                 0 kB
WritebackTmp:           0 kB
CommitLimit:      6647132 kB
Committed_AS:     1987656 kB
VmallocTotal:   34359738367 kB
VmallocUsed:         9444 kB
VmallocChunk:           0 kB
Percpu:              1320 kB
HardwareCorrupted:      0 kB
AnonHugePages:      22528 kB
ShmemHugePages:         0 kB
ShmemPmdMapped:         0 kB
FileHugePages:          0 kB
FilePmdMapped:          0 kB
CmaTotal:               0 kB
CmaFree:                0 kB
HugePages_Total:        0
HugePages_Free:         0
HugePages_Rsvd:         0
HugePages_Surp:         0
Hugepagesize:        2048 kB
Hugetlb:                0 kB
DirectMap4k:        82744 kB
DirectMap2M:      4108288 kB
DirectMap1G:     11534336 kB
```

Listing 6.3: Memory information

Chapter 7

Results and conclusion

To evaluate the system I used the Normalized Discounted Cumulative Gain (NDCG).

7.1 Normalized Discounted Cumulative Gain (NDCG)

Before talking about the NDCG we need to understand what is CG (Cumulative Gain) and DCG (Discounted Cumulative Gain) [14]. Then there are two main assumptions:

- Highly relevant documents are more useful when appearing earlier in the search engine results list.
- Highly relevant documents are more useful than marginally relevant documents, which are more useful than non-relevant documents

CG

If every recommendation has a graded relevance score associated with it, CG is the sum of graded relevance values of all results in a search result list as reported in this formula:

$$CG_p = \sum_{i=1}^{p} rel_i$$

The Cumulative Gain at a particular rank position p, where the rel_i is the graded relevance of the result at position i.

The problem with CG is that it does not take into consideration the rank of the result set when determining the usefulness of a result set.

DCG

To overcome this we introduce DCG. DCG penalizes highly relevant documents that appear lower in the search by reducing the graded relevance value logarithimically proportional to the position of the result.

$$DCG_p = \sum_{i=1}^{p} \frac{2^{rel_i} - 1}{log_2(i+1)}$$

NDCG

An issue arises with DCG when we want to compare the search engines performance from one query to the next because search results list can vary in length depending on the query that has been provided.We perform this by sorting all the relevant documents in the corpus by their relative relevance producing the max possible DCG through position p (a.k.a Ideal Discounted Cumulative Gain). Here the formula:

$$nDCG_p = \frac{DCG}{IDCG_p}$$

where

$$IDCG_p = \sum_{i=1}^{|REL_p|} \frac{2^{rel_i} - 1}{log_2(i+1)}$$

REL_p represents the list of relevant documents (ordered by their relevance) in the corpus up to position p.

The ratios will always be in the range of [0, 1] with 1 being a perfect score — meaning that the DCG is the same as the IDCG. It was what happened in the section 5.4.1. With the recommender system seen in the section 5.5 the `NDCG: 0.33141829507855797`.This would indicate a low result suggesting recommended results that may not be very relevant. It could have been influenced by the reduction of the dataset or perhaps more user-related information could also be included through a textual analysis on the reviews.

Listings

List of Figures

List of Tables

Bibliography

[1] Giorgio Presti. Template latex. https://homes.di.unimi.it/presti/index.php?p=3&l=1, 2023.

[2] Anand Rajaraman, Jure Leskovec, and Jeffrey D. Ullman. *Mining Massive Datasets*. Cambridge University Press, 2014.

[3] Apache Spark. Pyspark overview. https://spark.apache.org/docs/latest/api/python/, 2023.

[4] Apache Spark. pyspark.sparkconf. https://spark.apache.org/docs/latest/api/python/reference/api/pyspark.SparkConf.html, 2023.

[5] Apache Spark. pyspark.sparksession. https://spark.apache.org/docs/3.2.0/api/java/org/apache/spark/sql/SparkSession.html, 2023.

[6] Apache Spark. pyspark.ml. https://spark.apache.org/docs/2.2.0/api/python/pyspark.ml.html, 2023.

[7] Apache Spark. pyspark.sql.function. https://spark.apache.org/docs/latest/api/python/reference/pyspark.sql/functions.html, 2023.

[8] Mahmoud Hassan Mahmoud. What is kaggle? https://www.kaggle.com/general/328265, 2022.

[9] Pandas. Pandas package overview. https://pandas.pydata.org/docs/getting_started/overview.html, 2023.

[10] codecademy. What is scikit-learn? https://www.codecademy.com/article/scikit-learn, 2023.

[11] codecademy. What is numpy? https://numpy.org/doc/stable/user/whatisnumpy.html, 2023.

[12] Aryan Sakhala. Kaggle notebook. https://www.kaggle.com/code/aryansakhala/recommend, 2021.

[13] towardsdatascience. Alternating least square (als) matrix factorization in collaborative filtering. https://towardsdatascience.com, 2018.

[14] towardsdatascience. Normalized discounted cumulative gain. https://towardsdatascience.com/normalized-discounted-cumulative-gain-37e6f75090e9, 2020.