Perform Collaborative-Filtering with AWS Technology and Terraform

Arriu Simone, Serra Sergio

12 July 2020

Contents

1 Introduction

The goal of this application is to perform a Collaborative-Filtering approach in order to obtain the best recommendation for a new user based on other users' ratings. The Collaborative-Filtering approach consists in making predictions about the interests of a user by collecting preferences (filtering) or taste information from many users (collaborative).

The idea is that if a new user Bob rates a movie with a certain mark and another user Alice marks that movie in a likely way, we'll have more probabilities that for a movie chosen randomly, they will have the same opinion.

Finally, the algorithm will return the best 25 recommendations for this new user.

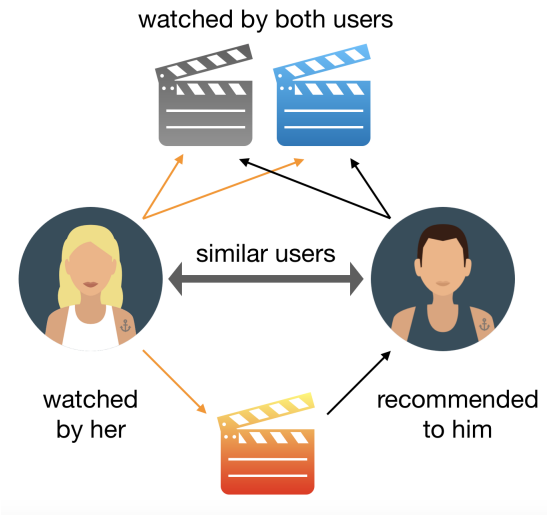

Figure 1: How the Collavorative-Filtering works

2 Used Technologies

- **AWS:** Amazon Web Services is a platform that provides cloud computing services in order to help users to create, protect and share the Big Data's applications. With AWS is possible to distribute the infrastructure nearly instantly, store, analyze and compute the data.
- **Terraform:** is a tool that allows to create, modify and improve the infrastructure development. The developers will write the infrastructure as a code and they will check the changes before applying them permanently. Terraform is also Automation Friendly: you can encode each element in order to automate the process.
- **Apache Spark:** is an engine that works with large data. It's open-source distributed cluster-computing framework. Spark's architecture is composed by a master and one or more slaves which execute the code in a distributed way.

3 Programming with Terraform

In Terraform there are different keywords that allow you to develop your infrastructure:

- **Provider "*name*":** is used in order to use the provider's resource. It's necessary specify the access credentials and the region of our provider account.
- **Resource "*type_resource*" "*local_name*":** when you want to create an infrastructure object, you have to specify the type of the resource and give it a local name that you will use inside the code.
- **Data "*data_source*" "*local_name*"**: data source allows to get external data in order to use it into Terraform.

3.1 Setup Environment

First of all we have defined the aws provider with the credentials of our account. Since the credential changes every three hours, you'll have to change them in the code in order to connect with your account.

```
provider "aws" {
    access_key = "your_access_key"
    secret_key = "your_secret_key"
    token = "your_token"
    region = "your_region"
}
```

Secondly, in order to connect to the master, we have generated the keys using the following command:

```
ssh-keygen -t rsa
ssh-keygen -f cluster-key -e -m pem
```

Where *cluster-key* is the name of our keys file. Then the resource for giving the keys from the file has been declared:

```
resource "aws_key_pair" "emr_key_pair" {
  key_name  = "emr-key"
  public_key = file("cluster-key.pub")
}
```

3.2 Amazon VPC

The first service that we have used is Amazon VPC: it is a service which allows the developers to create a virtual network that takes advantage of the AWS infrastructure. The developer has the whole control on this environment: subnet creation, configuration of routing table and gateway, and other services.

```
resource "aws_default_vpc" "default" {
  enable_dns_hostnames = true
  enable_dns_support = true

  tags = {
    Name = "Default VPC"
  }
}

resource "aws_default_subnet" "default" {
  availability_zone = "us-east-1b"

  tags = {
    Name = "Default subnet"
  }
}
```

In this case, the *aws_deafult_vpc* resource will get the default VPC already defined in AWS. In a similar way we have defined the default subnet. In this case it has been important declare the *availability_zone* on the base of our region (us-east-1) specifying the subnet (us-east-1**b**).

3.3 Amazon EMR

The second service that we have chosen is EMR Cluster because it's optimized for working with Apache Spark. In fact, it can reach three times the speed of the standard version of Spark. It's also linked with services like EC2, S3, AWS Glue, and other services we'll talk about during the documentation.

```
resource "aws_emr_cluster" "emr-spark-cluster" {
   name = "EMR-cluster-example"
   release_label = "emr-5.13.0"
   applications = [ "Spark", "Hadoop"]

   ec2_attributes {
     instance_profile = aws_iam_instance_profile.emr_profile.arn
     key_name = aws_key_pair.emr_key_pair.key_name
     subnet_id = aws_default_subnet.default.id
     emr_managed_master_security_group = aws_security_group.master_security_group.id
     emr_managed_slave_security_group = aws_security_group.slave_security_group.id
   }
   core_instance_group {
     instance_type = "m4.large"
     instance_count = 2
   }
   master_instance_group {
    instance_type = "m4.large"
   }

   configurations_json = <<EOF
   [
     {
       "Classification": "spark",
       "Properties": {
         "maximizeResourceAllocation": "true"
       }
     }
   ]
 EOF
```

In this case, we have defined our EMR cluster specifying the name of the resource, the local name, the version and the applications that it will use (Spark and Hadoop). Inside, we have declared the EC2 instances, in particular one *m4.large* master and two *m4.large* slaves. In the case where you want to specify some Spark configurations, you need to insert them in the EOF list in *configurations_json*. In this case, the *maximizeResourceAllocation* property allows to use all the slaves specified above in the *instance_count* statement.

EC2 services offers computational capacity through virtual machines. His aim is satisfy the need for computing power for who can't afford large investments.

To make the EMR Cluster and the EC2 instances work, we have needed to define the IAM role. With IAM service, the developers can distribute their credentials to every instances and in this way access to the AWS' resources securely.

First of all, the IAM roles have been defined. Inside the *EOF Statement*, the *EMR* and *Athena* services have been specified. Instead, with *"Effect": "Allow"* this role has been allowed to use these services.

```
resource "aws_iam_role" "spark_cluster_iam_emr_service_role" {
    name = "spark_cluster_emr_service_role"

    assume_role_policy = <<EOF
{
    "Version": "2008-10-17",
    "Statement": [
        {
            "Sid": "",
            "Effect": "Allow",
            "Principal": {
                "Service": "elasticmapreduce.amazonaws.com"
            },
            "Action": "sts:AssumeRole"
        },
        {
          "Sid": "Stmt1547414166585",
          "Effect": "Allow",
          "Principal":{
                "Service": "athena.amazonaws.com"
          },
          "Action": "sts:AssumeRole"
        }
    ]
}
EOF
}
```

In addition, we have specified the policies which have been attached to the IAM roles.

```
resource "aws_iam_role_policy_attachment" "emr-service-policy-attach" {
   role = aws_iam_role.spark_cluster_iam_emr_service_role.id
   for_each = toset([
    "arn:aws:iam::aws:policy/service-role/AmazonElasticMapReduceRole",
    "arn:aws:iam::aws:policy/AmazonAthenaFullAccess"
   ])
   policy_arn= each.value
}
```

For instance, we have defined the role *spark_cluster_iam_emr_service_role* and for that we have attached the EMR and Athena policies. Hence, cluster will have the permission to access to those services. In order to allow the communications between the services, we have defined the security group for the master and the slaves.

To do that, we have specified the *vpc_id* and all the rules that manage the ingress and egress traffic. The following traffic rules have been expected for the master node:

```
resource "aws_security_group" "master_security_group" {
  name       = "master_security_group"
  description = "Allow inbound traffic from VPN"
  vpc_id     = aws_default_vpc.default.id
  revoke_rules_on_delete = true

  ingress {
    from_port  = "0"
    to_port    = "0"
    protocol   = "-1"
    self       = true
  }

  ingress {
     from_port  = "8443"
     to_port    = "8443"
     protocol   = "TCP"
  }

  egress {
    from_port  = "0"
    to_port    = "0"
    protocol   = "-1"
    cidr_blocks = ["0.0.0.0/0"]
  }

  ingress {
    from_port  = 22
    to_port    = 22
    protocol   = "TCP"
    cidr_blocks = ["${chomp(data.http.myip.body)}/32"]
  }

  ingress {
    from_port  = 8088
    to_port    = 8088
    protocol   = "TCP"
    cidr_blocks = ["${chomp(data.http.myip.body)}/32"]
  }

  ingress {
     from_port  = 18080
     to_port    = 18080
     protocol   = "TCP"
     cidr_blocks = ["${chomp(data.http.myip.body)}/32"]
    }

  ingress {
     from_port  = 4040
     to_port    = 4040
     protocol   = "TCP"
     cidr_blocks = ["${chomp(data.http.myip.body)}/32"]
  }
  lifecycle {
    ignore_changes = [ingress,egress]
  }
  tags = {
    name = "emr_test"
  }
}
```

The above rules refer to the following services:

- Port 0: allows communication between nodes in the VPC.
- Port 8443: allows communication between cluster manager and master/slave nodes.
- Port 22: allows SSH connection with the master node
- Port 8088: allows connection to the Yarn UI
- Port 18080: allows connection to the Spark History
- Port 4040: allows connection to the Spark UI

In order not to change every time your ip address, the data block has been used. Thanks to it you can store your ip in the *myip* variable used inside *cidr_blocks*. The same has done for the slaves, except for the fact that they don't need to interface with the Spark and Yarn UI.

```
data "http" "myip" {
  url = "http://ipv4.icanhazip.com"
}
```

3.4 Amazon S3

In order to store the data, a third service has been chosen: Amazon S3 which is highly scalable, secured and with low latency. From the stored data, it's easy to make a task for recovering data. First of all the aws bucket has been defined specifying a unique name (bucket variable) and the region of our account. Since the name is unique, you have to change it.

```
resource "aws_s3_bucket" "your_bucket_name" {
  bucket = "your_bucket_name"
  region = "your_region"

  versioning {
    enabled = true
  }
}
```

Then the data was taken from the local machine specifying the bucket where the data will be stored. Inside *files* folder there are the Python and CSV files. In order to be able to use the command *fileset*, it's necessary to install the last version of Terraform, otherwise there'll be several issues. An important thing is to set the local path where the files that you want to upload are.

```
resource "aws_s3_bucket_object" "your_bucket_name" {
  for_each = fileset(path.module, "files/*")
  bucket = aws_s3_bucket.your_bucket_name.bucket
  key = each.value
  source = "${path.module}/${each.value}"
}
```

3.5 Amazon Athena & Amazon Glue

The last service used is Amazon Athena with AWS Glue. Athena is a query service which simplify the data analysis in Amazon S3 bucket with SQL expression. Furthermore, it is linked to AWS Glue which is a service for data management. In this project it has been used for creating the table where the queries were run. An example of creating table is the following:

```
resource "aws_glue_catalog_table" "table_movies" {
  name          = "movies_table"
  database_name = aws_athena_database.hoge.name
  table_type    = "EXTERNAL_TABLE"
    owner = "owner"
    storage_descriptor {
        location      = "s3://${aws_s3_bucket.your_bucket.bucket}/files/movie"
        input_format  = "org.apache.hadoop.mapred.TextInputFormat"
        output_format  = "org.apache.hadoop.mapred.TextInputFormat"
        compressed = "false"
        number_of_buckets = -1
        ser_de_info {
            name   = "movies"
            serialization_library = "org.apache.hadoop.hive.serde2.lazy.LazySimpleSerDe"
            parameters = {
                "field.delim" = ","
                "skip.header.line.count" = 1 # Skip file headers
            }
        }
        columns {
            name   = "movie_id"
            type   = "int"
        }
        columns {
            name   = "title"
            type   = "string"
        }
        columns {
            name   = "genres"
            type   = "string"
        }
    }
}
```

In this code the name and the type of the table have been set, the name of the database, the S3 location where the service has to take the data and the columns of the tables. In the same way, the ratings table has been defined. Another important parameter is *skip.header.line.count* thanks to which the header row has been skipped.

```
resource "aws_athena_database" "hoge" {
  name   = "users"
  bucket = aws_s3_bucket.your_bucket.bucket
}
```

The Athena database has been defined with the name and the bucket in order to link itself to S3. After defining the resources for Athena and Glue, the query has been created. Seven queries have been expected:

Query 1: Select all the movies with *Mystery* genre

```
resource "aws_athena_named_query" "one" {
  name     = "query_1"
  database = aws_athena_database.hoge.name
  query    = "SELECT title FROM ${aws_glue_catalog_table.table_movies.name} WHERE genres LIKE
      '%Mystery%'"
}
```

Query 2: Select all the movies of 2018

```
resource "aws_athena_named_query" "two" {
  name     = "query_2"
  database = aws_athena_database.hoge.name
  query    = "SELECT title FROM ${aws_glue_catalog_table.table_movies.name} WHERE title LIKE
      '%(2018)%'"
}
```

Query 3: Select all the movies of 2016 or 2018

```
resource "aws_athena_named_query" "three" {
  name     = "query_3"
  database = aws_athena_database.hoge.name
  query    = "SELECT title FROM ${aws_glue_catalog_table.table_movies.name} WHERE title LIKE
      '%(2016)%' OR title LIKE '%(2018)%"
}
```

Query 4: Select all the movies with the best score (rating = 5.0)

```
resource "aws_athena_named_query" "four" {
  name     = "query_4"
  database = aws_athena_database.hoge.name
  query    = "SELECT DISTINCT title FROM ${aws_glue_catalog_table.table_movies.name} AS M JOIN
      ${aws_glue_catalog_table.table_ratings.name} AS R ON M.movie_id=R.movie_id WHERE rating=5.0"
}
```

Query 5: Select all the movies with *Fantasy* and *Adventure* genres and with the best score

```
resource "aws_athena_named_query" "five" {
  name     = "query_5"
  database = aws_athena_database.hoge.name
  query    = "SELECT title FROM ${aws_glue_catalog_table.table_movies.name} AS M JOIN
      ${aws_glue_catalog_table.table_ratings.name} AS R ON M.movie_id=R.movie_id WHERE genres LIKE
      '%Fantasy%' AND genres LIKE '%Adventure%' AND rating=5.0"
}
```

Query 6: Select all the movies with rating 1 or 2 of 1995

```
resource "aws_athena_named_query" "six" {
  name     = "query_6"
  database = aws_athena_database.hoge.name
  query    = "SELECT title FROM ${aws_glue_catalog_table.table_movies.name} AS M JOIN
      ${aws_glue_catalog_table.table_ratings.name} AS R ON M.movie_id=R.movie_id WHERE rating=1 OR
      rating=2 AND title LIKE '%(1995)%'"
}
```

Query 7: Select all the movies and ratings ordered in a descending way

```
resource "aws_athena_named_query" "seven" {
  name     = "query_7"
  database = aws_athena_database.hoge.name
  query    = "SELECT DISTINCT title, rating FROM ${aws_glue_catalog_table.table_movies.name} AS M
      JOIN ${aws_glue_catalog_table.table_ratings.name} AS R ON M.movie_id=R.movie_id ORDER BY rating
      DESC"
}
```

More specifically, AWS Glue is a managed extraction, transfer and loading (ETL) service that allows you to automate long data preparation procedures for analysis. AWS Glue automatically detects and profiles data through the appropriate catalog, suggests and generates ETL code to transform the source data into target schemes and performs ETL processes in a flexible and horizontally scalable Apache Spark environment to load information in the target path. In addition, this service allows you to configure, orchestrate and monitor complex data flows.

To work, Glue needs a *crawler*. It scans the different datastores available to automatically collect the schemes and structure of the partitions, filling in the Glue data catalog with the corresponding table definitions and statistics.

As before, the IAM roles have been defined, but in this case for configuring the Glue crawler in a proper way.

```
resource "aws_iam_role" "glue_crawler_role" {
  name = "analytics_glue_crawler_role"

  assume_role_policy = <<EOF
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Action": "sts:AssumeRole",
      "Principal": {
        "Service": "glue.amazonaws.com"
      },
      "Effect": "Allow",
      "Sid": ""
    }
  ]
}
EOF
}
```

Also in this case, the service *glue.amazonaws.com* and the *"Effect": "Allow"* have been associated to the *glue_crawler_role*.

In order to link the crawler to the database and to the S3 bucket, the following code has been defined:

```
resource "aws_glue_crawler" "product_crawler" {
  database_name = aws_athena_database.hoge.name
  name = "analytics-product-crawler"
  role = aws_iam_role.glue_crawler_role.arn

  schedule = "cron(0 0 * * ? *)"

  configuration = "{\"Version\": 1.0, \"CrawlerOutput\": { \"Partitions\": { \"AddOrUpdateBehavior\":
      \"InheritFromTable\" }, \"Tables\": {\"AddOrUpdateBehavior\": \"MergeNewColumns\" } } }"

  schema_change_policy {
    delete_behavior = "DELETE_FROM_DATABASE"
  }

  s3_target {
    path = "s3://${aws_s3_bucket.your_bucket.bucket}/files"
  }
}
```

Once the role has been defined, in order to give to this role the full access for the data management, the following policies have been associated with.

```
resource "aws_iam_role_policy" "glue_crawler_role_policy" {
  name = "analytics_glue_crawler_role_policy"
  role = aws_iam_role.glue_crawler_role.id

  policy = <<EOF
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Effect": "Allow",
      "Action": [
        "glue:*"
      ],
      "Resource": [
        "*"
      ]
    },
    {
      "Effect": "Allow",
      "Action": [
        "s3:GetBucketLocation",
        "s3:ListBucket",
        "s3:GetBucketAcl",
        "s3:GetObject",
        "s3:PutObject",
        "s3:DeleteObject"
      ],
      "Resource": [
        "arn:aws:s3:::analytics-product-data",
        "arn:aws:s3:::analytics-product-data/*"
      ]
    },
    {
      "Effect": "Allow",
      "Action": [
        "logs:CreateLogGroup",
        "logs:CreateLogStream",
        "logs:PutLogEvents"
      ],
      "Resource": [
        "arn:aws:logs:*:*:/aws-glue/*"
      ]
    }
  ]
}
EOF
}
```

3.6 Our goal and run steps

Our goal is to make the project execution as automatic as possible, so that the user has to enter as little information as possible. To do that, inside the cluster, several steps have been defined, where each one performs a specific task, including those that the user should have performed from the AWS CLI.

Those steps are the following:

1. Create a directory *datasetS3* inside *hadoop*, the master node. This folder will contains all the python files.

```
step {
     action_on_failure = "CONTINUE"
     name              = "Create directory."

     hadoop_jar_step {
       jar  = "command-runner.jar"
       args = ["mkdir", "/home/hadoop/datasetS3/"]
     }
   }
```

2. Copy the file python *main.py* from the S3 bucket to the master folder *datasetS3*. The same has been done for the other python file (*methods.py* and *baseline.py*).

```
step {
     action_on_failure = "CONTINUE"
     name              = "Copy main.py script from s3"

     hadoop_jar_step {
       jar  = "command-runner.jar"
       args = ["aws", "s3", "cp", "s3://${aws_s3_bucket.your_bucket.bucket}/files/main.py",
           "/home/hadoop/datasetS3/" ]
     }
   }
```

3. Run the code. *main.py* is the main python file, which recalls *methods.py*. The master node will execute *main.py* from his folder *datasetS3*. In order to run the Spark application, the command used was *spark-submit*, specifying the following parameters:

   - *deploy-mode*: *"client"* if the number of executors does not include the master node as a worker. In this case, the master node is in charge of dividing the work among the slaves.
   - *num-executors*: the number of slaves.
   - *py-files*: since our application is computed thanks to two python files, it has been necessary to zip those files. After this command, the zip's path on S3 was specified.
   - *Number of partitions*: after the path of the main python file it's necessary to specify the number of partitions into which the RDDs have been splitted.

```
step {
    action_on_failure = "CONTINUE"
    name              = "Run code"

    hadoop_jar_step {
      jar  = "command-runner.jar"
      args = ["spark-submit", "--deploy-mode", "client", "--master", "yarn"
          ,"--num-executors", "2", "--conf",
          "spark.dynamicAllocation.enabled=false","--py-files",
          "s3://${aws_s3_bucket.your_bucket.bucket}/files/zippo.zip",
          "/home/hadoop/datasetS3/main.py", "8"]
    }
}
```

4. Copy the *movies.csv* to a separate folder in order to compute the query in the next step with *Amazon Athena*. The same has been done for the other csv file called *ratings.csv*.

```
step {
     action_on_failure = "CONTINUE"
     name              = "Copy csv file from s3 to movie folder."

     hadoop_jar_step {
       jar  = "command-runner.jar"
       args = ["aws", "s3", "cp", "s3://${aws_s3_bucket.your_bucket.bucket}/files/movies.csv",
           "s3://${aws_s3_bucket.your_bucket.bucket}/files/movie/"]
     }
   }
```

5. Run the query on the specified database. It's important to specify the query that you want to execute, the name of the database (take a look to the *Database* parameter inside *args*) and the S3 output location where the results will appear.

```
step {
     action_on_failure = "CONTINUE"
     name              = "Execute query"

     hadoop_jar_step {
       jar  = "command-runner.jar"
       args = ["aws", "athena", "start-query-execution", "--query-string",
           "${aws_athena_named_query.foo.query}", "--query-execution-context",
           "Database=${aws_athena_database.hoge.name}", "--result-configuration",
           "OutputLocation=s3://${aws_s3_bucket.your_bucket.bucket}/files/movie/" ]
     }
   }
```

4 Programming with Pyhon

In order to develop the Collaborative-Filtering approach, two python files have been written:

- *main.py*
- *methods.py*

4.1 Main.py

It's the main program which recall several methods from *methods.py*. Among the structures to be used to manage the data, the *RDD* (Resilient Distributed Dataset) has been chosen. Since the RDDs born and live in memory, the execution of the iterative jobs or the jobs with the goal to transform the dataset several times are immensely faster than a map-reduce sequence.

This file will print the best 25 movies recommendation for a new user. So as to do that, a Machine Learning approach has been used which takes the common rated movies and, on the base of the ratings, recommends the best unrated movies.

When you work with Spark and a programming language, it's necessary the Spark Configuration (*SparkConf()*); with this statement the Spark application can be executed by the cluster. Furthermore, it's necessary to define a SparkContext. It allows to run the application by the executors. The *sc* variable will contain the SparkContext.

In addition, you can define the *setLogLevel* in order to see only the *ERROR* messages and if you have to perform some query you can define a new context from the previous one: *SQLContext*.

```
conf = SparkConf().setAppName("MoviesReccomandations")
sc = SparkContext(conf=conf)
sc.setLogLevel("ERROR")
sqlContext = SQLContext(sc)
```

4.2 Methods.py

This file contains all the methods recalled by *main.py*. In particular, it's important to linger on two methods:

- *ObtainRDD*:

```
def ObtainRDD_Ratings(sc,n_partitions):

        ratings_data_temp = sc.textFile("s3://bucketbigdataemr/files/ratings.csv",n_partitions)

        ratings_data_header = ratings_data_temp.take(1)[0]

        ratings_data = ratings_data_temp.filter(lambda line: line!=ratings_data_header)\
            .map(lambda line: line.split(",")).map(lambda tokens:
                (tokens[0],tokens[1],tokens[2])).cache()
```

  In order to obtain the RDD from the ratings CSV, two parameters have been specified:

  - The S3 path containing the CSV file.
  - The number of partitions to split the RDD into, that is equal to the number of executors × number of cores inside the cluster.

  Afterwards, the header row has been taken and at the end the RDD has been filtered excluding the header row and taking only the first three columns (*timestamp* column was not necessary). The *cache()* method allows to make the RDD available in memory without the need to read it over again.

  The same procedure has been done for the movie CSV.

- *DefNewUser*:

  Given that the Collaborative-Filtering is an algorithm that helps the new users in order to have new recommendations on the base of his interests, the defining of a new user is the main task of this approach.

```
def DefNewUser(sc):

    new_user_ID = 0

    new_user_ratings = [
      (0,260,4),
      (0,1,3),
      (0,16,3),
      (0,25,4),
      (0,32,4),
      (0,335,1),
      (0,379,1),
      (0,296,3),
      (0,858,5),
      (0,50,4)
    ]

    new_user_ratings_RDD = sc.parallelize(new_user_ratings)

    return new_user_ratings_RDD, new_user_ID, new_user_ratings
```

  First of all, to the *new_user_ID* variable has been associated an ID equal to 0 because it's the only one that is not already used in the dataset. Subsequently, a list with the (*user_id*, *movie_id*, *rating*) has been populated. The movies inside the list are a small set of movies that the new user has rated.

  In order to create a distributed dataset that can be operated on in parallel the *sc.parallelize()* instruction has been used.

4.3 How the code works

The code starts with the Spark's configuration. The following code has been explained in the previous Main.py section.

```
conf = SparkConf().setAppName("MoviesReccomandations")
sc = SparkContext(conf=conf)
sc.setLogLevel("ERROR")
sqlContext = SQLContext(sc)
```

Then, the *ratings RDD* and the *movies RDD* have been obtained. The variable *n_partitions* takes the value from the Terraform step and splits the RDDs in partitions on the bases of the value of this variable. The method *ObtainRDD* has been explained in the Methods.py section.

```
n_partitions = int(sys.argv[1])

ratings_data = ObtainRDD_Ratings(sc,n_partitions)

movies_data, movie_titles = ObtainRDD_Movies(sc,n_partitions)
```

As illustrated before, *DefNewUser()* allows you to insert a new user, specifying the movies and the ratings associated with. Take a look to the previous Methods.py section in order to see how this method works.

Afterwards, the dataset *ratings_data* has been splitted into *training* and *test* set, respectively for the 80% and 20%.

```
user_ratings, user_ID, list_user_ratings = DefNewUser(sc)

training_final, test_final = ratings_data.randomSplit([8,2], seed=42)
```

The following phase is very important for the model evaluation. In order to evaluate the model in a different way then the common metrics, the RDD of the movies rated by this new user has been splitted into two new sets in an equal way (50% and 50%). This step will be explained more specifically in the Model evaluation section.

```
training_userRatings,test_userRatings = user_ratings.randomSplit([5,5], seed=42)
```

Once this is done the final *training* and *test* set were obtained by the union of the training and test set of the ratings and the user.

```
final_test = test_final.union(test_userRatings)
final_training = training_final.union(training_userRatings)
```

Subsequently, the ML model were trained by the *ALS* method, giving as parameters the *final_training* set, the *best_rank* for our dataset, the *randomization factor* seed = 5 and the default values for the *iterations* and *lambda*, which is the regularization parameter. ALS is the most useful method for training in the recommendation system.

```
new_ratings_model = ALS.train(final_training, 20, seed=5, iterations=10, lambda_=0.1)
```

GetRealAndPreds is a method whose aim is to make predictions on the given *test* set by using the final model just obtained. Once the predictions have been performed, those were joined with the test set in order to get the real and predicted values for every rated movie.

Now, it's possible to compute the *RMSE* and *MSE* metrics, in order to see the trained model error.

```
GetRealAndPreds(new_ratings_model, final_test, 1)

def GetRealAndPreds(final_model, final_test, mode):
    test_for_predict_RDD = final_test.map(lambda x: (x[0], x[1]))
    predictions = final_model.predictAll(test_for_predict_RDD).map(lambda r: ((r[0], r[1]),
        round(r[2]*2)/2))

    rates_and_preds = final_test.map(lambda r: ((int(r[0]), int(r[1])),
        float(r[2]))).join(predictions)

    error = math.sqrt(rates_and_preds.map(lambda r: (r[1][0] - r[1][1]) ** 2).mean())

    print('For testing data the RMSE is %s' % (error))
    mse = rates_and_preds.map(lambda r: (r[1][0] - r[1][1]) ** 2).reduce(lambda x, y: x + y) /
        rates_and_preds.count()
    print('For testing data the MSE is %s' % (mse))
    if mode == 2:
       print(rates_and_preds.take(rates_and_preds.count()))
```

After that, from the *list_user_ratings* containing the movie rated by the new user, was taken only the *movie_id*, with a *map* method.

```
newUser_movies_rated = map(lambda x: x[1], list_user_ratings)
```

FilterMovie method filters all the movies not yet seen by the new user. Taking the *newUser_movies_rated* RDD, the *movies_data* set will be filtered.

```
newUser_unrated_movies_RDD = sc.emptyRDD()
newUser_unrated_movies_RDD = FilterMovie(newUser_unrated_movies_RDD, newUser_movies_rated,
    movies_data)
```

In order to obtain tuples of the type (*id_new_user*,*movie_id*), the previous RDD has been mapped. Then, the model were used for getting the predictions on the unrated movies for the new user.

```
newUser_unrated_movies_RDD = newUser_unrated_movies_RDD.map(lambda x: (user_ID, x[0]))
newUser_moviePrediction_RDD = new_ratings_model.predictAll(newUser_unrated_movies_RDD)
```

Subsequently, several *map* operations were carried out so as to obtain the tuples in a compatible format for the join operation. This join matches the predictions for the unrated movies to their title and the number of the total ratings for every movie (*IDRatings*).

```
newUser_moviePrediction_rating_RDD = newUser_moviePrediction_RDD.map(lambda x: (x.product, x.rating))

IDRatings = ratings_data.map(lambda row: (int(row[1]), 1)).reduceByKey(add)

newUser_join_recommendations_RDD = \
    newUser_moviePrediction_rating_RDD.join(movie_titles).join(IDRatings)

new_final_recommendations_RDD = \
    newUser_join_recommendations_RDD.map(lambda r: (r[1][0][1], r[1][0][0], r[1][1]))
```

The goal of the Collaborative Filtering approach is to get new recommendations for a new user. Now, we can obtain the best 25 recommendations ordered by their predicted value in a descending way. (This is done by setting *-x[1]*)

```
top_movies = new_final_recommendations_RDD.filter(lambda r: r[2]>=25).takeOrdered(25, key=lambda x:
    -x[1])

print('Top Movie Recommandation:\n%s' % '\n'.join(map(str, top_movies)))
```

The Collaborative-Filtering final output is the following:

```
Top Movie Recommandation:
(u'Cranford (2007)', 5.249198153338587, 35)
(u"Won't You Be My Neighbor? (2018)", 5.224156593889124, 83)
(u'Johnny Cash at Folsom Prison (2008)', 5.144937187566037, 25)
(u'Home Room (2002)', 5.136343954923534, 57)
(u'Slaying the Badger', 5.039285786204488, 25)
(u'Endeavour (2012)', 5.024407846035874, 32)
(u'"I', 4.952139367406934, 85)
(u'The Mask You Live In (2015)', 4.9272156540640015, 25)
(u'"Amelie (Fabuleux destin d\'Am\xe9lie Poulain', 4.920952080967787, 37167)
(u"Winter on Fire: Ukraine's Fight for Freedom (2015)", 4.917111617485844, 85)
(u'The Frame (2014)', 4.914469551140936, 35)
(u'Adventures Of Sherlock Holmes And Dr. Watson: The Twentieth Century Approaches (1986)',
    4.899779733673344, 105)
(u'The Keepers (2017)', 4.879710503371473, 47)
(u'56 Up (2012)', 4.871985031864572, 193)
(u'Cloudburst (2011)', 4.866902635084169, 33)
(u'The Barkley Marathons: The Race That Eats Its Young (2015)', 4.837383965006364, 64)
(u'Seven Up! (1964)', 4.836731462072093, 506)
(u'Magic & Bird: A Courtship of Rivals (2010)', 4.834353468091068, 27)
(u'The Hunting Ground (2015)', 4.831144926092497, 51)
(u'The Missing (2014)', 4.827181552241241, 45)
(u'Baseball (1994)', 4.826185531118856, 42)
(u'BlacKkKlansman (2018)', 4.8235369514467, 147)
(u'Olive Kitteridge (2014)', 4.821066879048896, 211)
(u'Oklahoma City (2017)', 4.818981665285516, 31)
(u'28 Up (1985)', 4.816066609154298, 857)

RMSE is 0.827018124277
MSE is 0.683958977882
```

4.3.1 Model evaluation

Apart from *RMSE* and *MSE* metrics the model has been evaluated in another way. Take a look at the following code:

```
training_final, test_final = ratings_data.randomSplit([8,2], seed=42)
training_userRatings,test_userRatings = user_ratings.randomSplit([5,5], seed=42)

final_test = test_final.union(test_userRatings)
final_training = training_final.union(training_userRatings)
```

As illustrated above, the final training and test set are not only given by the split of the *ratings_data*, but also by the split of the *user_ratings* RDD containing all the movies rated by this new user. More specifically, in order to see if the predictions of the already rated movies will be similar to the real ratings, 50% of the *user_rating* has been used as training and the rest 50% as test (*test_userRatings*).

Using the same model trained before *GetRealAndPreds*, in this case, will return the predictions only for the movies contained in *test_userRatings*. The final output will be tuples with the format: ((*user_id*, *movie_id*),(*real_rate*,*predicted_rate*)). Now it's possible to understand if the model is predicting in a well way for the new user.

```
GetRealAndPreds(new_ratings_model, test_userRatings, 2)
```

The final output is:

```
RMSE is 0.57735026919
MSE is 0.333333333333
[((0, 1449), (3.5, 4.5)), ((0, 2478), (2.5, 3.0)), ((0, 307), (4.0, 4.0)), ((0, 1590), (3.0, 2.5)),
    ((0, 2134), (3.0, 3.5)), ((0, 481), (3.0, 3.5))]
```

This RMSE score is the best that we have obtained. As you can see, the difference between the real and the predicted rating is really low. The model is able to predict with an acceptable error values. In this way, we can state that the model is good.

The RMSE score fluctuates between 0.57 and 0.87 on the base on how the dataset has been splitted. Inside the *DefNewUser()* method there are several movies commented that you can use in order to check how the RMSE changes.

4.3.2 Comparison to other works

In order to see if our errors were good, we have done a research. We have found several papers that compute Collaborative-Filtering with an higher or similar error value than ours considering the model trained in order to obtain the best 25 recommendation.

The two paper mentioned are:

- *Movie Recommender System Based on Collaborative Filtering Using Apache Spark* [**1**]: at the page 292, in table 3 there are several experiments where the RMSE is always higher than ours.
- *Comparative Analysis Of Movie Recommendation System Using Collaborative Filtering In Spark Engine* [**2**]: at the page 13, in table 2 with the same split for training and test set, there are several tables in which the RMSE error is similar to ours, considering the same parameters (*lambda = 0.1* and *iterations* = 10). The rest of them are all higher than our error values except for that one resulting from an experiment with *lambda* = 0.1 and *iterations* = 20 that is better than ours for 0.01.

4.4 Baseline.py

This code represents a naive algorithm against which to compare the Collaborative-Filtering algorithm. In this case, no Machine Learning algorithm has been used. The output is the best 25 movies of Mystery genre, obtained performing an average of the ratings given by different users.

Regarding the data preprocessing, the same steps already illustrated in section Main.py have been performed. At the end, we have two RDDs ready for the baseline approach.

```
conf=SparkConf().setAppName("MoviesReccomandations baseline")
sc=SparkContext(conf=conf)
sc.setLogLevel("ERROR")
sqlContext=SQLContext(sc)

n_partitions = int(sys.argv[1])

ratings_data_temp=sc.textFile("s3://bucketbigdataemr/files/ratings.csv",n_partitions)
ratings_data_header=ratings_data_temp.take(1)[0]
ratings_data = ratings_data_temp.filter(lambda line: line!=ratings_data_header)\
    .map(lambda line: line.split(",")).map(lambda tokens: (int (tokens[1]),float(tokens[2]))).cache()

movie_data_temp = sc.textFile("s3://bucketbigdataemr/files/movies.csv", n_partitions)
movie_data_header = movie_data_temp.take(1)[0]
movie_data = movie_data_temp.filter(lambda line: line != movie_data_header) \
    .map(lambda line: line.split(",")).map(lambda tokens: (int(tokens[0]), tokens[1],
        tokens[2])).cache()
```

At this point, the *movie_data* RDD has been filtered taking only the movies with *Mystery* genre.

```
movie_data = movie_data.filter(lambda x: "Mystery" in x[2]).cache()
```

Afterwards, the join between the ratings and movie set has been done, and the result were mapped in order to scale tuples one level. Note that if you have python3, you have to change the + symbol to the * symbol in the *map* method.

```
temp_RDD=ratings_data.join(movie_data).map(lambda x: (x[0], ) +x[1])
```

In order to simplify the average operation the RDD has been temporally converted into a DataFrame, specifying the proper schema.

```
df_movie=temp_RDD.toDF(schema=["movie_id", "rating", "title"])
df_avg=df_movie.groupby("title").avg("rating")
```

At the end, the Dataframe has been reconverted into a RDD. The code returns the 25 best recommendations on the base of the best average on the Mystery movies.

```
topMistery_movies=df_avg.rdd.takeOrdered(25, key=lambda x: -x[1])
print('Top Average Mistery Movie:\n%s' % '\n'.join(map(str, topMistery_movies)))
```

The baseline final output is the following:

```
Top Movie Recommandation:
Row(title=u'Hush Little Baby (2007)', avg(rating)=5.0)
Row(title=u'The Duel (1981)', avg(rating)=5.0)
Row(title=u'Loser Takes All! (Qui perd gagne !) (2003)', avg(rating)=5.0)
Row(title=u'Duel of Hearts (1992)', avg(rating)=5.0)
Row(title=u'Hands of a Murderer (1990)', avg(rating)=5.0)
Row(title=u'Hart to Hart: Till Death Do Us Hart (1996)', avg(rating)=5.0)
Row(title=u'London Conspiracy (1974)', avg(rating)=5.0)
Row(title=u'Roxy Hunter and the Secret of the Shaman (2008)', avg(rating)=5.0)
Row(title=u"A Sister's Revenge (2013)", avg(rating)=5.0)
Row(title=u"A Daughter's Nightmare (2014)", avg(rating)=5.0)
Row(title=u'Despite the Night (2015)', avg(rating)=5.0)
Row(title=u'Love Finds You In Sugarcreek (2014)', avg(rating)=5.0)
Row(title=u'Poison (2000)', avg(rating)=5.0)
Row(title=u'Witness for the Prosecution (1982)', avg(rating)=5.0)
Row(title=u"A Sister's Secret (2009)", avg(rating)=5.0)
Row(title=u'The Perfect Assistant (2008)', avg(rating)=5.0)
Row(title=u'The Good Mother (2013)', avg(rating)=5.0)
Row(title=u'The Capture of the Green River Killer (2008)', avg(rating)=5.0)
Row(title=u'Say Nothing (2001)', avg(rating)=5.0)
Row(title=u'Roxy Hunter and the Myth of the Mermaid (2008)', avg(rating)=5.0)
Row(title=u'Tainted Blood (1993)', avg(rating)=5.0)
Row(title=u'Sleep Has Her House (2017)', avg(rating)=5.0)
Row(title=u'Murder She Baked: A Deadly Recipe (2016)', avg(rating)=4.75)
Row(title=u'Gourmet Detective (2015)', avg(rating)=4.75)
Row(title=u'Death Al Dente: A Gourmet Detective Mystery (2016)', avg(rating)=4.75)
```

5 How to launch the project

In order not to have problems, it's suggested having the last version of Terraform (at the moment it's 0.12.26). The main commands are:

- *terraform init* inside the folder where there is the *main.tf* file to initialize a working directory containing configuration files.
- Then, you can launch *terraform plan* in order to check which objects will be created.
- Afterwards, so as to apply this plan, launch the command *terraform apply -auto-approve* (this action will take several minutes).
- Finally, if you want to shoot down your infrastructure, launch the command *terraform destroy -auto-approve* (*-auto-approve* allows to skip interactive approval of the plan). Sometimes you have to manually destroy the S3 bucket from the AWS console.

6 How to check the project status

Once the infrastructure has been built, you can check the status of the steps in different sections:

- If you want to check the output results of the python codes, you can go to the AWS console, search "EMR"and click on the "Phases"tab. Each steps has a log section where you can find the output results.
- If you want to check which and how the slaves are running inside the cluster, you can go to the "Summary"tab, and below "Application user interfaces", there are two links, one for the Spark UI and the second one for the Hadoop UI.
- If you want to check the output query, you can go to the Athena service, then on "History", where will be all the executed queries, and on their right the output file. Otherwise, you can go to the S3 service, and inside the output folder specificated in the Terraform *"Execute query"* step you'll find the output file.

7 What do you need to change

In order to run the whole infrastructure, you have to change some parameters inside the Terraform and python code:

- The S3 bucket name, because every bucket is unique. Change it in Amazon S3 section (line 246 in *main.tf*) and its occurrences.
- The number of executors and partitions, in order to check the performance with different cluster architecture. Change them in the cluster steps with name *Run Collaborative-Filtering* and *Run baseline* in Our goal and run steps section (line 590 and 605 in *main.tf*). The executors are the number of slaves while the number of partitions in which to split the RDDs is equal to n° executors × n° cores (4 both for m4.large and c5.xlarge).
- The AWS credentials inside the *aws* provider (Setup Environment section) for linking your account to the script (line 237 in *main.tf*)
- The instance name that you want to use. Go to Amazon EMR section inside the *aws_emr_cluster* resource (line 502 and 507 in *main.tf*). If you want to change the number of slaves inside the EMR cluster, you have to change the parameter *instance_count* (line 503 in *main.tf* - each EC2 instance has a proper vCPU limit)
- If you want to execute only the Collaborative-Filtering with the baseline or also with the query executed by Athena, you have to comment some code inside the *main.tf* file. In particular, if you want to execute only the Collaborative-Filtering and the baseline, comment from 263 to 477 and the steps inside the cluster from 615 to 724.
- Inside the *methods.py* and *baseline.py* files, in order to get the CSV files from S3, it's necessary to change the path given inside *sc.textFile()* with your S3 bucket name. Edit those in lines 15 and 37 in *methods.py* and in lines 31 and 48 in *baseline.py*.

8 Test

Several tests have been computed in order to check the performance of the cluster. These tests have been computed using two different types of instances:

- **m4.large:**
  - 2,4 GHz Intel Xeon E5-2676 v3
  - 2 vCPU
  - 8 GiB
  - EBS bandwidth 450 (Mb/s)
- **c5.xlarge:**
  - Custom 2nd generation Intel Xeon Scalable Processors (Cascade Lake) with a sustained all core Turbo frequency of 3.6GHz and single core turbo frequency of up to 3.9GHz or 1stgeneration Intel Xeon Platinum 8000 series (Skylake-SP) processor with a sustained all core Turbo frequency of up to 3.4GHz, and single core turbo frequency of up to 3.5 GHz.
  - 4 vCPU
  - 8 GiB
  - EBS Bandwidth up to 4,750 (Mbps)

The m4.large instances have been chosen for their balance of compute, memory, and network resources, whereas the c5.xlarge instances, which are better for the ML tasks, have been used in order to check how the performance would have changed compare to the m4.large.

8.0.1 Test with m4.large

Table 1: Test on the cluster that executes only the Collaborative Filtering

EMR Architecture	N° Cores	N° Partitions (n° executors × n° cores)	Collaborative Filtering
1 master - 2 slaves	4	8	13 min
1 master - 4 slaves	4	16	6.8 min
1 master - 6 slaves	4	24	5.5 min
1 master - 8 slaves	4	32	4.6 min

The following two tables show the performance of the Collaborative-Filtering with the query inside the python code and with the query executed by Athena.

Table 2: Test on the cluster that executes the Collaborative Filtering with the query inside the python code

EMR Architecture	N° Cores	N° Partitions (n° executors × n° cores)	Collaborative Filtering with query
1 master - 2 slaves	4	8	13 min
1 master - 4 slaves	4	16	6.8 min
1 master - 6 slaves	4	24	5.7 min
1 master - 8 slaves	4	32	4.7 min

Table 3: Test on the cluster that perform the Collaborative Filtering with the query executed by Athena

EMR Architecture	N° Cores	N° Partitions (n° executors × n° cores)	Collaborative Filtering with Athena
1 master - 2 slaves	4	8	13 min + 15.25s
1 master - 4 slaves	4	16	7.0 min + 14.49s
1 master - 6 slaves	4	24	5.7 min + 16.5s
1 master - 8 slaves	4	32	4.3 min + 15.08s

The + operator in the fourth field adds the execution time of the Collaborative-Filtering and the execution time of the query executed by Athena.

Comparing the last two tests, you can see that the results are pretty similar. Athena takes a few more seconds because it have to print and save the results on S3.

8.0.2 Test with c5.xlarge

Test using *c5.xlarge* instances.

Table 4: Test on the cluster that executes only the Collaborative Filtering

EMR Architecture	N° Cores	N° Partitions (n° executors × n° cores)	Collaborative Filtering
1 master - 2 slaves	4	8	5.0 min
1 master - 4 slaves	4	16	3.0 min
1 master - 6 slaves	4	24	2.5 min
1 master - 7 slaves	4	28	2.3 min

Table 5: Test on the cluster that executes the Collaborative Filtering with the query inside the python code

EMR Architecture	N° Cores	N° Partitions (n° executors × n° cores)	Collaborative Filtering with query
1 master - 2 slaves	4	8	5.1 min
1 master - 4 slaves	4	16	3.1 min
1 master - 6 slaves	4	24	2.5 min
1 master - 7 slaves	4	28	2.3 min

Table 6: Test on the cluster that perform the Collaborative Filtering with the query executed by Athena

EMR Architecture	N° Cores	N° Partitions (n° executors × n° cores)	Collaborative Filtering with Athena
1 master - 2 slaves	4	8	5.0 min + 18.42s
1 master - 4 slaves	4	16	3.0 min + 17.23s
1 master - 6 slaves	4	24	2.5 min + 16.66s
1 master - 7 slaves	4	28	2.3 min + 16.40s

As you can see from the above tables the *c5.xlarge* instances perform the Collaborative-Filtering task with about half the time, so for this kind of ML oriented approaches, these instances are more suitable. We can see that the execution times in the last two tests are almost equal, but if we compare the Athena execution time with those computed by *m4.large*, it takes more time.

8.0.3 Test with baseline algorithm

Table 7: Test on the cluster with m4.large instances that executes the baseline algorithm

EMR Architecture	N° Cores	N° Partitions (n° executors × n° cores)	Baseline Algorithm
1 master - 2 slaves	4	8	3.6 min
1 master - 4 slaves	4	16	2.2 min
1 master - 6 slaves	4	24	1.8 min
1 master - 8 slaves	4	32	1.7 min

Table 8: Test on the cluster with c5.xlarge instances that executes the baseline algorithm

EMR Architecture	N° Cores	N° Partitions (n° executors × n° cores)	Baseline Algorithm
1 master - 2 slaves	4	8	1.6 min
1 master - 4 slaves	4	16	57 s
1 master - 6 slaves	4	24	49 s
1 master - 7 slaves	4	28	44 s

Two comparisons have been made:

1. **Qualitative** in order to compare the results between the Collaborative and baseline approach.
2. **Quantitative** in order to compare the execution time of these two approaches.

Summarize, the Collaborative-Filtering returns the best 25 recommendations for a new user, while the baseline approach returns the best 25 *Mystery* movies recommendations. From the point of view of the result, the Collaborative-Filtering returns more accurate recommendations than the baseline approach because it has been performed with ML, whose aim is to make predictions on the base of similarity got by a certain calculations depending on the dataset and the algorithm that you have to develop.

To do that, the training and test set are two really important components that for the baseline approach can't be used. For this reason the final output of the baseline algorithm is performed in a general way, without considering the users interests and for that, performed with only an average operation on the whole set of Mystery movies.

In conclusion, we can say that, if we have to think to the Collaborative-Filtering goal, i.e. getting the best 25 recommendations on the base of the interest of a new user, the result of this approach is better than the baseline. On the other hand, if we consider only a recommendation list given by the average of all the users' ratings and so without considering the preferences of the new user, the baseline approach is better and faster than the Collaborative-Filtering.

Hence, we can consider two scenarios:

- A user wants to see the movies recommended for him: use the Collaborative approach
- A user wants to see the top movies on a specific genre: use the baseline approach

Instead, regarding the quantitative comparison, the baseline approach is faster than the Collaborative approach because there aren't any ML operations. The time is about 1/3 less than the Collaborative approach both for *m4.large* and *c5.xlarge*.

9 Github Repository

You can find the github repository at the following link:

`https://github.com/sserra74/Collaborative-Filtering.git`

Bibliography

[1] Movie Recommender System Based on Collaborative Filtering Using Apache Spark, https://www.researchgate.net/publication/326146390_Movie_Recommender_System_Based_on_Collaborative_Filtering_Using_Apache_Spark

[2] Comparative Analysis Of Movie Recommendation System Using Collaborative Filtering In Spark Engine, https://www.researchgate.net/publication/321058803_COMPARATIVE_ANALYSIS_OF_MOVIE_RECOMMENDATION_SYSTEM_USING_COLLABORATIVE_FILTERING_IN_SPARK_ENGINE