Collaborative Filtering for the Prediction of Drug-Target Associations

Mirco Perna

A.A. 2024/2025

1 Introduction

The aim of the project is to study and develop Collaborative Filtering, a technique used in Recommender Systems to predict a user's interests based on the preferences of other users with similar tastes, to predict new targets for a given drug, based on the protein-protein interactions (PPI).

The main idea is to identify new targets for existing drugs by searching for among 'neighbors' of their known targets within the associated protein-protein interaction network. It is important to note that not every protein that physically interacts with a drug's known targets should necessarily be considered a potential target itself. However, if a set of known drug-target associations is used to train the recommendation system, only those proteins that exhibit behavior similar to that of actual targets are likely to be identified.

2 Collaborative Filtering

2.1 Recommendation systems

Recommendation systems are made up of filtering algorithms that aim to predict a rating or preference a user would assign to a given item, which have become increasingly important across a variety of commercial domain.

These systems generally produce recommendations through one of two methods:

1. Content based filtering
2. Collaborative based filtering

Content-based filtering techniques use attributes of an item in order to recommend future items with similar attributes. Collaborative filtering builds a model from a user's past behavior, activities, or preferences and makes recommendations to the user based on similarities with other users. However, collaborative filtering methods generally offer higher accuracy than content-based approaches.

2.2 Approaches

Collaborative filtering is broken down into two primary approaches: model-based and memory-based.

The **memory-based** approaches take user rating data to compute similarities between users and items in order to make a recommendation. The most famous memory-based approach are neighborhood-based algorithms. One of the most commonly used algorithms for this approach is the K-nearest neighbor (KNN) algorithm.

The **model-based** (or matrix factorization based) approaches build models based on modern machine learning algorithms discovering patterns in the training data. The models are then used to make predictions on real data.

2.3 Matrix Factorization based

Matrix factorization is a simple embedding model. Given the feedback matrix

$$A \in \mathbb{R}^{M \times N}$$

where M is the number of users and N is the number of items. The model learns:

- A user embedding matrix
$$U \in \mathbb{R}^{M \times D}$$
- A item embedding matrix
$$V \in \mathbb{R}^{D \times N}$$

The value of the dimension D is a system parameter and it is called latent factors. The embeddings are learned such that the product of matrix U and the transpose of matrix V is a good approximation of the feedback matrix A.

$$A \approx UV^\top$$

Observe that (i, j) the entry of $UV^\top$ is simply the dot product $< U_i \cdot V_j >$ of the embeddings of user i and item j. Ideally $a_{i,j} =< u_i, v_j > \forall i, j$ but in practice we need to minimize the loss function, which in this case is the root mean square error (RMSE)

$$\text{RMSE} = \sqrt{\frac{1}{n}\sum_{i=1}^{n}(p_{u,v} - a_{u,v})^2}$$

where $p_{u,v}$ and $r_{u,v}$ are respectively predicted and observed ratings.

2.4 Alternating Least Square

The chosen algorithm is Alternating Least Squares (ALS), which performs well on large and sparse datasets. The ALS rotates between fixing one of the unknowns u_i or v_j. When one is fixed the other can be computed by solving the least-squares problem. This approach is useful because it turns the previous non-convex problem into a quadratic that can be solved optimally

Algorithm 1 ALS

Step 1: Initialize matrix V by assigning the average rating for that item as the first row, and small random numbers for the remaining entries.

Step:2 Fix V, solve U by minimizing the RMSE function.

Step:3 Fix U, solve V by minimizing the RMSE function similarly.

Step:4 Repeat Steps 2 and 3 until convergence.

3 Experiment

3.1 Dataset

The Data Set was built manually and is composed by two sets of pairs:

- Protein-Protein interactions (PPI)
$$< Protein, Protein >$$
- Drug-Target associations
$$< Drug, Protein >$$

After the two sets were built, they were inserted in an instance of MongoDB, a document database used to build highly available and scalable internet applications, and it uses flexible schema approach. The document used to create the database is shown below.

```
{
    "_id": "collaborative_filtering",
    "PPI": [
        { "ID_Protein_A": "PT0001", "ID_Protein_B": "PT1002" },
        { "ID_Protein_A": "PT0001", "ID_Protein_B": "PT1004" },
        { "ID_Protein_A": "PT0001", "ID_Protein_B": "PT1005" },
        { "ID_Protein_A": "PT0002", "ID_Protein_B": "PT1004" },
        { "ID_Protein_A": "PT0003", "ID_Protein_B": "PT1005" },
        { "ID_Protein_A": "PT0003", "ID_Protein_B": "PT1004" },
        { "ID_Protein_A": "PT0004", "ID_Protein_B": "PT1001" },
        { "ID_Protein_A": "PT0005", "ID_Protein_B": "PT1006" },
        { "ID_Protein_A": "PT0005", "ID_Protein_B": "PT1000" },
        { "ID_Protein_A": "PT0006", "ID_Protein_B": "PT1001" },
        { "ID_Protein_A": "PT0007", "ID_Protein_B": "PT1005" },
        { "ID_Protein_A": "PT0007", "ID_Protein_B": "PT1003" },
        { "ID_Protein_A": "PT0008", "ID_Protein_B": "PT1003" },
        { "ID_Protein_A": "PT0009", "ID_Protein_B": "PT1001" },
        { "ID_Protein_A": "PT0009", "ID_Protein_B": "PT1005" },
        { "ID_Protein_A": "PT0010", "ID_Protein_B": "PT1003" },
        { "ID_Protein_A": "PT0011", "ID_Protein_B": "PT1005" },
        { "ID_Protein_A": "PT0011", "ID_Protein_B": "PT1003" },
        { "ID_Protein_A": "PT0011", "ID_Protein_B": "PT1002" },
        { "ID_Protein_A": "PT0012", "ID_Protein_B": "PT1000" },
        { "ID_Protein_A": "PT0013", "ID_Protein_B": "PT1002" },
        { "ID_Protein_A": "PT0013", "ID_Protein_B": "PT1003" },
        { "ID_Protein_A": "PT0014", "ID_Protein_B": "PT1005" },
        { "ID_Protein_A": "PT0015", "ID_Protein_B": "PT1006" },
        { "ID_Protein_A": "PT0015", "ID_Protein_B": "PT1005" },
        { "ID_Protein_A": "PT0019", "ID_Protein_B": "PT1000" },
        { "ID_Protein_A": "PT0019", "ID_Protein_B": "PT1001" },
        { "ID_Protein_A": "PT0020", "ID_Protein_B": "PT1001" }
    ],
    "Drug_Target": [
        { "ID_DrugBank": "DB0001", "ID_Target": "PT0001" },
        { "ID_DrugBank": "DB0001", "ID_Target": "PT0002" },
        { "ID_DrugBank": "DB0001", "ID_Target": "PT0003" },
        { "ID_DrugBank": "DB0001", "ID_Target": "PT0004" },
        { "ID_DrugBank": "DB0001", "ID_Target": "PT0005" },
        { "ID_DrugBank": "DB0002", "ID_Target": "PT0006" },
        { "ID_DrugBank": "DB0002", "ID_Target": "PT0007" },
        { "ID_DrugBank": "DB0002", "ID_Target": "PT0008" },
        { "ID_DrugBank": "DB0002", "ID_Target": "PT0009" },
        { "ID_DrugBank": "DB0002", "ID_Target": "PT0010" },
        { "ID_DrugBank": "DB0003", "ID_Target": "PT0011" },
        { "ID_DrugBank": "DB0003", "ID_Target": "PT0012" },
        { "ID_DrugBank": "DB0003", "ID_Target": "PT0013" },
        { "ID_DrugBank": "DB0003", "ID_Target": "PT0014" },
        { "ID_DrugBank": "DB0003", "ID_Target": "PT0015" },
        { "ID_DrugBank": "DB0004", "ID_Target": "PT0016" },
        { "ID_DrugBank": "DB0004", "ID_Target": "PT0017" },
        { "ID_DrugBank": "DB0004", "ID_Target": "PT0018" },
        { "ID_DrugBank": "DB0004", "ID_Target": "PT0019" },
        { "ID_DrugBank": "DB0004", "ID_Target": "PT0020" },
        { "ID_DrugBank": "DB0005", "ID_Target": "PT0001" },
        { "ID_DrugBank": "DB0005", "ID_Target": "PT0006" },
        { "ID_DrugBank": "DB0005", "ID_Target": "PT0011" },
        { "ID_DrugBank": "DB0005", "ID_Target": "PT0016" },
        { "ID_DrugBank": "DB0005", "ID_Target": "PT0020" }

    ]
}
```

3.2 Code

Python was used as a programming language and the following libraries:

- PySpark
- Numpy
- PyMongo

The main steps of the experiment are as follows, and they are described in detail in the following paragraphs:

1. Obtain the two sets by MongoDB
2. Process the sets to obtain the dataset for training the model
3. Training the model
4. Evaluate the recommended protein targets for the drugs

3.3 DataAccess

In order to obtain the sets from the database, the DataAccess class was implemented. It contains the methods get_PPIs and get_drug_target_interactions, which are used to extract the sets from the database and return a list for each of them, respectively.

```
from pymongo import MongoClient
from PPI import PPI
from drug_target import DrugTarget

class DataAccess():
    def __init__(self):
        self.client = MongoClient("mongodb://localhost:27017/")
        self.database = self.client["Biological_Network"]
        self.collection = self.database["Collaborative_Filtering"]
        self.id_collection = "collaborative_filtering"


    def get_PPIs(self):
        data = self.collection.find_one({"_id":self.id_collection})

        PPIs = []
        for interaction in data["PPI"]:
            new_ppi = PPI()
            new_ppi.set_properties(interaction["ID_Protein_A"], interaction["
                ID_Protein_B"])
            PPIs.append(new_ppi)

        return PPIs

    def get_drug_target_interactions(self):
        data = self.collection.find_one({"_id":self.id_collection})

        drug_target_interactions = []
        for interaction in data["Drug_Target"]:
            new_drug_target = DrugTarget()
            new_drug_target.set_properties(interaction["ID_DrugBank"],
                interaction["ID_Target"])
            drug_target_interactions.append(new_drug_target)

        return drug_target_interactions

    def close_connection(self):
        self.client.close()
```

3.4 DataFrame

In order to obtain the dataset required to train and test our model, the class DataFrame was implemented.

```
from pyspark.sql import SparkSession
from pyspark.sql.functions import collect_list
from pyspark.sql.functions import lit
from pyspark.sql.functions import expr
from pyspark.ml.feature import StringIndexer
from pyspark.ml import Pipeline

class DataFrame():

    def __init__(self, PPIs, drug_taget_interactions):
        self.spark = SparkSession.builder \
                .appName("Collaborative_Filtering") \
                .config("spark.driver.host", "localhost") \
                .config("spark.driver.bindAddress", "127.0.0.1") \
                .getOrCreate()

        self.spark.sparkContext.setLogLevel("ERROR")

        self.create_ppi_dataframe(PPIs)

        self.create_drug_taget_interactions_dataframe(drug_taget_interactions)


    def create_ppi_dataframe(self,PPIs):
        PPI_list = [PPI.to_list() for PPI in PPIs]
        self.PPIs_df = self.spark.createDataFrame(PPI_list, ["ID_Protein_A", "
            ID_Protein_B"])

    def create_drug_taget_interactions_dataframe(self,drug_taget_interactions)
        :
        drug_target_interactions_list = [interaction.to_list() for interaction
             in drug_taget_interactions]
        self.drug_taget_interactions_df = self.spark.createDataFrame(
            drug_target_interactions_list, ["ID_DrugBank", "ID_Target"])

    def create_date_frame_for_als(self):
        def create_drug_interactions():
            self.drug_interactions_df = self.drug_taget_interactions_df
                            .groupBy("ID_DrugBank")
                            .agg(collect_list("ID_Target")
                            .alias("Proteins"))
                            .orderBy("ID_DrugBank")

        def join_drug_target_with_ppi():
            self.joined_df = self.drug_taget_interactions_df.join(self.PPIs_df
                , self.drug_taget_interactions_df.ID_Target == self.PPIs_df.
                ID_Protein_A)
            self.joined_df = self.joined_df.withColumnRenamed("ID_DrugBank", "
                ID_Drug")

        def filter_interactions():
            self.joined_df = self.joined_df.join(self.drug_interactions_df,
                self.joined_df.ID_Drug == self.drug_interactions_df.
                ID_DrugBank)
            self.joined_df = self.joined_df.select(self.joined_df["ID_Drug"],
                self.joined_df["ID_Target"], self.joined_df["ID_Protein_A"],
                self.joined_df["ID_Protein_B"], self.joined_df["Proteins"])
```

```
47
48              self.joined_df = self.joined_df.withColumn("Interactor_drug_target
                    ", expr("array_contains(Proteins, ID_Protein_B)"))
49
50              self.joined_df = self.joined_df.filter(self.joined_df.
                    Interactor_drug_target == False)
51
52          def calculate_interaction():
53              columns_to_group_by = ["ID_Drug", "ID_Protein_B"]
54              self.joined_df = self.joined_df
55                               .groupBy(columns_to_group_by)
56                               .count()
57
58              self.joined_df = self.joined_df
59                               .withColumnRenamed("count", "Interactions")
60              self.joined_df = self.joined_df
61                               .withColumnRenamed("ID_Protein_B", "ID_Protein")
62
63              create_drug_interactions()
64
65              join_drug_target_with_ppi()
66
67              filter_interactions()
68
69              calculate_interaction()
```

The role of each method will not be explained in detail; instead, the focus will be on the general process that leads to the construction of the final dataset which consists of the following steps:

1. Create the PPIs_df and drug_target_interactions_df DataFrames.
2. Create the drug_interactions_df DataFrame, where the first column contains the drug and in the other column the list of proteins associated with that drug.
3. Create the joined_df DataFrame which is generated by the join of PPIs_df and drug_target_interactions_df DataFrames, where the ID_Target matches ID_Protein_A
4. Join the joined_df and drug_interactions_df DataFrames, where the ID_Drug matches ID_DrugBank. Then, use a select operation to extract the columns: ID_Drug, ID_Target, ID_Protein_A, ID_Protein_B and Proteins
5. Remove all the records where ID_Protein_B is in Proteins
6. Using a group-by operation on the ID_Target and ID_Protein_B columns, we compute the value of the Interactions column, which represents the number of target proteins associated with the drug ID_Target that interact with the protein ID_Protein_B. This value will be used as a rating for the recommendation system.
7. Finally, perform a select operation to extract the columns ID_Drug , ID_Protein_B, which will be renamed to ID_Protein, and Interactions in order to obtain the final DataFrame for the model.

3.5 ALSModel

The ALSModel class includes methods to train the model and to evaluate new potential targets. The ALSModel class includes methods to train the model and to evaluate new potential targets. Since the training values must be of type Integer or Float, the StringIndexer function from PySpark was used to encode all string-type fields. However, as this function can only be applied to one column at a time, and the resulting indexes must remain distinct across columns,the Pipeline function from PySpark was used to manage and apply the transformations efficiently. The indexing_name method is responsible for all these operations.
During training, a evaluation is performed to find the optimal hyperparameters that minimize the RMSE. The model requires the following hyperparameters:

Hyperparameter	Description
maxIter	Maximum number of iterations to run
regParam	Specifies the regularization parameter in ALS
rank	Number of latent factors in the model
alpha	Parameter applicable to the implicit feedback variant of ALS that governs the baseline confidence in preference observations
coldStartStrategy	Strategy to manage new or unknown items/users. Setting it to 'drop' excludes these items from the results

To evaluate the RMSE value for the chosen regParam, rank and alpha parameters the RegressionEvaluator function is used.
Finally, the method calculate_recommended_proteins evaluates new potential targets.

```
from pyspark.ml.recommendation import ALS
from pyspark.ml.evaluation import RegressionEvaluator
from pyspark.sql.functions import explode
from pyspark.sql.functions import first
from pyspark.ml.feature import StringIndexer
from pyspark.ml.feature import IndexToString
from pyspark.ml import Pipeline

class ALSModel():
    def __init__(self, data):
        self.indexing_name(data)

    def indexing_name(self,data):
        self.drug_indexer = StringIndexer(inputCol = "ID_Drug", outputCol = "
            ID_Drug_Index").fit(data)
        self.prontein_indexer = StringIndexer(inputCol = "ID_Protein",
            outputCol = "ID_Protein_Index").fit(data)
        pipeline = Pipeline(stages = [self.drug_indexer, self.prontein_indexer
            ])
        self.data = pipeline.fit(data).transform(data)
        self.data.show()

    def train(self):
        (training, test) = self.data.randomSplit([0.8, 0.2], seed=42)

        regParams = [0.01, 0.1]
        ranks = [25,30,35]
        alphas = [10.0, 20.0, 40.0, 60.0, 80.0, 100.0]

        self.aus_regParam = 0.0
        self.aus_rank = 0
        self.aus_alpha = 0.0
```

```
        self.aus_rmse = 0.0

        for regParam in regParams:
            for rank in ranks:
                for alpha in alphas:
                    aus_als = ALS(maxIter = 10, regParam = regParam, rank =
                        rank, alpha = alpha, userCol = "ID_Drug_Index",
                                  itemCol = "ID_Protein_Index", ratingCol = "
                                      Interactions",coldStartStrategy = "drop"
                                      )

                    aus_model = aus_als.fit(training)
                    predictions = aus_model.transform(test)
                    evaluator = RegressionEvaluator(metricName = "rmse",
                        labelCol = "Interactions", predictionCol = "prediction
                        ")
                    rmse = evaluator.evaluate(predictions)

                    if(self.aus_rmse == 0.0 or rmse < self.aus_rmse):
                        self.aus_regParam = regParam
                        self.aus_rank = rank
                        self.aus_alpha = alpha
                        self.aus_rmse = rmse
                        self.model = aus_model

                    print("For regParam: {0}, rank:{1}, alpha:{2}, RMSE:{3}".
                        format(regParam, rank, alpha, rmse))

        print("Chosen parameters: regParam: {0}, rank:{1}, alpha:{2}, RMSE:{3}
            ".format(self.aus_regParam, self.aus_rank, self.aus_alpha, self.
            aus_rmse))

    def from_index_to_name(self, proteins_recommended):
            drug_name = IndexToString(inputCol = "ID_Drug_Index", outputCol =
                "ID_Drug", labels = self.drug_indexer.labels)

            prontein_name= IndexToString(inputCol = "ID_Protein_Index",
                outputCol = "ID_Protein",

            labels = self.prontein_indexer.labels)

            pipeline = Pipeline(stages = [drug_name, prontein_name])

            self.drug_proteins_recommended = pipeline.fit(self.data)
            .transform(proteins_recommended)

            self.drug_proteins_recommended =   self.drug_proteins_recommended.
                select("ID_Drug","ID_Protein","rating").orderBy("ID_Drug","
                rating")

    def calculate_recommended_proteins(self):
            self.train()

            amount_proteins_for_drug = 4

            proteins_recommended = self.model.recommendForAllUsers(
                amount_proteins_for_drug)

            proteins_recommended = proteins_recommended.withColumn("
                proteinAndRating", explode(proteins_recommended.
                recommendations))
            .select("ID_Drug_Index", "proteinAndRating.*")
```

```
77
78                  self.from_index_to_name(proteins_recommended)
```

3.6 Results

The calculate_recommended_proteins function evaluates the top protein recommendations for each drug. The following figure shows the predicted proteins for the DB0003 drug are actually those with the highest Interactions values.

```
+-------+----------+---------+
|ID_Drug|ID_Protein|   rating|
+-------+----------+---------+
| DB0003|    PT1005|2.9748125|
| DB0003|    PT1004| 2.027082|
| DB0003|    PT1002|2.0130541|
| DB0003|    PT1003|2.0003428|
| DB0003|    PT1001|1.6661369|
| DB0003|    PT1000|1.0055522|
| DB0003|    PT1006|0.6756939|
+-------+----------+---------+

+-------+----------+------------+
|ID_Drug|ID_Protein|Interactions|
+-------+----------+------------+
| DB0003|    PT1005|           3|
| DB0003|    PT1003|           2|
| DB0003|    PT1002|           2|
| DB0003|    PT1006|           1|
| DB0003|    PT1000|           1|
+-------+----------+------------+
```

Finally, the following image shows both the table of actual values and the table of predicted values, highlighting the small differences between them.

```
+-------+------+------+------+------+------+------+------+
|ID_Drug|PT1000|PT1001|PT1002|PT1003|PT1004|PT1005|PT1006|
+-------+------+------+------+------+------+------+------+
| DB0001|     1|     1|     1|  NULL|     3|     2|     1|
| DB0002|  NULL|     2|  NULL|     3|  NULL|     2|  NULL|
| DB0003|     1|  NULL|     2|     2|  NULL|     3|     1|
| DB0004|     1|     2|  NULL|  NULL|  NULL|  NULL|  NULL|
| DB0005|  NULL|     2|     2|     1|     1|     2|  NULL|
+-------+------+------+------+------+------+------+------+

+-------+---------+---------+---------+---------+---------+---------+---------+
|ID_Drug|   PT1000|   PT1001|   PT1002|   PT1003|   PT1004|   PT1005|   PT1006|
+-------+---------+---------+---------+---------+---------+---------+---------+
| DB0001|0.9926122|1.0018413|1.1318059|0.9785331| 2.984873| 2.000412| 0.9949578|
| DB0002| 0.660511|1.2651672|1.5605621| 2.996085|1.0428447|2.2388773| 0.3476149|
| DB0003|1.0055522|1.6661369|2.0130541|2.0003428| 2.027082|2.9748125| 0.6756939|
| DB0004|0.9903602|1.9968805| 2.082445|1.3171701|1.7327752|2.4978905| 0.5775917|
| DB0005|0.7375037|1.9904449|1.9897145|1.0000297|0.6952576|2.0055344|0.2317525|
+-------+---------+---------+---------+---------+---------+---------+---------+
```