MIDDLE EAST TECHNICAL UNIVERSITY

Department Of Computer Engineering

CENG 300

Summer Practice Report

METU Data Mining Research Group
Start Date: End Date:
Total Working Dates:

October 6, 2020

Student:
Onat ÖZDEMİR

Supervisors:
Prof.Dr. Pınar KARAGÖZ

Prof.Dr. İsmail Hakkı TOROSLU

Student's Signature

Organization Approval

Contents

1 Introduction

I have done my summer internship at METU Data Mining Research Group under the supervision of Prof.Dr.Pınar KARAGÖZ and Prof.Dr.İsmail Hakkı TOROSLU. The task I have worked on was implementing a Trust Based Recommender using the collaborative filtering method and testing it on provided dataset. The dataset contains information about customers and the products they have bought. In addition to dataset, I was able to use eigentrust calculation and community detection modules provided by the TACOREC.

2 Project

During the internship, I implemented two trust based recommenders with different weighting methods:

1. Eigentrust Weighted Recommender
2. Inverse Distance Weighted Recommender

The details of these two recommenders can be found in section 2.4 and 2.5, respectively.

2.1 Analysis Phase

The definition and calculation of trust may differ in many sources and researches. For instance, [9] approaches the issue from the probabilistic side and calculates trust through successful and unsuccessful transactions while the trust calculation method in [12] is based on "Pearson Correlation Similarity" between users. In addition to given examples, there are also recommenders[11] which use trust scores explicitly given by users to make recommendations.

In our case, although the dataset provided by Stockmount doesn't contain explicit trust information given by customers as in [11], In addition to transaction records in the dataset, I was able to use "Community Detection" and "Eigentrust Calculation" modules provided by TACoRec[6]. The challenging part was finding a way to integrate "Eigentrust scores" to the recommendation process.

For the "Inverse Distance Weighted Recommender" however, I experienced the difficulty of inferring trust from implicit customer behaviors in addition to integrating part.

2.2 Design Phase

2.2.1 For Eigentrust Weighted Recommender

Eigentrust represents how strongly connected the customers are to their communities (for the detailed information, please check section 2.4.1) and stored as a property of the relationship between the customer and his/her community. We can draw two conclusions from this information;

1. We don't need to recalculate trust values, it's already stored in the database and all we need to get it using neo4j driver.

2. Since eigentrust values are community dependent, rather than iterating over customer list and storing all the eigentrust values in a huge matrix, we can iterate over communities and create eigentrust array for the members of each community.

Based on these conclusions, I designed two modules: Recommender and Filterer. The only task of the Recommender is getting recommendations from Filterer module and writing them to database. The remaining weight of the project is carried by the Filterer: getting transactions and eigentrust values via neo4j driver, calculating recommendation coefficients for each product, sorting these coefficients and composing recommended items list from k number of products with the highest recommendation coefficients, etc.

For the detailed information and the recommender structure, please check section 2.4 and Figure 1.

2.2.2 For Inverse Distance Weighted Recommender

"Inverse Distance Weighting" method is inspired by an article[10] that suggests calculating the trust scores between two customers by the reciprocal of shortest distance between them on a trust network. Since I worked on datasets with no explicit trust information, I decided to use customer-product records to build my own trust network.

So with this approach, In addition to Filterer and Recommender, I implemented the Graph Module which is responsible from building the graph accoring to preferred method (unweighted or euclidean distance weighted) from transaction records and calculating trust scores using the distances between customers in the graph.

For the detailed information and the recommender structure, please check section 2.5 and Figure 3.

2.3 Implementation Phase

Since there are two different implementations, I have divided the implementational details of the recommenders into two subsections: section 2.4 and 2.5. Under this subsection, libraries and technologies used in implementations are explained.

2.3.1 Neo4j [1]

Driver Installation :

```
pip install neo4j

```

Configuration :

```
import neo4j
...

uri = self._config["database"]["neo4j"]["uri"]
user = self._config["database"]["neo4j"]["user"]
password = self._config["database"]["neo4j"]["password"]

self._driver = neo4j.Driver(uri, auth=(user, password))


```

Sample Usage :

```
import neo4j
...

```

```
  def get_customer_trust(self, customer_id):

    query = (
      f"MATCH (u:Customer)-[r:BELONGS_IN]->(:Community) "
      f"WHERE u.id = {repr(customer_id)} "
      f"RETURN r.eigentrust"
    )

    with self._driver.session() as session:
      return tuple(session.run(query).single())


```

Listing 1: Neo4j driver example

2.3.2 Numpy [2]

Installation :

```
pip install numpy

```

Sample Usage :

```
import numpy as np

class TrustBasedFilterer(object):
...

  def _create_customers_versus_products_table(self):

    self._customers_versus_products_table = np.zeros(
      (self._unique_customers.shape[0],
      self._unique_products.shape[0]),
      dtype=np.bool,
      )

    self._customers_versus_products_table[
      self._sales[:, 0],
      self._sales[:, 1],
    ] = True

```

Listing 2: Numpy example

2.3.3 Scipy [3]

Installation :

```
pip install scipy

```

Sample Usage :

```
from scipy.sparse import csr_matrix
from scipy.sparse.csgraph import dijkstra

class Graph(object):
...

  def _create_distance_matrix(self):

    self._create_adjacency_matrix()

    self._adjacency_matrix = \
      csr_matrix(self._adjacency_matrix)

    self._distance_matrix = dijkstra(
      csgraph=self._adjacency_matrix,
      directed=False,
      return_predecessors=False,
      unweighted=True,
      limit=self._max_distance)

    self._distance_matrix\
      [~np.isfinite(self._distance_matrix)] = 0

```

Listing 3: Scipy example

2.4 Eigentrust Weighted Trust Based Recommender

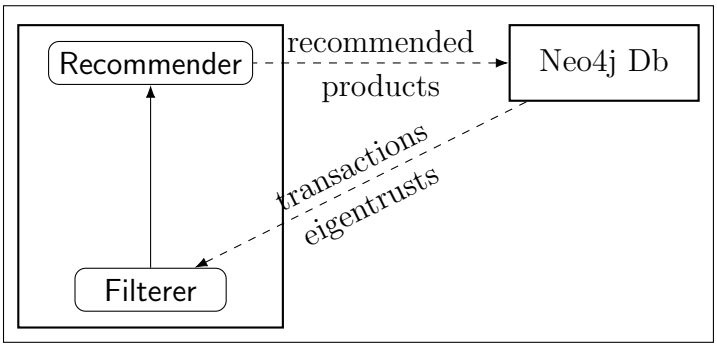

Figure 1: Recommender Structure

2.4.1 About Eigentrust

Eigentrust[9] is a reputation calculation algorithm based on the number of positive and negative transactions between customers and mainly designed for peer-to-peer networks. In our case, Eigentrust represents how strongly connected the customers are to their communities. Eigentrust values calculated by the eigentrust module provided by TACoRec[6] and stored in Neo4j database as a property of the relationship between a customer and his/her community.

Problem encountered with Eigentrust: Especially for the customers connected to communities with small size and low densities, eigentrust values stored in the database are either very small or equal to zero (check Figure 2). Most of the customers with zero eigentrust values are eliminated after filtering the network from customers with a small number of products. Unfortunately, eigentrust values are still quite small.

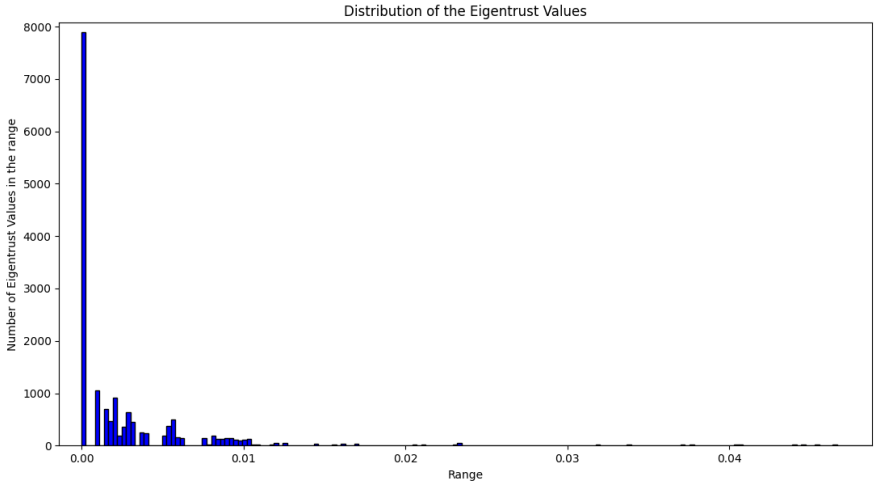

Figure 2: Distribution of the Eigentrust Values. As can be seen nearly 8000 of the customers have zero eigentrust value.

2.4.2 Filterer Module

The Filterer Module is responsible for;

- Cleaning/Filtering the provided transaction list (optional)
- Calculating cosine similarities between customers in the same community
- Calculating weights between customers in the same community
- If the dataset consists of implicit ratings calculating recommendation coefficients otherwise making predictions for products
- Creating recommended products list based on recommendation coefficient/predictions for each customer

2.4.2.1 Calculating Weights

$$w_{c_{target}}(c_2) = \frac{sim(c_{target}, c_2) * trust(c_2)}{sim(c_{target}, c_2) + trust(c_2)}$$

where $sim(c_{target}, c_2)$ represents "cosine similarity" between customers and $trust(c_2)$ represents eigentrust belonged to c_2.

2.4.2.2 Calculating Recommendation Coefficients

$$RC(i) = \frac{\sum_{c \in C} w_{c_{target}}(c) * b_c}{\sum_{c \in C} w_{c_{target}}(c)}$$

where $RC(i)$ represents the recommendation coefficient calculated for product i and b_c is a boolean value which indicates whether the product was bought by person c or not.

2.4.2.3 Making Predictions

$$p(i) = \frac{\sum_{c \in C} w_{c_{target}}(c) * r_c}{\sum_{c \in C} w_{c_{target}}(c)}$$

where $p(i)$ represents the prediction for product i and r_c represents rating given by customer c for product i. C customer set only contains the customers who purchased product i.

Important remark: C community set used in above functions consists only of customers belonged to the same community with c_{target} while there is no such restriction in 2.5.

2.4.3 Recommender Module

Recommender Module is basically responsible from reading/writing. The module has two tasks:

1. Getting transaction list which contains customer id product id pairs from the Neo4j database using neo4j driver and sending the list to the Filterer module as parameter.

2. Getting the recommendation list which contains ids of the customers and corresponding recommended products from the Filterer module and writing these recommendations to Neo4j database as a relationship between the customer and the recommended product using neo4j driver.

2.5 Inverse Distance Weighted Trust Based Recommender

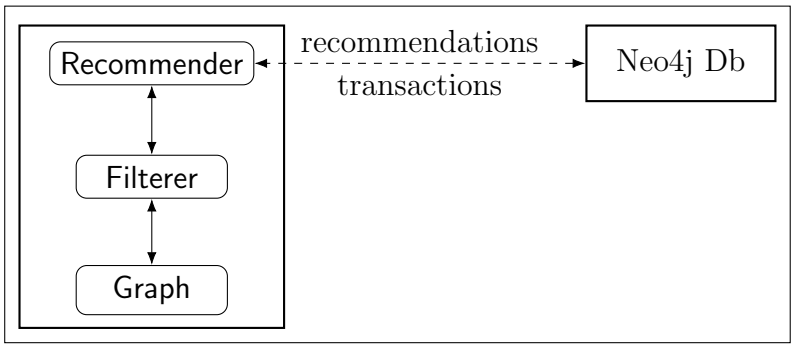

Figure 3: Recommender Structure

Inverse Distance Weighted Trust Based Recommender consists of three modules:

2.5.1 Graph Module

Graph Module is responsible for three tasks:

1. Constructing "adjacency matrix" from "customer versus product table" provided by Filterer module
2. Constructing "distance matrix" using "adjacency matrix"
3. Constructing "trust matrix" using "distance matrix"

2.5.1.1 Constructing Adjacency Matrix and Distance Matrix

Since the recommender is tested in both the datasets with implicit ratings and explicit ratings, to construct the "adjacency matrix" from customer versus products table, I propose two methods:

Proposed Method 1: Unweighted Graph

In this method, the "adjacency matrix" is constructed based on whether customers purchased a joint product or not. In other words, edge between two customers can exist if and only if the intersection of the set of products

they purchased is not the empty set. This method is proposed for especially the datasets with **implicit ratings**.

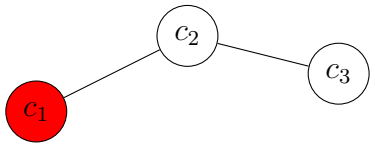

Figure 4: c_1 and c_2 have purchased at least 1 joint product, but c_1 and c_3 do not have a joint product

Problem encountered with Proposed Method 1: During the tests applied on Movielens100k dataset, I observed that although the dataset is sparse, the maximum distance between two users was calculated as 2. Since the distances were distributed in such small range, trust values calculated with this method were not so meaningful. With this observation, I decided to propose another method and use "Unweighted Graph" method in extremely sparse datasets.

Proposed Method 2: Euclidean Distance Weighted Graph
In this method, the "adjacency matrix" is constructed based on the "euclidean distances"1 between customers. This method is proposed for especially the datasets with **explicit ratings**.

$$adj[c_1]][c_2] = \sqrt{\sum_{i \in I_1 \cap I2} (r1_i - r2_i)^2} \tag{1}$$

where $r1_i$ and $r2_i$ represents ratings given by c_1 and c_2 for product i. Unlike the commonly used "euclidean distance" calculation, in this method, only ratings given to joint products are included in the calculation.

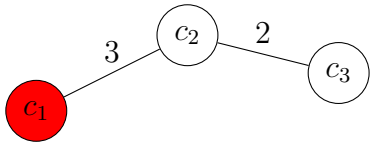

Figure 5: euclidean distance between c_1 and c_2 equals to 3, and c_1 and c_3 do not have a joint product

Dijkstra's Algorithm
To construct the "distance matrix", the graph module uses "Dijkstra's Algorithm"[7] which takes the adjacency matrix as parameter and returns the distance matrix.

2.5.1.2 Trust Calculation

After calculating the shortest distance between each pair of customers using "Dijkstra's Algorithm", to calculate the trust scores between customers Graph module uses

$$T(c_1, c_2) = \begin{cases} \frac{1}{d(c_1,c_2)} & d(c_1, c_2) \neq np.inf \\ 0 & d(c_1, c_2) = np.inf \end{cases}$$

function where $d(c_1, c_2)$ represents the shortest distance between the $customer_1$ and $customer_2$. If $d(c_1, c_2)$ equals $np.inf$ that means either there is no path connecting the customers or the shortest distance between the customers exceeds the distance limit specified in the config file.

A benefit of the method: Especially for excessively sparse datasets, recommenders using euclidean distance-based similarity fails since they cannot calculate similarity score for the the customer pairs with no common products. Since the "Dijkstra's Algorithm" propagates weights even for the customer pairs with no common products, we are able to calculate trust scores between the customers.

2.5.2 Filterer Module

The Filterer Module is responsible for;

- Cleaning/Filtering the provided transaction list (optional)
- Calculating cosine similarities between customers (optional)
- Calculating weights between customers
- If the dataset consists of implicit ratings calculating recommendation coefficients otherwise making predictions for products
- Creating recommended products list based on recommendation coefficient/predictions for each customer

2.5.2.1 Calculating Weights

If 2.5.1.1 being used,

$$w(c_1, c_2) = \alpha * sim(c_1, c_2) + (1 - \alpha) * trust(c_1, c_2)$$

where $sim(c_1, c_2)$ represents "cosine similarity" between customers, $trust(c_1, c_2)$ represents trust calculated by the Graph Module and α is a weight ratio that changes according to the dataset .

Otherwise, $w(c_1, c_2)$ directly equals to $trust(c_1, c_2)$ since 2.5.1.1 method is already based on similarity.

2.5.2.2 Calculating Recommendation Coefficients

$$RC(i) = \frac{\sum_{c \in C} w(c_{target}, c) * b_c}{\sum_{c \in C} w(c_{target}, c)}$$

where $RC(i)$ represents the recommendation coefficient calculated for product i and b_c is a boolean value which indicates whether the product was bought by person c or not.

2.5.2.3 Making Predictions

$$p(i) = \frac{\sum_{c \in C} w(c_{target}, c) * r_c}{\sum_{c \in C} w(c_{target}, c)}$$

where $p(i)$ represents the prediction for product i and r_c represents rating given by customer c for product i. C customer set only contains the customers who purchased product i.

2.5.3 Recommender Module

Recommender Module is basically responsible from reading/writing. The module has two tasks:

1. Getting transaction list which contains customer id product id pairs from the Neo4j database using neo4j driver and sending the list to the Filterer module as parameter.

2. Getting the recommendation list which contains ids of the customers and corresponding recommended products from the Filterer module and writing these recommendations to Neo4j database as a relationship between the customer and the recommended product using neo4j driver.

2.6 Testing Phase

2.6.1 Methods

2.6.1.1 Leave-one-out Cross Validation

2.6.1.2 K-Fold Cross Validation

2.6.1.3 Shuffle Split Cross Validation

2.6.2 Results

2.6.2.1 Stockmount

About the dataset: Dataset origanally consists of 142501 customers and 73482 products with implicit ratings (i.e. purchased or not). However, to make the customer versus product matrix a little denser, customers with less than 2 products and products with less than 2 customers are eliminated. After filtering,

	filtering threshold = 2
Number of Customers	
Number of Products	
Sparsity	99.808124 %

Testing method: Since the dataset contains implicit ratings, I preferred to use "Hit Rate" rather than RMSE or MAE as the test metric and "Leave-one-out CV" as the cross validation iterator. Basically, for each deleted transaction, test module checked whether the product of the deleted transaction is within the recommended products.
Tested Recommender: Eigentrust Weighted Trust Based Recommender 2.4

Results:

	5 Recommendations	10 Recommendations
Number of Tests	4160	4160
Number of Hits	1367	1983
Hit Rate	0.328605	0.476682

2.6.2.2 Movielens100k [8]

About the dataset:

	Movielens 100k
Number of ratings	100.000
Number of users	943
Number of movies	1682
Rating range	1-5
Sparsity	93.6953 %

Testing method: I preferred to use RMSE and MAE as the test metric and "K-Fold CV" as the cross validation iterator. Tests were done using Surprise 2.6.3.1.

Tested Recommender: Inverse Distance Weighted Trust Based Recommender 2.5

Results:

2.6.2.3 Amazon Food Reviews [5]

About the dataset:

	Amazon Food Reviews (threshold = 10)
Number of Users	4276
Number of Products	1140
Rating Range	1-5

Testing method: For this dataset, I wanted to see how efficiently the recommender works on extremely sparse datasets, as a result I prefered to use "Shuffle Split CV" as the cross validation iterator since it is easy to set up the test and train set ratios. For the test metric, RMSE and MAE were used. Tests were done using Surprise 2.6.3.1.

Tested Recommender: Inverse Distance Weighted Trust Based Recommender 2.5

Benchmark:
Number of Splits: 3, Trainset: 0.5, Testset: 0.5, Sparsity: 0.9931

	Trust Based	MSD	SVD	Slope One	KNN	NMF
RMSE	0.7411	0.7898	0.8152	0.7553	0.7679	0.7516
MAE	0.3996	0.4037	0.61073	0.4264	0.4902	0.4832

Number of Splits: 3, Trainset: 0.2, Testset: 0.8, Sparsity: 0.997

	Trust Based	MSD	SVD	Slope One	KNN	NMF
RMSE	0.9766	1.551	1.0252	0.9045	1.0501	0.9659
MAE	0.6321	0.9870	0.8062	0.5104	0.7172	0.6932

Number of Splits: 3, Trainset: 0.1, Testset: 0.9, Sparsity: 0.998

	Trust Based	MSD	SVD	Slope One	KNN	NMF
RMSE	1.275	2.2077	1.1096	1.015	1.2562	1.1271
MAE	0.8909	1.6566	0.8806	0.6178	0.9067	0.8613

2.6.3 Libraries I Used

2.6.3.1 Surprise [4]

Installation :

```
pip install scikit-surprise

```

Sample Usage :

```
from surprise import AlgoBase, PredictionImpossible,
 Dataset
from surprise.model_selection import cross_validate

class Inverse_distance_weighted_tbr(AlgoBase):
...

```

```
reader = reader = Reader(line_format='user item rating
 timestamp', sep=';', rating_scale=(1, 5))

data = Dataset.load_from_file('./dataset.csv', reader=
 reader)
algo = Inverse_distance_weighted_tbr()

cross_validate(algo, data, cv=5, verbose=True)

```

Listing 4: Surprise example

2.6.3.2 Matplotlib

Installation :

```
pip install matplotlib

```

Sample Usage :

```
import matplotlib.pyplot as plt
...
plt.hist(eigentrust_list,
color = 'blue',
edgecolor = 'black',
bins = bins)
plt.title('Distribution of the Eigentrust Values')
plt.xlabel('Range')
plt.ylabel('Number of Eigentrust Values in the range')
plt.show()

```

Listing 5: Matplotlib example

3 Organization

3.1 METU Data Mining Research Group

4 Conclusion

For the future,

1. Not only customers and products records but extra attributes such as product supplier, platform, product category, seller, store, etc. can be made part of the system either adding them as nodes to the Graph of Inverse Distance Recommender or using them as an extra filtering layer as in "Hybrid Recommender Systems".

References

[1] Neo4j. https://neo4j.com/.

[2] Numpy library. https://numpy.org/.

[3] Scipy library. https://www.scipy.org/.

[4] Surprise library. http://surpriselib.com/.

[5] Amazon food reviews. https://snap.stanford.edu/data/web-FineFoods.html, 2020.

[6] Aksoy, K., Odabas, M., Bozdogan, I., and Temur, A. Tacorec. https://senior.ceng.metu.edu.tr/2020/tacorec/, 2020.

[7] Dijkstra, E. W. Dijkstra's algorithm. https://en.wikipedia.org/wiki/Dijkstra%27s_algorithm.

[8] Harper, F. M., and Konstan, J. A. The movielens datasets: History and context. *ACM Trans. Interact. Intell. Syst. 5*, 4 (Dec. 2015).

[9] Kamvar, S. D., Schlosser, M. T., and Garcia-Molina, H. The eigentrust algorithm for reputation management in p2p networks. In *Proceedings of the 12th International Conference on World Wide Web* (New York, NY, USA, 2003), WWW '03, Association for Computing Machinery, p. 640–651.

[10] Li, Y., Liu, J., Ren, J., and Chang, Y. A novel implicit trust recommendation approach for rating prediction. *IEEE Access 8* (2020), 98305–98315.

[11] Massa, P., and Avesani, P. Trust-aware recommender systems. In *Proceedings of the 2007 ACM Conference on Recommender Systems*

(New York, NY, USA, 2007), RecSys '07, Association for Computing Machinery, p. 17–24.

[12] Papagelis, M., Plexousakis, D., and Kutsuras, T. Alleviating the sparsity problem of collaborative filtering using trust inferences. In *Trust Management* (Berlin, Heidelberg, 2005), P. Herrmann, V. Issarny, and S. Shiu, Eds., Springer Berlin Heidelberg, pp. 224–239.