Movie Recommendation System using Cosine Similarity

Amninder S Narota and Rajan Subramaniam
Department of Computer Science, Central Michigan University
Mount Pleasant, MI 48859
(narot1a, rajan1s)@cmich.edu

Abstract

In this paper, we compare commonly used distance measures in vector models, namely, cosine Angle Distance (CAD) queries in high dimensional data spaces. Using theoretical analysis and experimental results, we showed the retrieval results based on CAD when dimension is high. We have applied CAD for rating based similarity retrieval.

1 Introduction

1.1 Applications of Cosine Similarity

Cosine similarity is an efficient method for calculating the similarity between the two independent object or vectors and it is used in the following applications:

1. **Text Mining:** In text mining the cosine similarity is used to find the similarity between the two documents by using the normalized score obtained from each document.
2. **Real-Time Traffic Classification:** Real - Time Traffic Classification covers the critical role of a network administrator to perform network, maintain the quality of service by performing efficient classification process using some statistical analysis process known as weighted cosine similarity.
3. **Semantic Similarity Calculation:** In semantic similarity calculation taxonomy based cosine similarity is used to find whether the two documents are similar or not.
4. **Direction of Motion Calculation in Routing Algorithms (EBGR):** Cosine similarity is one of the functional units used in Edge Node Based Routing which is used to find whether the peer nodes are in the direction of the destination node in a dynamic wireless network known as Vanet.
5. **Text Clustering:** Cosine similarity is used to find the similar word in a document there by reducing the unordered text documents into meaningful related clusters.
6. **Google's Page Rang Algorithm:** Cosine similarity is one of the statistical components used to formulate a web graph by calculating the page score obtained from each node from the search query pattern.

2 Related Work

Yehunda Koren[1] in his article "*The BellKor Solution to the Netflix Grand Prize*" explained about the algorithm Netflix is using for the movie recommendation system. The team of Netflix have more than 40 people hand-tagging TV shows and movies for them. These are typically freelancers who do this to supplement their income. All of their analysts are TV and movie buffs, and many have some experience working in the entertainment industry. They obviously have personal tastes, but their job as an analyst is to be objective.

Another important element in Netflix personalization is awareness. They want members to be aware of how they are adapting to their tastes. Above all, the algorithm that was developed as part of Netflix million dollar prize are blends of a large number of different machine learning techniques. Two of the most notable aspects that emerged from the competition were using matrix factorisation and the so-called "temporal dynamics"[1] to perform collaborative filtering; the full details can be found on the forum page (which has links to papers written by the winning team):

3 Methodology

In this project we are finding the similarities between the different movies by implementing cosine similarity concept on the user reviews obtained from different users. Angular similarity is a similarity metric obtained from the two movies which are considered as two documents by implementing cosine similarity between them. In general the cosine value will be between 0 and 1.If the cosine value of the two vectors are large then the inner product space between the vectors is large which means the two vectors have high degree of dissimilarity. If the cosine value is small then the inner product space between the vectors is small that denotes the two vectors are similar in nature.

Cosine Similarity[2] is a metric used to find the similarity between the two objects in terms of orientation. Here the two objects are considered as vectors in a user defined n dimensional space. The similarity is based only upon the orientation of the two different vectors and it does not consider the magnitude of the two vectors. The similarity level is generally based on the angle derived from the cosine similarity measure which denotes the cosine angular similarity between the two vectors. The two vectors are said to be similar if they have a smaller cosine angle. The smaller angle denotes the inner product space between the two vectors or objects are small which displays that the two vectors are similar. If the cosine angle is large there will be a large inner product space between the two vectors which denotes the two vectors are dissimilar to each other.

$$\begin{aligned}\cos\theta &= \frac{A.B}{||A||\ ||B||} \\ &= \frac{\sum_{i=1}^{n} A_i \times B_i}{\sqrt{\sum_{i=1}^{n}(A_i)^2} \times \sum_{i=1}^{n}(B_i)^2}\end{aligned}$$

3.1 Term Definition

Following are the term definitions we used in this project:

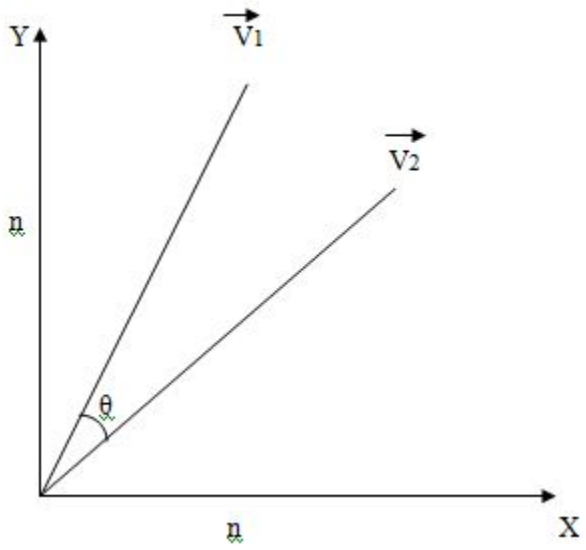

Figure 1: Cosine angular similarity graph

1. **Document:** Collection of item sets. Eg: Movie A, Movie B, Movie C etc.
2. **Item Sets:** Denotes the set of attributes related to the particular document. Eg: Movie ID, User rating, Date information.
3. **Term Frequency:** Measures the number of times the item set occur in a document.
4. **Normalized Term Frequency [NTF]:** To get the values (item sets) in range (normalized) we are dividing each value by the total number of values.
5. **Inverse Document Frequency [IDF]:** It is a potential score which denotes the weight of each item set in a document.

3.2 Formulas Used

1. **IDF[item set]**:

$$\text{IDF[item set]} = 1 + \log_e\left(\frac{\text{Total numbers of users}}{\text{Number of times the item set repeated}}\right) \quad (1)$$

2. **Cosine Similarity:** Finds the similarity between the given item set with the stored item set in a document and generats the angular similarity between the two documents *i.e* Movies.

$$\text{Cosine similarity (item-set, doc)} = \frac{\text{Dot product (item-set, Document)}}{||\text{item-set}|| \times ||\text{ Document}||} \quad (2)$$

3.3 Algorithm

STEP 1: Measuring the Term Frequency for each item set (user ratings) in a document (Movie).

STEP 2: Normalize the Term frequency to get into an appropriate value range.

STEP 3: Generate Inverse Document Frequency (IDF) for each item-set (user ratings) in a document.

STEP 4: Calculating the angular similarity between the two documents

3.3.1 Calculation

For explanation we created sample data for movies, say *Movie A, Movie B* which are as follows:

1. Normalized values for all the user ratings in Movie A are in Table 1

   Document 1: Movie A
   Number of Users: 5
   Rating Limit: 5
   UR: User Ratings
   NUR: Normalized User Ratings

Number of Users	UR1	UR2	UR3	UR4	UR5
UR	3	4	1	2	1
NUR	0.6	0.8	0.2	0.4	0.2

Table 1: Document 1; Movie A

2. Normalized values for all the user ratings in Movie B are in Table 2.

   Document 2: Movie B
   Number of Users: 5
   Rating Limit: 5
   UR: User Ratings
   NUR: Normalized User Ratings

Number of Users	UR1	UR2	UR3	UR4	UR5
UR	3	2.5	3	5	4
NUR	0.6	0.5	0.6	1	0.8

Table 2: Document 2; Movie B

3. Inverse Document Frequency [IDF] is calculated using *Equation 2* of Section 3.2 Item 1 and result is in *Table 3*

4. Normalized values for all the user ratings in Movie C are in Table 5.

   Given Document: Movie C
   Number of Users: 2
   Rating Limit: 5
   UR: User Ratings
   NUR: Normalized User Ratings

Users	IDF Calculation	IDF
U1	$1 + \log_e(\frac{10}{3})$	2.2039
U2	$1 + \log_e(\frac{10}{2})$	2.6094
U3	$1 + \log_e(\frac{10}{2})$	2.6094
U4	$1 + \log_e(\frac{10}{1})$	3.3025
U5	$1 + \log_e(\frac{10}{2})$	2.6094
U6	$1 + \log_e(\frac{10}{3})$	2.2039
U7	$1 + \log_e(\frac{10}{3})$	3.3025
U8	$1 + \log_e(\frac{10}{3})$	2.2039
U9	$1 + \log_e(\frac{10}{1})$	3.3025
U10	$1 + \log_e(\frac{10}{2})$	2.6094

Table 3: IDF Values

Number of Users	UR1	UR2
UR	2.5	1.5
NUR	0.5	0.3

Table 4: Document 2; Movie B

5. Creating (UR × IDF) matrix for the given document - Movie C and result is in Table 5
6. Createing (UR × IDF) matrix by comparing Movie C with Movie A and Movie B. The values are in Table 6
7. Cosine Similarity between two movies can be calculated using the formula in *Equation 2* of Section 3.2 Item 2. Cosine Similarity between Movie C and Movie A is:

$$\begin{aligned}\text{Cosine Similarity(Movie C, Movie A)} &= \frac{(1.6512 \times 1.321) + (0.7828 \times 0.5218)}{\sqrt{(1.6512^2 + 0.7828^2) \times (1.321^2 + 0.5218^2)}} \\ &= 0.99\end{aligned}$$

$$\begin{aligned}\text{Cosine Similarity(Movie C, Movie B)} &= \frac{(1.6512 \times 1.6512) + (0.7828 \times 0)}{\sqrt{(1.6512^2 + 0.7828^2) \times (1.6512^2 + 0)}} \\ &= 0.90\end{aligned}$$

The Cosine Similarity shows the cosine angle between two matrices. The cosine similarity between Movie C and Movie B is smaller when compared to cosine similarity between Movie C and Movie A. Hence with the small cosine value denotes that the inner product space between Movie C and Movie B is small thereby stating that Movie C is more similar to Movie B when compared to Movie A in terms of user ratings.

Number of Users	IDF	UR×IDF
0.5	3.3025	1.6512
0.3	2.6094	0.7828

Table 5: (UR × IDF)

Movie C	Movie A	Movie B
0.5	1.321	1.6512
0.3	0.5218	0

Table 6: (UR × IDF)

3.4 Development

For the development of this project our development environment was Python and to fine Cosine Similarity we used the following libraries:

1. **recsys:** python-recsys is build on top of Divisi2. It is used to calculate cosine similarity calculation
2. **Divisi2:** Divisi is particularly designed for working with knowledge in semantic networks.
3. **SciPy:** It is a computing environment and open source ecosystem of software for the Python programming language used by scientists, analysts and engineers doing scientific computing and technical computing.
4. **Numpy:** NumPy is the fundamental package for scientific computing with Python. It contains among other things:
   - a powerful N-dimensional array object
   - sophisticated (broadcasting) functions
   - tools for integrating C/C++ and Fortran code
   - useful linear algebra, Fourier transform, and random number capabilities

The contents of *movie.dat* are stored in the variable *movieNames* and *ratings.dat* is stored in *ratings*. Every movie name is allotted an unique ID and *movie_dict* is dictionary variable defined to map name to the id of the movie. The similarity matrix is stored in a form of dictionary mapping the movie with 10 top similar movies arranged in the defending order. This variable is then rendered in JSON file. *svd* defined in method *readDat()* as mentioned in Code snippet is used to store the data from the file and stored in the form of matrix.

The values of matrix is stored in *svdlibc* by converting into sparse matrix. If a matrix A is stored in ordinary (dense) format, then the command S = sparse(A) creates a copy of a matrix stored in sparse format. The rating of the movies are separated by *::* in the following structure:

UserID :: MovieID :: Ratings :: TimeStamp

1. **UserID:** Every user who rating the movie is given an unique ID

2. **MovieID:** Every movie is given an unique ID
3. **Ratings:** This is the rating given by the user. The rating value is from 1 through 5.
4. **TimeStamp:** This is the date on which user rated the move title. It is the total seconds elapsed since 1st of January 1970.

```
def readDat():
        global svd
        global svdlibc
        global tree
        global similar_title
        svdlibc = SVDLIBC(ratings)
        svdlibc.to_sparse_matrix(sep='::', \
                format={'col':0, 'row':1, 'value':2, 'ids': int})
        svdlibc.compute(k=100)
        svd = svdlibc.export()
        tree.append(svd.similar(ITEMID1))
        for name in tree:
                for n in name[1:]:
                        similar_title.append(n[0])
                        if svd.similar(n[0]) not in tree:
                                tree.append(svd.similar(n[0]))
                movie_similarity[name[0][0]] = similar_title
                similar_title = []
```

The syntax to find the similarity between two documents is

svd.similarity(item1, item2)
or
svd.similar(item)

Here *svd* is the object which holds the computed vector matrix of similarity. To find all the similar values we used *svd.similar(item)*. All the values are run through loops till it finds the end of movie file and finding all the similar values to the particular movie. All the similar movies are then stored in the form of list and mapped to the movie for which the value was to be found. Later this value is used to generate JSON file.

```
def generateResult(dict):
        for key, value in dict.iteritems():
                str = "%d"%key
                for v in value:
                        str += "::%d"%v
                with open("result.dat", "a") as resultFile:
                        resultFile.write(str+"\n")
                str = ''
```

```
def dat_to_json(dict):
        l = []
        for key, value in dict.iteritems():
                for v in value:
                        l.append(retMovieName(v))
```

movie_dict is dictionary for mapping movie name to the movie id. It opens *movie.dat* file and splits each line by "*::*" and stores in the form if list. The movie id is at 0^{th} index and name is at 1^{st} index.

```
def readMovieNames():
        global movieNames
        movies = open(movieNames, "r")
        lines = movies.readlines()
        for line in lines:
                x = line.split("::")
                movie_dict[x[0]] = x[1]
```

dat_to_json(dict) is method defined to create result in the form if JSON file from the dictionary provided as an argument. It opens "*result.json*" file and writes *movie_name_dict* value into the file.

```
def dat_to_json(dict):
        l = []
        for key, value in dict.iteritems():
                for v in value:
                        l.append(retMovieName(v))
                movie_name_dict[retMovieName(key)] = l
                l = []
        with open("result.json", 'a') as js:
                js.write(json.dumps(movie_name_dict,\
                  ensure_ascii=False))
```

This JSON file is used to render HTML web page linked to http://whispering-sands-8504.herokuapp.com

4 Experimental Results

5 Conclusion

References

[1] Yehuda Koren. 1 the bellkor solution to the netflix grand prize, 2009.

[2] Shiwei Zhu, Junjie Wu, Hui Xiong, and Guoping Xia. Scaling up top-k cosine similarity search. *Data Knowl. Eng.*, 70(1):60–83, 2011.