Beckett Johnson

3/12/2021

Web Information Management

Final Project Report

In the project report below, I go over each of the recommendation algorithm variants we discussed in class and subsequently discuss RMSE results and compare them to the performances of the other algorithms that I evaluated. At the end of the report, I also introduce and explain my own recommendation algorithm and then proceed to evaluate it and compare it to the existing algorithms.

User-Based Cosine Similarity

I implemented user-based collaborative filtering using the cosine similarity method in the file 'user_based_cosine.go'. After testing the algorithm against a segment of the training data, I computed the following RMSE for the algorithm:

```
User-Based Cosine Similarity RMSE: 1.117563
```

User-Based Pearson Correlation

I utilize the generic pearson correlation method in the file 'user_based_pearson.go'. After testing the algorithm against a segment of the training data, I computed the following RMSE for the algorithm:

```
User-Based Pearson Correlation RMSE: 0.822357
```

This result is an extraordinary improvement from the previous RMSE that was yielded from the user-based cosine similarity. From my understanding of this algorithm, I believe there are multiple reasons for this significant improvement; however, the most prominent factors are (1) how the algorithm considers strong inverse correlation just as much as strong correlation and (2) how the algorithm compares user's ratings by the difference from each user's average rating.

User-Based Pearson Correlation with Inverse User Frequency

I then implemented user-based collaborative filtering using pearson correlation adjusted with IUF in the file 'user_based_pearson(with_Inverse_User_Frequency).go'.

$$IUF(j) = \log\frac{m}{m_j}$$

m_j is the number of users that have rated item j
m is the total number of users

To implement IUF, I created a duplicate of the training data that was multiplied by the IUF factor which was calculated according to the formula above. The highlighted section of the code on the next page is where the IUF factor was multiplied into the new adjusted matrix

```go
// Returns an [1000][200]int array with predictions made for all of the last 25 user's existing ratings
func multiplierUsingIUF(ratings [1000][200]int) [1000][200]float64 {
    var adjustedRatings [1000][200]float64

    var noOfUsers int = 200 // Represents m

    for movie := 0; movie < 1000; movie++ {
        var noOfRatingsForMovie int // Represents mj
        for user := 0; user < 200; user++ {
            if ratings[movie][user] != 0 {
                noOfRatingsForMovie++
            }
        }

        for user := 0; user < 200; user++ {
            if ratings[movie][user] != 0 {
                adjustedRatings[movie][user] = float64(ratings[movie][user]) * (math.Log(float64(noOfUsers)) - math.Log(float64(noOfRatingsForMovie)))
            }
        }
    }

    return adjustedRatings
}
```

I then made some adjustments to how the similarity score was calculated; in that, I replaced the original ratings used for calculating the similarity score with the IUF-adjusted ratings as shown in the screenshot below.

```go
// Returns the cosine similarity score between two users, where user1 and user2 are indexes of users in ratings[1000][<user_index>]
func findAdjustedUserPearsonSimilarity(ratings [1000][200]int, adjustedRatings [1000][200]float64, activeUser int, user2 int) float64 {
    var summation1 float64 = 0 // Represents: summation( (Active_User_Movie_Rating - Active_User_Avg_Rating) * (User_2_Movie_Rating - User_2_Avg_Rating) )
    var summation2 float64 = 0 // Represents: summation( squared(Active_User_Movie_Rating - Active_User_Avg_Rating) )
    var summation3 float64 = 0 // Represents: summation( squared(User_2_Movie_Rating - User_2_Avg_Rating) )

    activeUserAvgRating := findUserAverageRatingWithFloats(adjustedRatings, activeUser)
    user2AvgRating := findUserAverageRatingWithFloats(adjustedRatings, user2)

    for movie := 0; movie < 1000; movie++ {
        if (ratings[movie][activeUser] != 0) && (ratings[movie][user2] != 0) {

            var normalizedActiveUserRating float64 = (float64(adjustedRatings[movie][activeUser]) - activeUserAvgRating)
            var normalizedUser2Rating float64 = (float64(adjustedRatings[movie][user2]) - user2AvgRating)

            summation1 += normalizedActiveUserRating * normalizedUser2Rating

            summation2 += normalizedActiveUserRating * normalizedActiveUserRating

            summation3 += normalizedUser2Rating * normalizedUser2Rating
        }
    }

    if (summation3 == 0) {
        return 0
    }

    var similarity float64 = summation1 / (math.Sqrt(summation2) * math.Sqrt(summation3))

    return similarity
}
```

After testing the algorithm against a segment of the training data, I computed the following RMSE for the algorithm:

```
User-Based Pearson Correlation with IUF RMSE: 0.934761
```

I was surprised by this because I expected the RMSE to be lower than that of the raw pearson correlation algorithm. Although I'm struggling to fully understand why this would yield a worse result, I would have to guess that this algorithm might be more effective if the IUF affected the original movie rating less harshly; in other words, I think the IUF might be focusing too much on the less popular movies and almost disregarding the popular movies, thus potentially decreasing accuracy.

User-Based Pearson Correlation with Case Amplification

I implement case amplification in the file 'user_based_pearson(with_Case_Modification).go'. To implement case amplification, I didn't have to change much from the raw pearson correlation algorithm; I only altered how the pearson correlation score between two users was calculated by taking the regular similarity score and then performing the following formula:

$$w'_{a,u} = w_{a,u} \cdot |w_{a,u}|^{\rho-1}$$

This can be seen in the highlighted bit of code within the function I made to calculate a similarity score between two users:

```
// Returns the cosine similarity score between two users, where user1 and user2 are indexes of users in ratings[1000][<user_index>]
func findUserPearsonSimilarity(ratings [1000][200]int, activeUser int, user2 int, activeUserAvgRating float64) float64 {
    var summation1 float64 = 0 // Represents: summation( (Active_User_Movie_Rating - Active_User_Avg_Rating) * (User_2_Movie_Rating - User_2_Avg_Rating) )
    var summation2 float64 = 0 // Represents: summation( squared(Active_User_Movie_Rating - Active_User_Avg_Rating) )
    var summation3 float64 = 0 // Represents: summation( squared(User_2_Movie_Rating - User_2_Avg_Rating) )

    user2AvgRating := findUserAverageRating(ratings, user2)

    for movie := 0; movie < 1000; movie++ {
        if (ratings[movie][activeUser] != 0) && (ratings[movie][user2] != 0) {

            var normalizedActiveUserRating float64 = (float64(ratings[movie][activeUser]) - activeUserAvgRating)
            var normalizedUser2Rating float64 = (float64(ratings[movie][user2]) - user2AvgRating)

            summation1 += normalizedActiveUserRating * normalizedUser2Rating

            summation2 += normalizedActiveUserRating * normalizedActiveUserRating

            summation3 += normalizedUser2Rating * normalizedUser2Rating
        }
    }

    if (summation3 == 0) {
        return 0
    }


    var similarity float64 = summation1 / (math.Sqrt(summation2) * math.Sqrt(summation3))

    p := 3.0

    caseModSimilarity := similarity * math.Abs(math.Pow(similarity, p - 1))

    return caseModSimilarity
}
```

After implementing the modification to the pearson correlation method for case amplification, I used a segment of the training data to test various p-values in order to determine which resulted in the best RSME. The results can be found on the beginning of the next page.

```
p-value: 1.000000   -   RMSE: 0.822357
p-value: 1.100000   -   RMSE: 0.863007
p-value: 1.200000   -   RMSE: 0.859425
p-value: 1.300000   -   RMSE: 0.857690
p-value: 1.400000   -   RMSE: 0.855520
p-value: 1.500000   -   RMSE: 0.850920
p-value: 1.600000   -   RMSE: 0.848668
p-value: 1.700000   -   RMSE: 0.847164
p-value: 1.800000   -   RMSE: 0.844399
p-value: 1.900000   -   RMSE: 0.843140
p-value: 2.000000   -   RMSE: 0.806028
p-value: 2.100000   -   RMSE: 0.841699
p-value: 2.200000   -   RMSE: 0.844041
p-value: 2.300000   -   RMSE: 0.845477
p-value: 2.400000   -   RMSE: 0.843285
```

```
p-value: 2.500000   -   RMSE: 0.840941
p-value: 2.600000   -   RMSE: 0.836381
p-value: 2.700000   -   RMSE: 0.837066
p-value: 2.800000   -   RMSE: 0.835796
p-value: 2.900000   -   RMSE: 0.834600
p-value: 3.000000   -   RMSE: 0.794719
p-value: 3.100000   -   RMSE: 0.831972
p-value: 3.200000   -   RMSE: 0.834013
p-value: 3.300000   -   RMSE: 0.834013
p-value: 3.400000   -   RMSE: 0.837142
p-value: 3.500000   -   RMSE: 0.838589
p-value: 3.600000   -   RMSE: 0.840362
p-value: 3.700000   -   RMSE: 0.841625
p-value: 3.800000   -   RMSE: 0.841625
p-value: 3.900000   -   RMSE: 0.841552
```

I then achieved the following RMSE result which ended up yielding the best (dataset-specific) RMSE of all of the algorithms I implemented:

```
User-Based Pearson Correlation with Case Amplification RSME: 0.794719
```

Although the case amplification with a p-value of 2.0 or 3.0 shows us improvement from the raw user-based pearson correlation algorithm, the testing above seems to indicate that case amplification is generally worse than raw, with the exception of really specific p-values. As a result, I believe that the p-values above are actually not representative of a widely successful algorithm, but instead an algorithm that has been too specifically fit to the dataset that we were testing it with. Hence, it still seems that raw user-based pearson correlation algorithm is the most effective algorithm with regard to use on most datasets. This hypothesis was reaffirmed when I got my best score of approximately

0.822 with raw pearson correlation using the official testing data, while all outcomes using case amplification were higher than 0.83 .

Item-Based Cosine Similarity

I implemented item-based cosine similarity in the file 'item_based_cosine.go'. To implement item-based cosine similarity, I adapted my previously-implemented user-based cosine similarity algorithm by comparing item-similarity scores to find the 'k' most similar *movies* instead of the 'k' most similar users. Then, the 'k' most similar movies are used to generate a predicted rating. This resulted in me changing functions *makeSinglePrediction*[line 43] and *findMovieCosineSimilarity*[line 88] by replacing variables which referenced user2 to variables that referenced a second movie instead. The actual formulas used were those found on the slides:

$$p_{a,i} = \frac{\sum_{j=1}^{k} w_{i,j} r_{a,j}}{\sum_{u=1}^{k} w_{i,j}}$$

$$sim\ (i_1, i_2) = \cos\ \vartheta_{i_1, i_2} = \frac{\sum_{j=1}^{m} r_{u_j, i_1} \times r_{u_j, i_2}}{\sqrt{\sum_{j=1}^{m} r_{u_j, i_1}^2} \times \sqrt{\sum_{i=1}^{m} r_{u_j, i_1}^2}}$$

After completing the implementation and testing the algorithm against a portion of the training data, I received the following RMSE score:

```
Item-Based Cosine Similarity RMSE: 1.204537
```

This score was the worst RMSE result of all of the algorithms implemented. I believe the reason that it led to poor results was because, semantically, item-based collaborative filtering makes less sense; in that movies don't rate users, users rate movies. Hence, I think there is sufficient reasoning to believe that these results are accurate and that user-based collaborative filtering yields better results.

Implementing my own algorithm

The algorithm I implemented by my own design was inspired in class. As the lecture slides were attempting to subtly push us towards the idea of adjusting movie ratings based on movie popularity(IUF), my mind actually started going in the direction of adjusting movie ratings based off of how polarized the ratings for a movie is. In other words, my algorithm is implemented in a similar manner to IUF, but instead of emphasizing rarer movies, my algorithm emphasizes movies that have larger standard deviations in their ratings.

I implemented this in the same way that I implemented IUF, but instead of multiplying the original ratings by an IUF factor (in order to compute user similarity scores), I multiplied the original ratings by a 'polarization factor,' which represents how much bigger(or smaller) the standard deviation of a movie's ratings are than the average movie rating standard deviation. The formula I used to compute the 'polarization factor' for each movie was:

$$Polarization\ Factor(movie\ j) = log(\frac{Standard\ Deviation\ of\ Ratings\ for\ movie\ j}{Average\ Standard\ Deviation\ of\ All\ Movie\ Ratings})$$

As detailed in how I implemented IUF, this polarization factor for each movie was multiplied by each movie's original ratings; thus, creating a new dataset that was used to determine user similarity scores. This process of creating this new adjusted dataset is shown in the function *emphasizeControversialMovies*[line 30].

As seen below, the adjusted dataset of ratings was used in computing the similarity score between users:

```
156  // Returns the cosine similarity score between two users, where user1 and user2 are indexes of users in ratings[1000][<user_index>]
157  func findAdjustedUserPearsonSimilarity(ratings [1000][200]int, adjustedRatings [1000][200]float64, activeUser int, user2 int) float64 {
158      var summation1 float64 = 0 // Represents: summation( (Active_User_Movie_Rating - Active_User_Avg_Rating) * (User_2_Movie_Rating - User_2_Avg_Rating) )
159      var summation2 float64 = 0 // Represents: summation( squared(Active_User_Movie_Rating - Active_User_Avg_Rating) )
160      var summation3 float64 = 0 // Represents: summation( squared(User_2_Movie_Rating - User_2_Avg_Rating) )
161
162      activeUserAvgRating := findUserAverageRatingWithFloats(adjustedRatings, activeUser)
163      user2AvgRating := findUserAverageRatingWithFloats(adjustedRatings, user2)
164
165      for movie := 0; movie < 1000; movie++ {
166          if (ratings[movie][activeUser] != 0) && (ratings[movie][user2] != 0) {
167
168              var normalizedActiveUserRating float64 = (float64(adjustedRatings[movie][activeUser]) - activeUserAvgRating)
169              var normalizedUser2Rating float64 = (float64(adjustedRatings[movie][user2]) - user2AvgRating)
170
171              summation1 += normalizedActiveUserRating * normalizedUser2Rating
172
173              summation2 += normalizedActiveUserRating * normalizedActiveUserRating
174
175              summation3 += normalizedUser2Rating * normalizedUser2Rating
176          }
177      }
178
179      if (summation3 == 0) {
180          return 0
181      }
182
183      var similarity float64 = summation1 / (math.Sqrt(summation2) * math.Sqrt(summation3))
184
185      return similarity
186  }
```

After implementation, I tested the algorithm and found that it yielded a RMSE of:

```
User-Based Pearson Correlation with Polarizing Movies Emphasized RMSE: 1.000420
```

While this still doesn't come close to beating the RMSE of the user-based pearson correlation with case modification or the raw user-based pearson correlation algorithm, it is quite interesting to me that it has a significantly better outcome than IUF. I can certainly see why this makes sense because, in some intuitive sense, considering the diversity of a movie's ratings seems like it would be more indicative of which movies a user has seen. For example, if a movie is really rare, but everyone rates it a 3, it still doesn't tell us much about the individual we are trying to understand. On the other hand, even if a movie is really popular, but the ratings are very polarized, it gives us a more substantial understanding of the user. Hence, this is how I might justify the better performance of this algorithm in comparison to IUF.