POLITECNICO DI MILANO

Recommender Systems competition final report

RS Course 2016/2017 (Prof. Cremonesi)

AUTHORS
Leonardo Turchi - 853738
Sara Pisani - 854223

ABSTRACT
The aim of this report is to clarify and retrace all the steps that our group has made to implement and to improve a recommender system algorithm. Our task was to find a way to recommend job posts to persons with different fields of study, competencies and nationalities, so as satisfy their preferences, by creating an algorithm as precise as possible. In this report you will find some parts of code: we were allowed to choose the programming language, and we implemented this algorithm in C#. We had to handle incomplete datasets, because normally there aren't information regarding every detail about a person's life. We had to make inferences about a person's tastes, because there were not explicit votes about jobs and we gained access only to person's behavior regarding clicks on these announcements. Therefore, in this document, we will explain how we have managed this kind of situations, which strategies we have adopted and how we had being able to meet deadlines.

TOP POPULAR

Initially we had a scant knowledge about the functioning of a recommender system, therefore we started with a simple approach: during the first week from the beginning of the competition, we tried to develop a basic algorithm in order to take confi dence with this world: the "Top popular" algorithm. This solution recommends the 5 most popular items in the data set to the user, but the submission's score provided by the Kaggle's evaluator was even worse than the "Random" one.
At this point we felt the need to develope a set of functions that let us to check the functioning of the algorithms without use the kaggle submissions, for this reason we created a "MAP" routine to get even only an idea of the good working of the new functions that we were going to write.

USER SIMILARITY

(compute all similarities for users)
This similarity was created by computing a cosine similarity between the vectors of each user.
This was the first experiment in the mid of October and with this algorithm we were able to increase the previous top popular score of 5-6 positions.

$$sim(a,b) = \frac{\vec{a} \cdot \vec{b}}{|\vec{a}| * |\vec{b}|} \quad (1)$$

CONTENT BASED

(compute all similarities for items)
The similarity was created by using different techniques, basing in the value that we had to check. For the cells that contains a list of values (like titles or tags), we compute the Jaccard similarity. For the cells that contains an int, we compute the similarity by checking the gap and how much this gap is from the other cell. For the cells of the LatLon positions, we used a function that return the Haversine calculation for distance between two coordinates in a sphere. The idea was to increase the precision of the predictions, by suggesting a job not so much far to the similar one. But after a review, we though that a generic user could decide to travel in order to find a great job. So due to the specific kind of datasets, we think that is not the best way to predict jobs with high precision.

In this example there is the Jaccard similarity computation fo the cells of titles and tags (of item profile):

```
//execute JACCARD on these values (int)
double intersect = row1CList.Intersect(row2CList).Count();
double union = row1CList.Union(row2CList).Count();

//check if no one in common
if (intersect == 0)
    return 0;

//compute the similarity
return intersect / union;
```

With this technique we were not able to increase constantly our ranking in the competition, and quickly we needed a new kind of approach.

COLLABORATIVE FILTERING

The first approach to the CF world was to create a similarity basing on the jobs titles and on the jobs tags.

Collaborative filtering based on job titles and on job tags.
In this case we computed a similarity between items (item-item) based only on the lists of tag in the job attributes, or only on the lists of jobs titles (so in our minds a job with a similar title to another, could be predictable to a similar user). After the creation of this lists, we searched for the interactions of an user, and we recommended to this user the first 5 items with similar title or tag. But due to the sparsity of the dataset, and due to the encoding of the strings in the titles, the predictions were not so accurate. This let us to raise in early-November, but in the following month we dropped out this approach. Another problem was to recommend items to users that had no interactions, we realized that a secondary CB approach could has been the solution.

Other approaches: ranking of items ratings.
In this case we tried to assign to every interaction, a rating (1 to 10) based on the number of clicks, the freshness, and the type of click. So we tried to 'convert' an 'implicit dataset' to an 'explicit' one. That was only an ongoing attempt.

$$score(u,i) = \sum_{inter-item} \frac{interaction(u,i) \cdot weight(i)}{popularity(i)} \tag{2}$$

Maybe because the sparsity of the data, or maybe because of a wrong implementation of the ranking algorithm, this approach was not so much useful.

FINAL ALGORITHM (HYBRID CF+CB DICT)
This algorithm is an hybrid one, makes use of an user-user similarity, of an item-item similarity, and of a collaborative filtering to increase the precision for the output predictions. The name 'DICT' comes from the use of 'dictionaries' that in C# have a time complexity much better than arrays.

The algorithm executes in a sequence the following sub routines: 1 - Execute CF and 2 - Execute CB (see 3. and 4. for the details).

We build a function to create a matrix 'User-Items' and a matrix 'Item-Users', like the following:

$$\begin{bmatrix} u_1 & i_1 & i_2 & \dots & i_m \\ u_2 & i_1 & i_2 & \dots & i_m \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ u_n & i_1 & i_2 & \dots & i_m \end{bmatrix} \wedge \begin{bmatrix} i_1 & u_1 & u_2 & \dots & u_n \\ i_2 & u_1 & u_2 & \dots & u_n \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ i_m & u_1 & u_2 & \dots & u_n \end{bmatrix}$$

These matrices (that are really dictionaries because by a 'key' interrogation we can get the value in O(1) in C#, so much better than lists or arrays that scan in O(n)), were built by enumerating for each user, all items that interacted, and for each item, all users that clicked on it. Basing on these matrices, we were able to build an iterating algorithm that for each clicked item (by an user), scan the other dictionary in search of 'predictable items', as follow on point 3.

Another approach was the TF and IDF [TermFrequency/InverseDocumentFrequency] computation, where tf is computed as the feature's count divided by the number n of unique features in the user profile:

$$tf(f) = \frac{count(f)}{n} \tag{3}$$

The inverse document frequency is calculated as the logarithm of the popularity of the most popular feature in the data set divided by the feature's popularity (popularity means recurrence).

$$idf(f) = \log \frac{maxPopularity}{popularity(f)} \tag{4}$$

With this algorithm we improved consistently the score in Kaggle, because this made possible to have more precision in the execution of the CB algorithm.

3 - compute CFCB Hybrid User User similarity
The similarity was created enumerating all users, and foreach item that the user interacted in past (from the User-Items matrix) we disposed the normalized coefficient of the similarity between them (where i is an interaction value), by using:

$$sim(a,b) = \frac{\sum_{i \in Itms} i_{(i,a)} \cdot i_{(i,b)}}{\sqrt{(|items_{(i,a)}|)} \cdot \sqrt{(|items_{(i,b)}|)} + Shrink} \tag{5}$$

and then the prediction was created recommending an item with the highest predicted score:

$$pred(u,i) = \frac{\sum_{c \in relatedItms} sim(i,c) \cdot idf(i)}{\sum_{c \in relatedItms} sim(i,c) + Shrink} \tag{6}$$

Numerator creation (based on interaction between the first user and every similar user to him):

```
//if the sim_user has interacted
num = interaction_type * interaction_type_of_sim_user;

//storing coefficients
if (CF_uu_sim_dict_num[user].ContainsKey(sim_user))
    CF_uu_sim_dict_num[user][sim_user] += num;
else
    //add to similarity dictionary
    CF_uu_sim_dict_num[user].Add(sim_user, num);
```

then, creating normalization coefficient (based on the count of interacted items by each user)

```
//for each user in the dictionary
foreach (var u in RManager.user_items_dictionary)
    //increase norm
    CF_uu_sim_dict_norm[u.Key] = Math.Sqrt(u.Value.Count());
```

then, computing prediction for every user to sim-user

```
//evaluate prediction of that sim_user for that user
double pred =
    CF_uu_sim_dict_num[user][sim_user] /
    ((CF_uu_sim_dict_norm[user]
      *
      CF_uu_sim_dict_norm[sim_user])
      + SIM_SHRINK_UB);
```

The Hybrid section in this case, let the CF and the CB algorithm to collaborate.
The first part of the CB algorithm is very similar to the CF one, but in this case, the matrices are not scrolled by using the user-items dictionary, instead we iterate by using the user-attributes dictionary. This matrix contains for each user, a list of string 'encoded' values, to let us to iterate over them to find correlations. All this values are encoded by a simple conversion function, that associate to a different role a different id: for example if a user has a 'career level' of '2', the resulting encoded string is 'UCL2'. By creating this matrix, we were able to perform a TF and a IDF computation, to increase the overall precision of the predictions. The similarities from CF and CB were so combined by this loop:

```
//similarity combination
foreach(var user in CF_uu_sim_dict.Keys.ToList())
    foreach(var user2 in CF_uu_sim_dict[user].Keys.ToList())
        if(CB_sim_dict[user].ContainsKey(user2))
            CF_uu_sim_dict[user][user2] +=
             (CB_sim_dict[user][user2] * CFCB_HYBRID_UB);
```

Then we normalized the entire prediction matrix, and for each target user, we compute the normalization of its prediction by searching the max value of whole predictions matrix.

4 - compute CFCB Hybrid Item Item similarity.
The algorithm is very similar to the 3 (user user), but in this case, the similarity is not computed between users, but between items and their attributes. So even in this case we generated a item attributes matrix, and for each attribute of an item (of a job) we encoded the value in a similar way to the user one (e.g. a 'discipline id' of '8' for a job would be encoded as 'IDI8').

5 - Merge algorithm 3 (user-user predictions) and algorithm 4 (item-item predictions) by assigning weights. Finally merge with algorithm 2 (CB) by assigning weights.

This last step is really necessary, because the algorithm 2 has also the similarity coefficients for the users with no interactions (so not predictable by a simple CF).
By merging we add these predictions to the final output, and this trick avoid us to insert the TOP popular predictions for these target-users that had never interacted with any job post.

$$\begin{bmatrix} tgt_1 & pred_1 & pred_2 & \dots & pred_5 \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ tgt_n & pred_1 & pred_2 & \dots & pred_5 \end{bmatrix}$$

The resulting matrix contains the normalized predictions for each user. So in the very last step we had only to get the top 5 predictions for every target-user.

BONUS
How we increased the submission score in the final days: we added a BONUS on these items (interacted by a user) that were clicked in the recent past. We based our bonus algorithm over the unix timestamp in the interactions dataset. We found the last timestamp of the dataset (when the datasets snapshot was been created) and from that, we found the 'limit timestamps' for '5 days in the past', for '7 days in the past' (for the given dataset the timestamp of the last 7 days was 1446336000).
In this way we grant a bonus to the interactions more 'fresh', and so with more chance to be relevant.

CONCLUSIONS
After an high number of trials, we can conclude that the best way to obtain a good score on the competition leaderboard was to implement a sort of hybrid recommender system, that uses a collaborative filtering (CF) strategy each time we have the possibility to manage an user that clicked on some other jobs, and that uses a content based filtering (CB) strategy otherwise.
We though that for our case, the Collaborative Filtering is more precise than a Content Based filtering, because of the sparsity of the items dataset, but CB is necessary to avoid the absence of a prediction in a relevant number of cases (when the CF could not be used because the target user has no interactions).
We developed our algorithm in C# using visual studio, because C# is an high level language that we also know and could be adapted to the data analysis (thanks to lambda functions, parallel for and serialization of local vars).
We want to specify that our choice is based on our programming experience, but this algorithm can be fully re-produced in any programming language following the main concepts explained before. One of the plus of this algorithm is that it avoid overfitting: in the last deadlines our score in the private leaderboard corresponds to about the score on the public one, and in the last submission had been even higher.

REFERENCES

- Microsoft Visual Studio and Microsoft C# reference https://msdn.microsoft.com/en-us/library/618ayhy6.aspx
- Building machine learning systems with python by Packt Publishing
- Personalized Ranking from Implicit Feedback
- Collaborative Filtering for Implicit Feedback Datasets
- Machine Learning using C# Free Ebook by Syncfusion