CSCI567 Project: Byte Cup 2016
MLClass_Recursion

Muhammad Rizwan Saeed saeedm@usc.edu
Muhammad Junaid Hundekar hundekar@usc.edu
Ankit Kothari ankothar@usc.edu

1 Objective

Toutiao Q&A is a mobile social platform, which has 530 million users and 300 thousand professional freelance writers. The interactions among users is in the format of Q&A. The objective is to match information with the right people, finding the best respondent to the questions, and the best readers to the answers. The data for the challenge is provided by them to conduct analysis on specific questions. Each data record includes expert tags, question data and question distribution data. Given certain questions, participants need to forecast which experts are more likely to answer which questions. Specifically, given each question and each expert, participants need to calculate the probability of that expert answering the question.

2 Dataset Description

The data set provided in the competition contains three types of information:

- `user_info.txt`: Expert tag data, contains IDs of all expert users, their interest tags, and processed profile descriptions.
- `question_info.txt`: Question data, contains IDs of all questions, processed question descriptions, question categories, total number of answers, total number of top quality answers, total number of upvotes.
- `invited_info_train.txt`: Question distribution data, 290,000 records of question push notification, each contains a question ID, an expert ID and whether or not the expert answered the question. For our offline evaluation, we divided this into training (80%) and validation (20%) sets, more on which in Section 4.1.
- `validate_nolabel.txt` and `test_nolabel.txt`: Two files for submitting results for online evaluation on validation and test sets.

3 Our Approach

3.1 Capturing Intuition through Multiple Naive Models

The basic question that needs to be determined is that *how likely a given user is to answer a given question based on certain features?*. The *features* are based on our intuition derived from the data set. For example, one such intuition can be that if the user has answered a similar question to a given question in the past then he is likely to answer the given question as well. Since, this single intuition on its own may not be able to provide acceptable results, hence we consider it a *naive model*. Each one of such naive models generates a score (or feature), and the outputs of multiple such naive models are combined in the final step to generate ranking of users for a given question in the data sets.

Based on our understanding of the data and the domain, we created the following hypotheses that for a given question and user:

1. If a user *similar* to given user have answered this question, then the given user is also likely to answer it.
2. If the given user has answered a similar question in the past, then he is also likely to answer this question.

3.2 Notion of Similarity

Our hypotheses is based on computing similarity among different users and among different questions. The data set provides multiple features of both questions and users that we can use to compute similarity. In our approach we computed similarity using the following features:

Between users:

- User tags
- User profiles (words)
- User profiles (characters)
- History of questions answered

Between questions:

- Question tags
- Question description (words)
- Question description (characters)
- Question features (number of answers, quality answers, upvotes etc.)
- Users who answered the questions being compared

Assuming, we have U users in `user_info.txt` and Q questions in `question_info.txt`, we compute multiple matrices of size $U \times U$ and $Q \times Q$ for similarity among users and among questions respectively. Moreover, based on the training data set `invited_info_train.txt`, we also populate a user-question history matrix W of dimensions $U \times Q$, such that

$$W_{ij} = \begin{cases} 1 & \text{if in the training set, } i^{th} \text{ user answered } j^{th} \text{ question} \\ -1 & \text{if in the training set, } i^{th} \text{ user ignored } j^{th} \text{ question} \\ 0 & \text{if no entry exists in the training set for } i^{th} \text{ user and } j^{th} \text{ question} \end{cases}$$

By using the similarity matrices and W, scores for the likelihood of a given user u to answer a given question q were computed. Assume that u is the index of the given user in the users data set and q is the index of the given question in questions data set and let V be one such matrix that is computed based on the cosine similarity between two users based on their tags. The dimensions of V are $U \times U$. The likelihood (score) for user u and question q can be computed as follows:

$$score < u, q >= \frac{1}{n} \sum_{i \neq u}^{U} V_{ui} \, W_{iq} \tag{1}$$

where n is the number of users (excluding u, by temporarily setting W_{uq} equal to 0) for which the entry in q^{th} column of W is non-zero. Then, the summation part of equation 1 is simply dot product of the u^{th} row of V (which is the similarity score of all users with given user u) and qth column of W (which is the list of users who answered or ignored question q). The dot product is later normalized to be in the range of 0 to 1.

3.3 Other Features

By inspection of data, it was evident that users answered even those questions the tags of which did not match users' own tags. Hence, in order to model the preference of certain question tags for each user, we computed probability of each user answering a question given a question tag. This is another naive model that represents how likely a user is to answer a question belonging to a given tag, based on his history. We computed another version of this likelihood score which we term as *weighted probability*. This can be explained by an example. Suppose there are two users u_1 and u_2 in the data set. u_1 was given a single question, which he answered. u_2 was given n ($n \geq 2$) questions, all of which were answered by him. Hence both have a likelihood of 100% in answering questions posed to them. However, we are more confident in making that conclusion for u_2, since we have observed more data. Therefore, we want to give higher weight to u_2 in terms of likelihood of answering future questions. This is done by multiplying the probability of answering the question for each user by the number of questions he has answered, which gives us *weighted probability*. We calculated weighted probability for each user for answering a question for each question tag and also for answering a question irrespective of the question tag. Table 1 provides the complete list of computed features. Each feature has a value between 0 and 1 and is an output of a naive model that captures our intuition about the data.

Features	**Description**
1	Similar user answered same question based on tag similarity
2	Similar user answered same question based on word similarity of profile description
3	Similar user answered same question based on char similarity of profile description
4	Same user answered similar question based on combination of upvotes, all answers, quality answers and question tag
5	Same user answered similar question based on words similarity of question description
6	Same user answered similar question based on char similarity of question description
7	Similar used answered same question based on question history similarity
8	Probability of answering question based on this tag for this user
9	Weighted probability of answering question based on this tag for this user
10	Question tag is in user tag
11	Same user answered similar question based on users who answered that question
12	Same user answered similar question based on upvotes
13	Same user answered similar question based on number of answers
14	Same user answered similar question based on number of quality answers
15	Percentage of quality answers
16	Probability of answering any question
17	Weighted probability of answering any question
18	Probability of this question getting answered
19	Weighted probability of this question getting answered
20	Same user answered similar question based on combination of upvotes, all answers and quality answers

Table 1: Intuition based naive models

3.4 RankSVM

As the final step in our approach, we use Ranking SVM (or RankSVM) [1] to generate ranked list of users for a each given question in training/test dataset. RankSVM is a pairwise method for designing ranking models and finding ordered lists of items. The algorithm uses pairwise difference vectors to produce ranked list of items [2].

4 Implementation

Most of the modules in our approach are implemented using Python. For the final step of computing ranked lists, we use the MATLAB implementation of Ranking SVM provided by Microsoft Research.[1]

4.1 Dataset Partitioning and Processing

The Byte Cup website provides the users a mechanism for evaluating the models online, but the number of attempts are limited to 3 per day. To have more flexibility for evaluating our models,

[1]http://research.microsoft.com/en-us/um/beijing/projects/letor/Baselines/RankSVM-Primal.html

we divided the given training data `invited_info_train.txt` into two parts. The split was done according to ratio of $80 - 20\%$ with 20% becoming our validation set. We used the validation set to evaulate our models offline. This approach has two benefits: i) prevented us from designing models specifically for entire training data (overfitting) and ii) gave us more flexibility in evaluating our models in addition to the online evaluation tool. Another thing to note is that out of more than $250,000$ records in the original training data, only 11% records are positive examples i.e., where a given user answered the given question, marked by 1. Hence the split of training data is done in a way so that the positive examples also get distributed by the ratio of $80 - 20\%$. Moreover, for the same user and question pair, multiple entries existed in the training set. We removed all such duplicates keeping only a single entry for each user and question pair. If there were multiple negative and one positive example for a given pair, then all negative examples were removed. This is illustrated through the synthetic example in Table 2, which shows that this user ignored the question for the initial few times when he was notified but eventually answered it. Since, we don't incorporate temporal information or use repeated notifications in our model, we ignore the duplicates, keeping only the positive example.

qid	**uid**	**answered**
56e9971a2044110219cc66a3e8f31819	f4063a9d8b51e6ec671137aae908fa49	0
56e9971a2044110219cc66a3e8f31819	f4063a9d8b51e6ec671137aae908fa49	0
56e9971a2044110219cc66a3e8f31819	f4063a9d8b51e6ec671137aae908fa49	0
56e9971a2044110219cc66a3e8f31819	f4063a9d8b51e6ec671137aae908fa49	1

Table 2: Example of duplicate records in training data set

4.2 Python Modules

The purpose of this section is to provide the details about each of the different programming modules (or files) and also the order in which these programs need to be executed to generate the intermediate files. Please note that two subdirectories should be created in the directory where all python files are placed. The names of those subdirectories should be:

- original_data (containing all files downloaded from byte cup website)
- modified_data (processed and partitioned files)

Table 3 gives the list of python files and their description, inputs and outputs. The files should be executed in the order given in the table to generate final output. A script `CSCI567_bytecup_python.sh` is included in the project deliverable which can be run by issuing following command on a terminal in Ubuntu:

```
bash CSCI567_bytecup_python.sh
```

This will take a while to execute every file listed in Table 3, generating intermediate matrices. The final output should be four files: `trainCV.txt, valiCV.txt, testCV.txt` and `submitCV.txt`. The format of the files is commonly used for learning to rank algorithms.

```
1 qid:1 1:1 2:1 3:0 4:0.2 5:0 # u2
0 qid:1 1:0 2:0 3:1 4:0.1 5:1 # u10
0 qid:1 1:0 2:1 3:0 4:0.4 5:0 # u4
1 qid:1 1:0 2:0 3:1 4:0.3 5:0 # u3
0 qid:2 1:0 2:0 3:1 4:0.2 5:0 # u9
1 qid:2 1:1 2:0 3:1 4:0.4 5:0 # u15
1 qid:2 1:0 2:0 3:1 4:0.1 5:0 # u1
0 qid:2 1:0 2:0 3:1 4:0.2 5:0 # u6
```

Same questions are grouped together in consecutive rows, along with features of users who answered them (likelihood scores in our case). The first entry is 0 or 1 depending on if the user answered the question or not. These files are inputs to the RankSVM Matlab module.

4.3 RankSVM Module

The Matlab implementation of RankSVM is divided into following files:

Sr.	File Name	Description
1	divide_dataset_info_full.py	Takes the original files from bytecup website and create joined versions used by subsequent modules.
2	divide_dataset_info_train.py	Creates 80-20% split of the `invited_info_train.txt`.
3	createUxQmatrices_CV.py	Same as above, but done for 80% training data
4	createUxTagsProbability_minus_CV.py	Computes the probability of a user answering a question based on question tag? For each user and each tag computes: P(Answered = Yes, | u,qtag)/P(Ans = yes | u,qtag) + P(Ans = no |u,qtag). Also computes the probability of each user of answering any question i.e., (Answered = Yes, | u)/P(Ans = yes | u) + P(Ans = no |u). Done for 80% training data.
5	uni_u2u_tagsim.py	Compute similarity between two users based on user tags
6	uni_q2q_tagsim.py	Compute similarity between two questions based on question tags
7	uni_q2q_featuressim.py	Compute similarity between two questions based on features (number of answers, quality answers, upvotes and question tags)
8	uni_q2q_featuressim2.py	Compute similarity between two questions based on features (number of answers, quality answers, upvotes)
9	uni_q2q_wordsim.py	Compute similarity between two questions based on words and characters of question description
10	uni_u2u_wordssim.py	Compute similarity between two users based on words and characters of user profiles
11	uni_u2u_qhistory_sim_CV.py	Compute similarity between two users based on the questions they have answered and ignored
12	compute_score_CV_submission.py	Compute scores for individual models based on the matrices computed above (from 80% training data), for question and user pairs for online validation file.
13	compute_score_CV_testing.py	Same as above, but done for entire testing file.
14	compute_score_CV_training.py	Same as above, but done for (80%) training data.
15	compute_score_CV_validation.py	Same as above, but done for remaining (20%) training data.
16	data_files_transform_CV.py	Convert the file into format used by Rank-SVM and other learning to rank software. Currently, the selection of naive models to be used in final step is done manually by setting the `features` variable in this file.

Table 3: List of python modules and their description

- CSCI567_bytecup_main.m (main file)
- compute_ndcg.m
- generate_constraints.m
- ranksvm.m
- read_letor.m
- write_out.m

RankSVM module can be executed by running `CSCI567_bytecup_main.m`, which computes the NDCG evaluation metric based on the byte cup evaluation criteria and displays results. The RankSVM module runs the evaluation for different values of the parameter *C* and selects the best result. The evaluation formula is:

NDCG@5 * 0.5 + NDCG@10 * 0.5

4.4 Files for Uploading to Byte Cup Website

Based on the best model, two files `temp.output` and `final.output` are also generated by RankSVM module which can be appended to `temp.csv` and `final.csv` (generated by Python modules) respectively. `temp.csv` is meant for submission of results on validation data whereas `final.csv` is meant for submission of results on test data.

5 Results

We used multiple permutations of the features to select the set that gave the highest offline validation accuracy (based on 20% of training data). The online validation and testing files were generated by RankSVM module which were than submitted to the byte cup website. Since, it is hard to tell which one of the test files scored the highest, we can only list some of the scores that we got for our online validation files. The selected features and scores from online validation are listed in Table 4.

Features	**Online validation score**
1-10	0.478304451155421
4,5,7,8,9	0.481747337503593
2,4,5,7,9,11	0.485681991898277
2,4,5,7,8,9,11,12,13,14	0.498012402913019
All	0.478304451155421

Table 4: Scores from online validation

6 Alternative Approaches

TPOT[2] is a Python tool that automatically creates and optimizes machine learning pipelines using genetic programming. TPOT is built on top of scikit-learn, so all of the code it generates is from the same library. We tried to use it with complete training set with all the features but using this data set the algorithm never ran to completion. We also worked with GraphLab[3] in the beginning but had to leave that route because it didn't qualify as an acceptable tool according to the rules of the challenge.

References

[1] R. Herbrich, T. Graepel, and K. Obermayer. Large Margin Rank Boundaries for Ordinal Regression. Advances in Large Margin Classifiers, 115-132, Liu Press, 2000.

[2] T. Joachims. Optimizing Search Engines Using Clickthrough Data, Proceedings of the ACM Conference on Knowledge Discovery and Data Mining (KDD), ACM, 2002.

[2]https://rhiever.github.io/tpot/

[3]https://turi.com/