# ORAC Recommendation System

**Junhua Chen 14-7-2021**

**1 Summary:**

The aim is to develop a model to recommend problems to users of the ORAC informatics training site. In the current environment where data privacy is emphasised, we aim to devise an accurate recommender subject to an unavailability of user information. We choose models with relatively low computational requirements as well as reliance on advanced analysis of the problems being recommended. Thus, the models produced can also be generalised to other unary (there is only 1 type of interaction between user and item) recommendation problems too.

**2.1 Available information**

- $N$ anonymised users and $P$ problems and a binary matrix $R$ with $R_{i,j} = 1$ if and only if user i have solved problem j and 0 otherwise.

- Basic information about the problems, namely its ID, which is in chronological order, and its problem class (see below). Problems not in one of these categories OR are sliding scale OR are not public are ignored.

- A list of problems each user has solved since September 2020, up to 30 problems per person. We define these as "recent solves"

| Class | Class name |
|---|---|
| 1 | Starter problem |
| 2 | AIO |
| 3 | AIC |
| 4 | AIIO |
| 5 | FARIO |

**2.2 Task and benchmark**

We aim to recommend problems to the user that he/she would have in natural circumstances been likely to solve next. We use metrics inspired by the clickthrough rate. For each person we collect is total solves. We then take a random set S of their recent solves where

$$|S| = \min\left(20, \left\lceil \frac{\#Total\ solves}{2} \right\rceil, \#Recent\ solves\right)$$

For convenience, if we ask our model to predict the top k most likely next solves for the ith person, denoted as the sets $T_i(k)$. We define 2 metrics of accuracy:

*Top-k-Accuracy: We* define this metric as the value

$$\sum_{i=1}^{N} \frac{|S_i \cap T_i(k)|}{|S_i|}$$

*Top-N+k-Accuracy:* We define this metric as the value

$$\sum_{i=1}^{N} \frac{|S_i \cap T_i(|S_i|+k)|}{|S_i|}$$

We consider 3 Values of Top-k-Accuracy; 5,10 and 20 and choose k=5 for Top-N+k.

### 2.3 Collection of Data

Data collection proceeded in 3 steps. The libraries BeautifulSoup was used to process HTML files whereas asyncio and aiohttp allowed asynchronous access of the numerous web pages required.

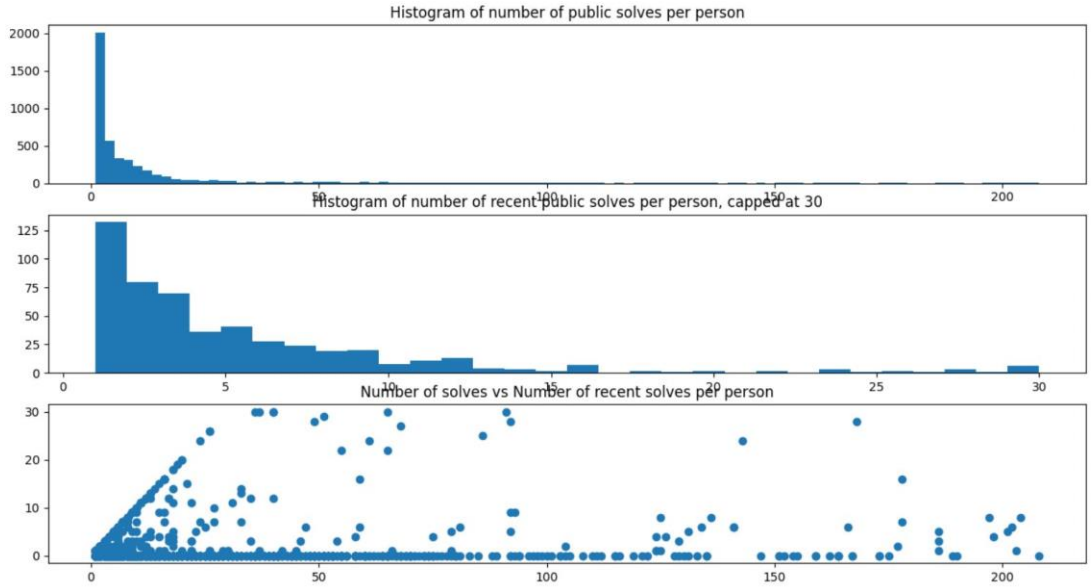
1) The training hub home was scraped to read out a list of problems and the problem classes. Regrettably aiohttp malfunctioned on this page, cause unknown, and the page source had to be downloaded locally and read instead.
2) The hall of fames for each problem were scraped to retrieve a list of people who solved each problem. Problems are stored by problem ID.
3) Lorikeet, a backdoor admin interface were used to get a list of recent solves. The alternative would have been to use the Wayback machine if such interfaces were unavailable.

### 2.4 Processing of Data

Python was used to process the Data. Solver data was divided into 3 sets, a cross validation set and training set each containing half the people with recent solves, selected by random, and the Train set, containing every person and their public solves, with solves selected for inclusion in cv and test sets redacted. A summary of data is below:

| | |
|---|---|
| Number of People with at least 1 solve: | 4377 |
| Number of problems: | 208 |
| Number of total solutions: | 48323 |
| Number of people with at least 1 recent solve: | 522 |
| Number of recent solves (cap at 30/person): | 2720 |
| Cross-Validation Set: | 247 People with 892 Solves |
| Test set: | 275 cases with 947 Solves |

Evidently, our dataset is quite small. Further, it was pointed out that the data is relatively low rank, with strong correlations between solving various problems. The 0-1 nature of the data was also suggestive of probabilistic approaches.



### 3.1 Baseline Models and rationale

Two baseline models utilised, K-Nearest Neighbour and Naïve Bayes.

#### 3.1.1 Person based KNN:

This is one of two standard collaborative filtering approaches. We consider the K most similar candidates to a particular candidate by hamming distance (in our 0-1 world Euclidean distance is sqrt of Hamming) then get them to vote on the most popular unsolved problems for a particular person. It is hoped that the KNN approach can capture both temporal and categorical factors (and hopefully others too) by considering other people who solved problems in similar time periods and categories without being explicitly told. The value of K will be tuned on the cross-validation set roughly.

#### 3.1.2 Item based approach (Naïve Bayes):

A Pearson correlation based multilinear regression approach applied to the items being recommended is common. However, the unary structure of our problem makes Naïve Bayes (Laplace Smoothed) both faster to update and query as well as be more tailored to our structure (for instance being stronger in cold-start scenarios). It should also be able to capture aforementioned factors. This approach is also significantly faster than the previous one, and does not need hyperparameter tuning.

### 3.2 The More complicated model

We use matrix factorization based collaborative filtering (CF) with feature engineering as a more advanced model. Each user and problem will be assigned $K$ parameters representing latent features as vectors, denoted $u_1 \dots u_n$ and $p_1 \dots p_n$.

Define $U = [u_1 \mid u_2 \mid \dots \mid u_n]$ and $P$ similarly.

(1) The base CF model generates a predictive matrix $U^T P$ where higher values indicate higher likelihood to solve a problem.
(2) Feature engineering:
    a. The problem category is encoded as a one-hot vector per problem $c_i$, and is merged into the matrix C as $C = [c_1 | c_2 | \dots | c_P]$. We define a person's affinity for each category of problem to be a real number, and thus define a user's category bias to be the vector $b_i = (bias\ for\ starter, bias\ for\ aio \dots etc)$. The matrix $B$ is defined similar to C and our new prediction is now $B^T C + U^T P$.
    b. We encode the value of time per problem as the value $t_i = \frac{PROBLEM\_ID_i}{MAX\_PROBLEM\_ID}$ to normalise. We add to user I's prediction for problem j the time-bias value $f_{i,j} = \sum_{k=0}^{Deg} a_i^{(k)} t_j^k$ (a polynomial of t), where Deg is a hyperparameter. In fashion similar to least squares linear regression, we express this as the product of 2 matrices $A^T T$. Thus our prediction becomes $B^T C + U^T P + A^T T$

We train by minimising $Loss\big(\sigma(R - U^T P - B^T C - A^T T)\big)$ where $\sigma(x) = \frac{1}{e^{-x}+1}$ is the logistic sigmoid (used to get values into [0,1]). Due to the potentially probabilistic structure of the problem, we choose logistic loss for our loss function. Further, we will apply a regularization parameter to the parameters in $U, P, B, A$. This parameter, along with K and Deg are hyperparameters to be tuned.

### 3.3 Training and results

The baseline models were implemented with ease. The only thing worthy of note is the hyperparameter of KNN, which is found to be optimal around 500-1000 round (750 was settled on). The matrix model was trained using pytorch's autograd using an AdamW optimizer and a learning rate of 0.05, running for 260 iterations. Various choices of hyperparameters were tried including:

- $Deg = 1,2$
- $Weight\ Decay = 0.01 \times 3^k, k = 0,1 \dots 4$
- $K = 2,3 \dots ,7$

In the end it was found that for the matrix model on cross validation set $Deg = 1, Decay = 0.71, K = 7$ performed best when all feature engineering was applied. Numpy.random was seeded with 42069 for this experiment.

Results on test set:

| Model | Top-5 Accuracy | Top-10 Accuracy | Top-20 Accuracy | Top-S+5 Accuracy |
|---|---|---|---|---|
| **750-NN** | 39.7% | **49.3%** | **62.4%** | 55.0% |
| Naïve Bayes | 38.9% | 47.0% | 57.9% | 52.0% |
| Un-engineered Matrix | <20% | <20% | <20% | <20% |
| Matrix with Categories data | 35.6% | 42.6% | 50.2% | 52.0% |
| **Matrix with Categories and time data** | **40.0%** | 47.8% | 53.7% | **55.6%** |

Surprisingly, 750-NN does the best in Top 10 and Top 20 accuracy, while the Matrix model, barring the fully fledged variant, is generally inferior. However, in the category that arguably matter most for applications (Top-5 and Top-S+5), the fully engineered matrix performs best. It should be noted that all 3 models have comparable performances in every category.

### 3.4 Practical considerations

When applying the model, scalability is an issue. While orac has a small dataset allowing successful application of the 750-NN model, its time complexity is problematic for bigger applications. Note that when comparing models we assume $N$ is much greater than $P$, and that we can update the model periodically rather than every time a user makes an action.

| Model | Update Complexity (less important) | Query complexity (more important) | Amount of persisted data |
|---|---|---|---|
| K-NN | $O(1)$ | $O(NP)$ Optimisable by vector operations e.g bitset for a factor of 32 | $O(1)$ |
| Naïve-Bayes | $O(P)$ | $O(P^2)$ | $O(P^2)$ |
| Matrix | $O(NP * \#its)$ (fast in practice) | O(P) | $O(N + P)$ (low constant) |

Thus, for large / high traffic applications, K-NN is slower (in lieu of advanced data structures). Further, the matrix model is most versatile, memory friendly and allows known data to be engineered in with relative ease as in our case. Therefore, Matrix is by no means suboptimal.

## 4 Thanks

Thanks to Jerry for checking my approach and Quang for introducing me to web scraping.

## 5 Appendix

All code used to produce the results can be found in https://github.com/anonymous3141/Orac-Recommendation-Sytem with the code in each folder doing what is expected of it by the title. However, some potentially sensitive data has been removed in the Data-Collection folder as they contain raw scraped materials.

**Data Collection:** All code in this folder scrapes data from Orac & the backdoor lorikeet. People can adapt the infrastructure as needed for their own scraping.

**Data Processing:** The sole python file takes rough data from scraping and produces the files which are now usable by the machine learning model as train/cross validation/test data. It also computes and plots basic aggregates of the data.

**Models:** recommender.py contains a recommender base class which contains basic code to read in training data, which is used by all models. The evaluate() function is used to benchmark the accuracy of model. In all 3 models KNN.py, Matrix_Factorization.py and Naïve_bayes.py the base class is inherited and the predict() and init() function overridden with aforementioned implementations. Sometimes other code is added to get accuracy and/or tune hyperparameters. If you wish to use the implementations, the files can be moved over as is and necessary IO code added.

**Library dependencies (all installable via pip/available by default):**

| **Web scraping** | **Data analysis & processing** |
|---|---|
| aiohttp | numpy |
| beautifulsoup | pandas |
| asyncio | pytorch |
| | matplotlib |