Github Recommender System

Divyanshu Talwar (2015028)
CSE, IIIT-Delhi
divyanshu15028@iiitd.ac.in

Viraj Parimi (2015068)
CSE. IIIT-Delhi
parimi15068@iiitd.ac.in

***Abstract*—In this work, we used implicit ratings and an auto-encoder with a modified cost function to make a GitHub Recommender System. First, we collect the data, construct the confidence and prediction matrices based on implicit rating schemes. Finally, we train an auto-encoder with a modified cost function and test the trained model using Recall metric.**

***Keywords*—Implicit Rating, Collaborative Filtering, GitHub, Auto-encoder.**

I. Introduction

GitHub is a popular development platform where developers showcase their source codes by uploading it to a repository. This platform offers distributed version control and source code management functionality of Git. Every user who wishes start a new project makes a new repository. Other users, can :

- **Watch a repository** - user gets notifications about changes made (if any) to the watched repository.
- **Star a repository** - A way to "like" the repository.
- **Fork** - to extend/fix bugs in the current source code (shows user's interest in the same type of repositories).
- **Follow another user** - the follower gets notifications about user's public actions.

Collaborative Filtering is commonly used for suggesting movies, songs etc. to users (based on prediction of a user's rating). We noticed that GitHub has no method to suggest repositories to a user and thus, decided to build one as our course project. The idea to recommend users a repository that he/she would like to contribute to, and thus, leading to the growth open source coding culture, highly motivated us.

II. Data Curation

There is no existing dataset available for our purpose, so we wrote scripts to extract the following features using the Github API-v3.

- Users
- Repositories
- Languages - *Repository and User Feature*
- Forks - *Repository Feature*
- Stars - *Repository Feature*
- Watchers - *Repository Feature*
- Users following - *User Feature*
- User followers - *User Feature*

The data - comprising of more than 10K users and 300K repositories - was collected and stored into a MongoDB database. For this project we used a subset of this dataset - a pool of 1000 users and 1000 repositories.

III. Methodology

We used, user-item interactions like stars, forks, watchers, user-followers, user-following and languages matched, as a proxy to indicate user's preference for a particular repository. If a user has starred, forked or watched a repository, it indicates his/her interest towards the same. However, we have no metric which depicts a user's disliking for a repository. This kind of indirect information about user-item preferences is known as implicit feedback.

We then used the data collected to construct a *user * item* confidence matrix 'C', where in the value at $C_{i,j}$ is calculated by taking a weighted sum of the collected implicit features of a user 'i' and a repository 'j'. Since the confidence matrix comprises of non-negative values we used it to construct a preference matrix 'P' with values $\in \{0, 1\}$ at it's indices .

We modified the cost function used in the auto-encoder approach of solving collaborative filtering problems (latent factor model)[1], as explained below.

Given a set S of vectors in R^d, and some number of hidden layer nodes = $k \in N_+$, an auto-encoder solves

$$\min_{\theta} \sum_{r \in S} ||r - h(r; \theta)||_2^2$$

where $h(r; \theta)$ is the reconstruction of the input $r \in R^d$, defined as :

$$h(r; \theta) = f(W.g(Vr + \mu) + b)$$

where, $f(.), g(.)$ are activation functions of the hidden layers of the auto-encoder and $\theta = \{W, V, \mu, b\}$; for encoder matrix $W \in R^{d*k}$, decoder matrix $V \in R^{k*d}$, and biases $\mu \in R^k, b \in R^d$. This objective function corresponds to an auto-associative neural network with a single k-dimensional hidden layer. The minimization parameters θ are learned using the standard back-propagration algorithm.

We used the user-based auto-encoder described in [1] to train on our model. Firstly, each r^i in the training dataset is partially observed, thus we only update the observed indices in r^i while back-propagating. Secondly, we regularize the learned parameters to prevent over-fitting on the observed ratings. Formally, the objective function for the user-based model is as follows :

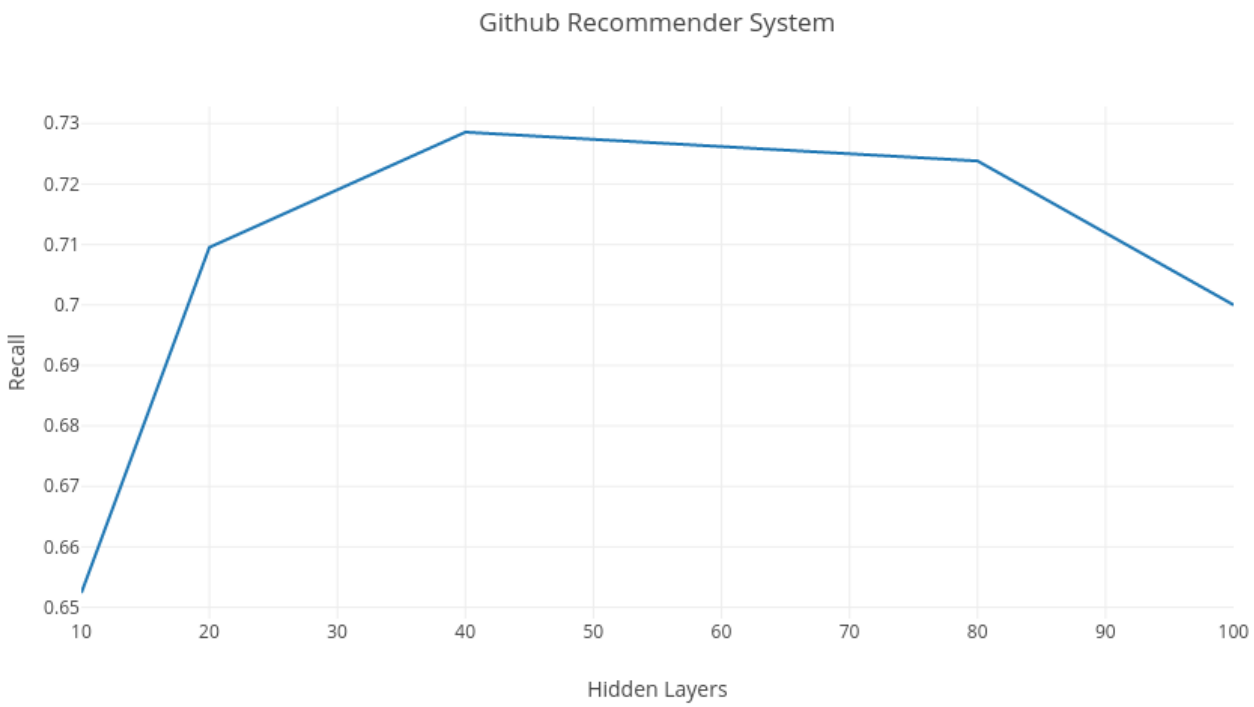

Fig. 1: Recall vs Hidden Layers

$$\min_\theta \sum_u \|c_u\|_2 \| \left(p_u - h(r^u;\theta)\right) \|_O^2 + \frac{\lambda}{2} \left(\sum_u \|W\|_F^2 + \sum_i \|V\|_F^2\right) \quad \text{– (1)}$$

where, c_u = Confidence vector of user u,
p_u = Preference vector of user u,
$||.||_O$ means, we consider the contributions of only the observed ratings.

Once the model is trained then we can predict the preference matrix R as follows:

$$R_{u,i} = \left(h(r^u;\theta)\right)_i$$

Clearly, $R_{u,i} \in \{0,1\}$, where, if $R_{u,i} = 1$, user u would supposedly like the repository i.

IV. Results

To test our model we do an 80:20 split of the data where $P_{u,i} = 1$ to get the test and train data respectively. We choose our test and train data this manner since, we are dealing with implicit ratings and hence do not have any negative feedback/dislike information.

To compare the prediction accuracy the standard metric of recall is used. Recall gives us the probability of a relevant repository being selected for recommendation. Mathematically,

$$Recall = \frac{t_p}{t_p + f_n}$$

where t_p = number of true positives
f_p = number of false negatives

Many studies report results using one of the following metrics : MAE (Mean Absolute Error), RMSE (Root Mean Squared Error), NMAE (Normalized Mean Absolute Error) between the predicted and the ground truth values. But, using one of these metrics for testing the trained model does not make sense when one is dealing with implicit ratings because the predictions and the ground truth values belong to $\{0,1\}$ which is a small set. Thus, the above mentioned metrics would also be small and would not give us the correct quantitative measure of the model's performance.

Precision is another information retrieval metric defined as follows,

$$Precision = \frac{t_p}{t_p + f_p}$$

where t_p = number of true positives
f_n = number of false positives

Since, our test dataset does not contain any negative feedbacks thus, there are no false positives. This renders the use of this metric as meaningless.

Hyper-parameter Tuning :

- We did a grid search on the following hyper-parameters:
  - **Regularization Strength** (λ) over the set $\{0.001, 0.01, 0.1\}$ and got best results for $\lambda_{best} = 0.01$

- **Number of nodes in hidden layer (k)** over the set $\{10, 20, 40, 80, 100\}$. Figure 1 shows that as we increase the hidden layer nodes from 40 to 80 nodes the model performs better, after 80 hidden layer nodes the recall value converges depicting **80** being **the best number of hidden layer nodes**.

- **RMSProp optimizer** was used to minimize the loss function as described in equation (1).
- We tested the trained model for both sigmoid and ReLU **hidden layer activation functions**. The model with hidden layer activation functions being the sigmoid function performed the best.

V. Future Work

- Currently, we trained on our model on a 1000 x 1000 subset of the original 10K x 300K data collected. In the near future we plan on training on the entire dataset to get a better understanding of the trained model.
- The auto-encoder model trained presently, has a single hidden layer. We could stack multiple hidden layers $i.e$ Stacked De-noising Auto-encoders(SdA) and report it's effect on the performance of the model.
- We can even try an item-based auto-encoder model and report it's effect on the model's performance.

References

[1] *Auto-Rec Paper*
Suvash Sedhain, Aditya Krishna Menon, Scott Sanner, Lexing Xie

[2] *Implicit Recommender System*