Hybrid Paper Recommender System

Wenyuan Feng, Zheyu Liu and Huizhe Li

1 Introduction

Researchers and academics have long been disturbed by time-consuming search of relevant papers. We aim at building an content-based paper recommender system that assists users in searching for papers of interest using machine-learning approaches.

2 Data Set

Our data set came from a subset of ACL papers, containing a total of 596 papers and opinions from 28 reviewers. Necessary pre-processing of this data set is discussed in the following sections in detail.

2.1 Dictinary Setup

We used the Word-presence model to represent the feature of a paper. Therefore, a dictinary, consisting of all the words that have appeared at least one time among all the papers, is to be built before we could proceed.

Luckily, our data set has the OCR-results of candidate papers in XML format, which significantly lightens our burden. By calling a built-in XML parser provided in the Java language, we are able to obtain all candidate words within half a minute.

However, this preliminary results cannot satisfy our needs, mostly because a huge number of "words" obtained in this way suffer from limitations of OCR. Typically, we have "words" that start or end with:

- *a hyphen.* This happens when a word is concatenated by a hyphen at the end of a line.
- *a comma, period, parenthesis or such alike.* This is purely caused by OCR-error.
- *numbers.*

This sort of invalid words can be eliminated by applying a regular expression filter.

Another issue can be described as redundancy caused by suffixes. A thorough exploring of this type of redundancy may be unpractival, so we simply merged words from same roots ending with "ed", "ing", "ly" and "s".

The size of the dictionary generated in this way reaches $30,000$ words, still a giant number. Further work needs to be done before we could pass a feature to our algorithm. But we will take a detour and first look at how we extracted features for each paper.

2.2 Feature Extraction

Having built the dictionary, this task becomes a trivial one. Just as defined by the Word-presence model, a paper is represented by a binary vector having the size equal to that of the dictionary, each element having values of 1 or 0 indicating whether a word is present in this paper or not. Each paper was scanned word by word and its feature extracted and stored in a plain-text file data.txt.

2.3 Before We Sail

It is unwise to carry a mountain with us before we sail, yet a vector of $30,000$ elements means even more than a mountain. To make our problem computable, we need to further reduce the size of our features. This was done by functional filtering.

Consider a word that appears:

- *very few times among all papers.* In the extreme case, a word that appears only one time gives no useful information. Unfortunately, we found more than $7,000$ isolated words in the dictionary.
- *in almost every paper.* This is essentially the same as the above case from the viewpoint of Information Theory.

With the features represented in the form of a binary matrix, it is very convenient and efficient to use Matlab in eliminating uninformative words. We first examined the distribution of word-presence, and decided to consider only words that appear between 10 and 50 times among all candidate papers. The number of effective words in our final dectionary was 3031.

With all the preliminary works done, we are now ready to apply our methods.

3 Method

Two machine-learning algorithms, i.e. AdaBoost and Artificial Neural Network, were applied to learn users' preferences. They will be discussed in the following sections respectively .

3.1 Adaptive Boosting

Boosting, as a meta-algorithm, combines "weak" learning algorithms (algorithms that give slightly better prediction than a pure random predictor) into an arbitrary strong one. We chose the AdaBoost, one of the implementations of Boosting algorithm, to combine weak learners generated by the Naive Bayes model and Decision Tree model, into an effective predictor.

The key idea of AdaBoost is to maintain a weighting distribution over input samples, and update this distribution after each iteration. Note that for this reason, the weak learners must have the ability to take advantage of weighted input samples before AdaBoost can be applied. In our project, we used different ways to adjust Naive Bayes and Decision Tree so that they satisfy the condition to be a weak learner. We present the details as follows.

Given: $(x_1, y_1), \ldots, (x_m, y_m)$ where $x_i \in X$, $y_i \in Y = \{-1, +1\}$
Initialize $D_1(i) = 1/m$.
For $t = 1, \ldots, T$:

- Train weak learner using distribution D_t.
- Get weak hypothesis $h_t : X \to \{-1, +1\}$ with error

$$\epsilon_t = \Pr_{i \sim D_t}\left[h_t(x_i) \neq y_i\right].$$

- Choose $\alpha_t = \frac{1}{2} \ln\left(\frac{1-\epsilon_t}{\epsilon_t}\right)$.
- Update:

$$\begin{aligned} D_{t+1}(i) &= \frac{D_t(i)}{Z_t} \times \begin{cases} e^{-\alpha_t} & \text{if } h_t(x_i) = y_i \\ e^{\alpha_t} & \text{if } h_t(x_i) \neq y_i \end{cases} \\ &= \frac{D_t(i) \exp(-\alpha_t y_i h_t(x_i))}{Z_t} \end{aligned}$$

where Z_t is a normalization factor (chosen so that D_{t+1} will be a distribution).

Output the final hypothesis:

$$H(x) = \operatorname{sign}\left(\sum_{t=1}^{T} \alpha_t h_t(x)\right).$$

Figure 1: The AdaBoost Algorithm

3.1.1 Applying AdaBoost to Naive Bayes.

When a certain iteration ends, the distribution matrix **D** is updated with values for the next iteration. Instead of directly applying this matrix, we sort it by rows(samples) and wipe out $N * F$ samples having the lowest weight, where N denotes the number of current samples and F the reducing factor, a hyper-parameter. In such a way we manage to force the model to focus on difficult samples, since having a low weight implies being correctly recognized in the current iteration. After that, the matrix **D** is cleared as $1/N$. This procedure iterates T times, another hyper-parameter, generating weak learners of the corresponding number.

In our implementation, $F = 0.1$ and $T = 10$. We made this choice by testing our model over spamXY.mat from homework assignments. According to our observations, a F value that was too small would require more iterations

to focus on difficult samples; and the weak learner generated in each iteration tended to "converge" in terms of performance after about 10 iterations.

3.1.2 Applying AdaBoost to Decision Tree.

With the Decision Tree model we have a convenient way to utilize data weightings. Recall that splitting a node involves calculating information gain, which can be obtained by computing probabilities. Here we sum up the weightings and normalize the result to get the probabilities, instead of naively counting numbers. This can be formally described as follows:

$$P = \sum_{i \in left} W_i / \sum W_i \tag{1}$$

Note that for a mis-classified sample (thus having a higher weight) in a leaf node, the new entropy (or any equivalent measurement of impurity or information) would become larger, which then reduces the total information gain, thus making the very split less likely to be taken.

The split constant for the decision tree was set to 0.05, which yielded a good balance between correctness and efficiency.

3.1.3 Building Up the Final Booster.

With the above "components" in hand, all that left for us was to compute the soft prediction, i.e. the weighted sum of hard predictions of each weak learner. Then, a threshold had to be specified to finally label a paper. Different choices of the threshold value yielded interesting results and had practical implications, so we leave the discussion of this parameter to the "result & discussion" section.

3.2 Artifical Neural Network

We were motivated to implement ANN since samples in our project are probably non-linear separable, given the huge size of their feature vectors. We assumed that it was a good stage where ANN could prove its representaional power.

We implemented a two-layer neural network (with one hidden layer) and chose $g(x)$ to be $\frac{1}{1+e^{-x}}$. The cost function of the neural network is shown in fig. 2. The learning rate was set to 0.1 to guarantee convergence as well as to avoid time consumption problem in the training stage.

Our first choice of one thousand neurons suffered from overfitting, without noticable performance improvement. Also, when the number of neurons reached below 60, a significant performance decrease could be observed. Finally the hidden layer was set to have one hundred neurons, which was in accordance with the complexity (feature size) of our samples.

The initial weights could simply be randomly set. However, naively applying $rand()$ proved problamatic as the algorithm took very long time to converge. A better initialization was found to be random numbers between -2 and 2 by performing brute-force tests.

$$-\frac{1}{m}[\sum_{i=1}^{m} y^{(i)} \log(h_\theta(x^{(i)})) + (1 - y^{(i)}) \log(1 - h_\theta(x^{(i)}))]$$
$$+ \frac{\lambda}{2m} \sum_{l=i}^{L-1} \sum_{i=1}^{sl} \sum_{j=1}^{sl+1} (\theta_{ji}^{(l)})^2$$

Figure 2: Cost Function of the Neural Network

4 Results & Discussion

Two criteria were used to measure performance:

- *Accuracy.* Accuracy was defined as the number of correctly recommended papers divided by total number of recommended papers.
- *Miss rate.* Miss rate was defined as the number of unrecognized relevant papers divided by total number of relevant papers.

Note that the above two criteria are generally in conflict with each oter. In a real-world application, a balance between them would be the goal.

4.1 AdaBoost

In this section, we first discuss the overall performance of four algorithms applied on the same data set in terms of the criteria mentioned above, and then focus on AdaBoost to explore the effect the threshold has on perfermance, as well as time consumption of AdaBoost.

4.1.1 Overall Performance

In this test, four algorithms, i.e. Naive Bayes, Decision Tree, Boosted Decision Tree and our final model are compared in terms of accuracy and miss rate. The data set was split on a 80-20 basis in training and testing, respectively.

From fig. 3 we could see that the Naive Bayes model has good accuracy, but suffers badly from a high miss rate, which means that although it does not recommend a lot of "spam" papers, it could only recognize 2 3 out of 10 relevant papers. From this point of view, we argue that our model (denoted as "Boosted DT & NB" in the figure) yielded the best results, as it both has good accuracy, though not the highest, and the lowest miss rate, which is a valuable feature for a paper recommender system.

4.1.2 Determining the Threshold

Threshold is the value against which we compare with the soft prediction given by our model to label a paper. Intuitively, a high threshold will guarantee high

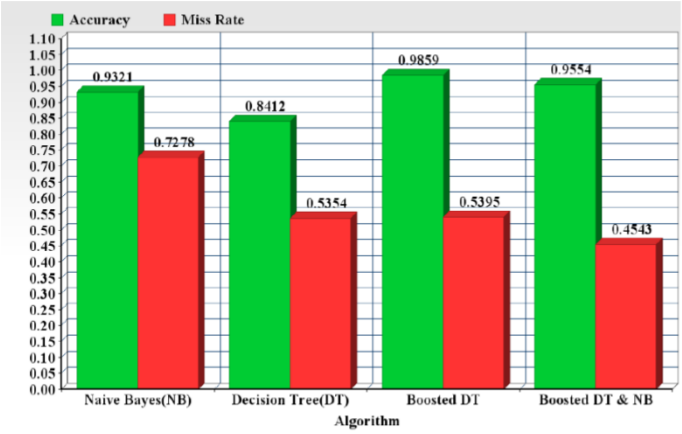

Figure 3: Overall Performance of Four Algorithms

accuracy, but will miss more relevant papers. This assumption is verified by fig. 4.

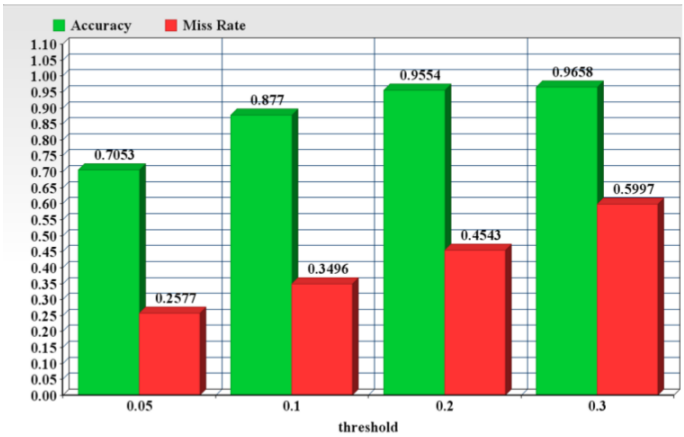

Figure 4: Performance Under Different Threshold Values

In practice, this value can be determined by looking at user preference or configuration of the application. A high threshold is useful for a researcher who is not willing to waste time in reading irrelevant papers, while a low threshold is helpful for one with a broad interest.

4.1.3 Time Consumption

Since our model iterates dozens of times over weak learners (see fig. 5), it consumes significantly more time compared to the original ones. However, the

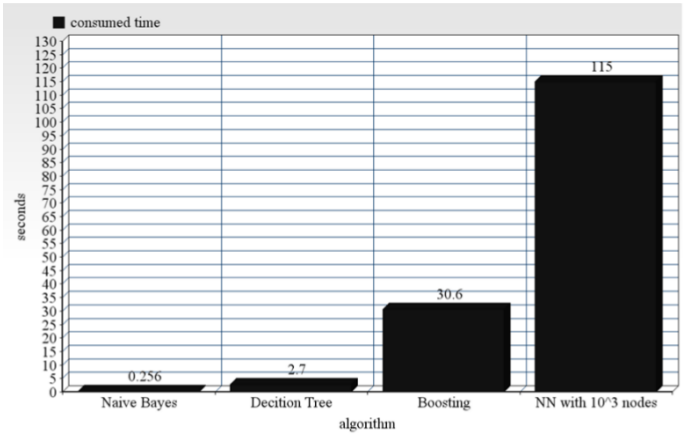

Figure 5: Time Consumption

performance benifit gained form this may not worth the time and energy it takes, especially in an intensive environment. The same conclusion holds for neural networks.

To explain, we propose that the "weak" learners in this scenario is not "weak" enough to show significant performance improvement, while the iterations still consume as much time. For example, from fig. 3 we can find that the Boosted DT only has slight improvement in accuracy compared to DT, which was gain at a cost of 10-time iteraion.

We leave the discussion of neural network to its own part that follows.

4.2 Artifical Neural Network

The first five hundred papers were used as training set and the rest test set. ANN was surprisingly strong on the training set - having an error rate of 0%. However, its performance decreased badly when applied to test set (see fig. 6 and fig. 7).

We have two assumptions for this overfitting problem. First, our data set was too small, which easily causes overfitting. Second, ANN is such a complex model that we have not found a best way to tune its parameters. This problem is even worse if we take into consideration the time it consumes to train a neural network.

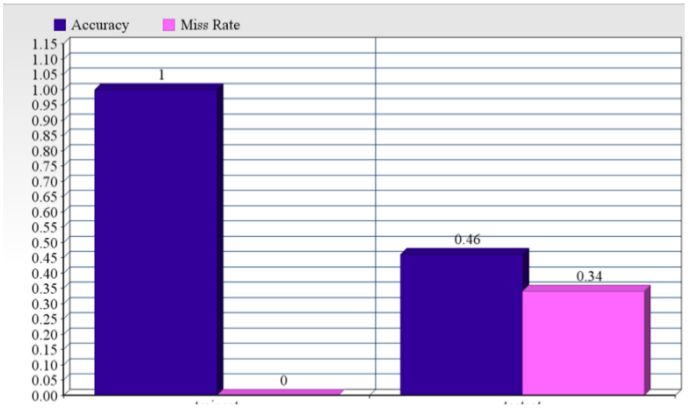

Figure 6: Performance on Training Set (left) & Test Set (right)

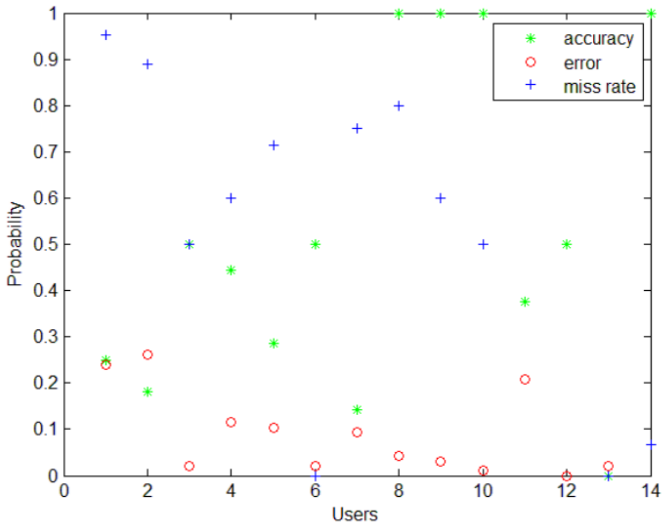

Figure 7: Results From Running ANN on Test Set

References

[1] Robert E. Schapire. *The Boosting Approach to Machine Learning-An overview.* 2001

[2] Yoav Freund & Robert E. Schapire. *A Short Introduction to Boosting.*

[3] Byron P. Roe, Hai-Jun Yang & Ji Zhu. *Boosted Decision Trees, A Powerful Event Classifier.*

[4] D. Wilson & T. Martinez. *The Need for Small Learning Rates on Large Problems.* 2011

[5] N. Srivastava, A. Krizhevsky, I. Sutskever & R. Salakhutdinovar. *Improving neural networks by preventing co-adaptation of feature detectors.* 2012