CS 229 Project Report
Keyword Extraction for Stack Exchange Questions

Jiaji Hu, Xuening Liu, Li Yi

1 Introduction

The Stack Exchange network is a group of question-and-answer websites with each site covering a specific topic, including Stack Overflow for computer programming questions. Similar to many websites, Stack Exchange sites use tags for fast item retrieval and flexible grouping, ultimately providing better user experience and easier management.

In this project, we build an automated keyword extraction system for Stack Exchange questions, which generates suggested tags for new questions based on the title and content of that question. This system has advantages over the status quo in terms of correctness and convenience.

Our keyword extraction system takes a question's title and text content as input, and outputs no less than one and up to five tags deemed to be suitable for the question. For each possible tag, a classifier is trained to predict the presence of that tag. Classification results from all the predictors are then aggregated to return the final output.

Our project focuses on evaluating a number of different kinds of features extraction methods as well as classification methods, in an attempt to gain insight into their effectiveness in the field of keyword extraction. Detailed test results are shown and error analysis given in the report, and cases regarding specific tags are discussed to reach preliminary conclusions.

2 Prior Work

Keyword extraction is a topic that has been generating increasing interest both in industry and academia. According to [3], well known keyword extraction algorithms rely on the fundamental concept of TF-IDF. In [1], a Naive Bayes Classifier was used along with TF-IDF for keyword extraction.

3 Data Collection and Processing

Data for training and testing is collected from the Stack Exchange sites. Stack Exchange provides a dump of its data every three months, which is free for us to use. We also collected data from a competition on *Kaggle.com*.

Our input data format has 'title' and 'content' fields representing the title and content of the question, and a 'tags' field which indicates the "correct answer" tags for that question.

After our initial collection of data, we had around 7 GB of text data, which contained around 6 million examples. In the process of developing our system, due to constraints in memory and time, we used up to 330,000 examples for training for testing. Fig. 1 shows the word frequency distribution of the first 10,000 examples in our dataset.

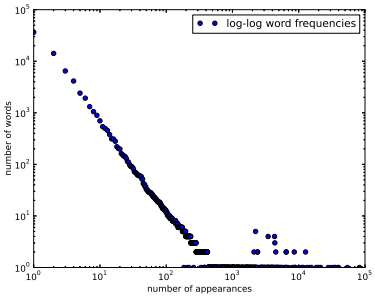

Figure 1: Word frequencies

From Fig. 1, we see that around 50 % of all words in the generated dictionary appear only once. Another 20% of words appear only twice. This will result in very sparse unigram feature vectors. The issue of sparse features is addressed when we describe our feature extraction later in section 5.2.

4 Model

Our keyword extraction model is as follows:

The model reads in two strings, 'title' and 'content' as input and performs unigram feature extraction on these strings to return a feature vector for that input. Then, the feature vector is input into a group of binary classifiers (one classifier for every possible tag) that predicts whether an input should have the specific tag. Finally, the classification results are aggregated so that all the tags where the predictor output was positive will be our final output. If a question is not assigned at least one tag in this process, we compare probabilities to give it one tag. If the system gives a question more than five tags, we take the five most common tags as the output.

Through practice, we find that the two cases above do not occur often, and throughout the project, we have focused on improving the effectiveness of the binary classifiers as opposed to working on aggregating their outputs.

We used the F1 score as the evaluation method for our system. The reason is that in our problem, the accuracy of the binary classifier may not be a good indication of the effectiveness, since for any tag, the vast majority of examples should not be assigned that tag. Therefore, a trivial predictor that always predicts 'No' would in fact reach very high accuracy, even though it is practically useless.

In cases like these, the F-score is a good measure to measure a test's accuracy. The F-score is computed by considering both the precision p and the recall r of the test. The F_1 score is calculated as follows:

$$F_1 = \frac{2 \times p \times r}{p + r} \tag{1}$$

In our project, we try to maximize the F1 score of our classifier.

5 Methods

5.1 Baseline System

For our baseline system, we used a Naive Bayes classifier using the multinomial event model and Laplace smoothing. We used unigram features extracted from the text without additional processing. We trained the Naive Bayes classifier on some tags to observe its performance. An example result is shown in Fig. 2.

From the results, we observed that due to the large skew towards negative examples, the Naive Bayes classifier was extremely hesitant to classify examples as positive. This resulted in a low F1 score, where precision was relatively high, but recall was very low. Note that even the training F1 score could not go higher than around 0.6.

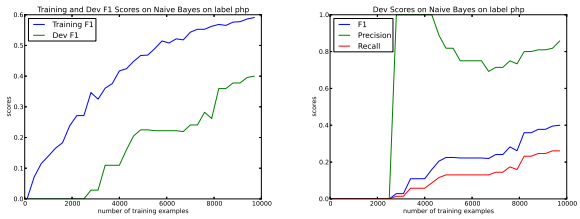

Figure 2: Learning curve and F-score for Naive Bayes

To improve out baseline system, we worked on both the classification algorithm and the feature extraction step. For the classification algorithm, we switched to a Support Vector Machines, and moved on to develop methods of enhancement such as boosting. For feature extraction, we tried four different methods with varying results.

5.2 Feature Extraction

For the baseline system, we first used word frequency to form unigram feature vectors. The feature vectors obtained through this process were very sparse. To address this issue, we tried stemming, stop-word removal and L1-based feature selection.

5.2.1 Stemming

Stemming is a practice often used in Natural Language Processing which reduces words to their roots. This process condenses the unigram feature vector by combining similar words. We used the Porter stemming algorithm on our text inputs with the following results:

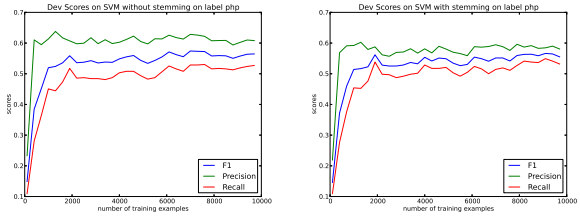

Figure 3: Classification results with and without stemming

With stemming, we reduced the feature vector dimension by 12%. However, results show that stemming causes the system to have lower precision, higher recall, and similar F1 score to a system without stemming. This means that stemming makes the system more prone to classifying examples as positive, but adds a fair amount of false positives.

We believe that the technical nature of our input caused trouble with the stemming algorithm, and by stemming, we may be losing information useful for classification. Therefore, we did not continue to use stemming for our feature extraction.

5.2.2 Stop-word Removal

For stop-word removal, we studied methods of removing particular words from our dictionary without affecting the classification performance. We eliminated stop words and rare words whose occurrences were below a certain threshold. In particular, by removing words that appeared only once in the whole training set, we were able to cut the feature dimension by 51%. If we moved the threshold to 2, we would cut another 18%.

Through testing, we found that removing one-occurrence words did not result in a noticeable difference from the classification results of our system. However, removing two-occurrence words slightly lowered the F1 score. Therefore, we concluded that it was only safe to exclude one-occurrence words.

In addition, since different words may have different importances for classification, we tried using some weighting method to filter out unimportant words and highlight indicative words. We tried using TF-IDF to do this, but had little success.

5.2.3 L1-based Feature Selection

Though removing rare words cut our feature dimension approximately by half, we were not satisfied with this rough method. For our feature extraction, we moved on to try out feature selection. The method we chose was L1-based feature selection.

After we acquire the feature matrix for our training set, we train a linear SVM using the L-1 penalty. The training results in a learned weight vector, which represents how important the classifier believes each feature dimension to be. Using the L-1 penalty, the learned weight vector will be sparse, with many zeros in the weight vector. For our feature selection, we do the following: For each feature dimension, if the same dimension of the weight vector is zero, discard that feature dimension. Using this method, we obtain a scheme for discarding and retaining features from the original feature matrix. In the future, we only select the features retained by the feature selection process.

The learning curves for a linear SVM classifier with features before and after feature selection is shown in Fig. 4.

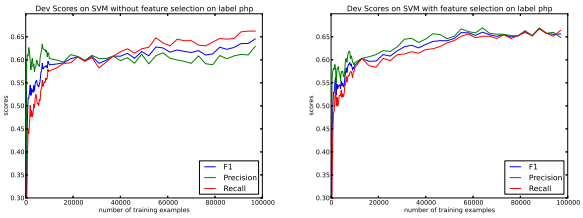

Figure 4: Classification results with and without feature selection

Using L1-based feature selection, we were able to reduce our feature dimension from 520,000 to 8,000 — a 98% reduction. Moreover, the classification F1 scores did not suffer from the lower feature dimension, and in fact even rose in most experiments.

5.2.4 Separating Title and Content

After we found a good way to reduce our feature dimension, we were free to try ways to increase features so that more useful information was input to the classification algorithm. We believed that the question's title contained different information from its content, and that we would benefit from treating them differently. Since we were able to separate the question title and content in our inputs, we tried treating words in the title and words in the content as different features, effectively doubling our feature dimension. We trained our classifier on the features before separation and after separation, with results shown in Fig. 5.

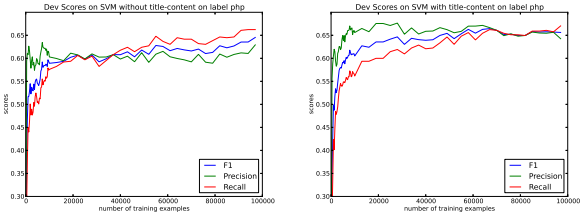

Figure 5: Classification results with and without separating title and content

From Fig. 5, we see that when the number of train-

ing examples is small, the performance of the classifier is significantly better with separating titles and content. As the amount of training data increases, separating title and content for features still performs better, with an F1 score approximately 0.02 higher when there are 100,000 training examples. We believe that separating title and content for features provides extra useful information to the classifier, so it needs fewer training examples to perform well. When the amount of training data is high, the extra information still helps raise performance.

5.3 Learning Algorithms

5.3.1 Support Vector Machines

To improve on our baseline system, we turned to using SVMs as our classification algorithm. After trying different kernels, we found that the linear kernel was easier to train and suffered less overfitting problems. Fig. 7 shows the performance of the rbf kernel and linear kernel on different tags. We see that the linear kernel is more consistent and does better on average.

5.3.2 Boosting

One major challenge in our problem is that the training examples for any tag are heavily biased towards negative examples. Therefore, the positive examples are more likely to be misclassified by our SVM (Note that we do very well in terms of true negatives). The boosting mechanism forces component classifier to focus on the misclassified examples, which makes it very suitable for our classification problem and evaluation method. According to [2], AdaBoost with heterogeneous SVM can work better compared with generally used AdaBoost approaches with Neural Networks or Decision Tree component classifiers. The problem is how to generate such kind of diverse and moderately accurate (weak) SVM component classifiers. [2] suggests using RBF-SVM, whose parameter σ can be adjusted to balance between accuracy and diversity, and to obtain a set of moderately accurate RBF-SVMs for AdaBoost, since large σ corresponds to less accurate classifiers and gives a chance that two classifiers can disagree with each other more. We adopted and implemented the Diverse AdaBoost SVM approach proposed in [2] and achieved better generalization performance than a single SVM. Experimental results and comparisons are made in Fig. 7 in section 6.1.

One caveat of our boosting algorithm was that it was very memory and computation intensive. As the number of examples and feature dimensions grew, the time and space requirements for training and storing component classifiers became more than we could handle. Therefore, we were unable to conduct our experiments on the full dataset, and only show results obtained on smaller training sets.

6 Results

6.1 Classification Performance

For our final system, we used unigram features after separating title and content and L1 based feature selection for the feature extraction step. For the classification step, we used a linear SVM with parameter C set by cross-validation. Fig. 6 and Table 1 shows the precision, recall and F1 score of the system on label 'php'. The performance of the system is similar on other tags.

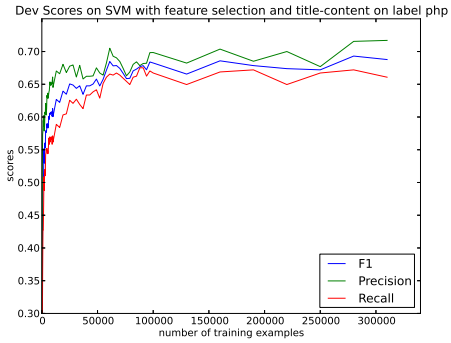

Figure 6: Classification results for label 'php'

	pos	neg
pos	418	184
neg	207	9215

Precision	0.694
Recall	0.669
F1 score	0.681
Accuracy	0.961

Table 1: Confusion matrix and classification statistics for label 'php'

From Fig. 6, we see that the performance of the system gets better as the number of training examples increase. With 330,000 examples, we achieve an F1 score of 0.681, which is a good improvement on our baseline system. To study the effect of the additional improvements to feature extraction that we implemented, we trained an SVM using features without feature selection or title-content separation. The resulting F1 score was 0.64. Therefore, it is confirmed

that our feature extraction methods result in an improvement of performance.

For boosting, due to the time and memory constraints, we could not run the Diverse AdaBoost SVM algorithm on the whole dataset. Instead, we used 1000 examples for training and testing, and compared the performance of Diverse AdaBoost SVM with a single SVM with linear or rbf kernel. Results in Fig. 7 show that AdaBoost SVM on average gets a 30% performance improvement compared with a single SVM.

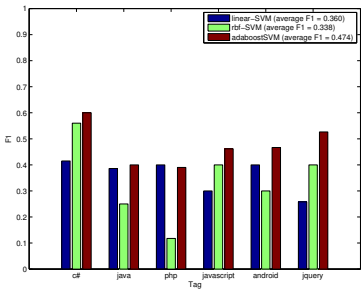

Figure 7: Comparison between AdaBoost-SVM, linear SVM and RBF-SVM

6.2 Feature Analysis

Using the weight vector learned by the linear SVM classifier, we can examine the weights to find out which are the most predictive features of any particular tag. By doing this, we can get a view of what our classifier learned from all the training data.

After analyzing the weight vectors, we found that the weight vector was rather noisy. For example, in the case of label 'java', some words that did not seem to be related to Java made it into the top 10 features. This shows that the classifier still suffers from overfitting. However, there were still learned features that were immediately recognizable as good features — for example, 'java' and 'eclipse' were both top features for label 'java'.

6.3 Error Analysis

We also analyzed some test examples that our classifiers got wrong. In our opinion, there were three main types of reasons for classification errors:

1. There was not enough information in the input. Some of the false negative examples were very short and did not contain the target label in the title or text. Our classifier was not able to extract enough information from the words to make a positive prediction. This is the most popular error.

2. Our classifier was mislead. If a strong keyword appeared in the text, the classifier is inclined to predict positive. However, the keyword may have been mentioned in passing, and was not a key part of the question. For example, "I know this is possible in php, but how to do it in Ruby-on-rails?" would made our 'php' classifier produce a false positive prediction.

3. The correct label was noisy. Some of the labels that our classifiers predicted made sense. Unfortunately, the given 'correct answer' did not contain that label. In that case, we got false positives.

7 Conclusion

In this project, we designed a system to predict tags for questions on Question-and-Answer sites such as Stack Overflow. We trained binary classifiers for different tags, experimenting with feature extraction methods and classification algorithms. For feature extraction, we studied the effects of stemming, stop-word removal and L1-based feature selection on feature dimension and classification performance. We also performed feature engineering by separating question title and content to improve performance. Improved feature extraction provided a 6% improvement on F1 score. In terms of classification algorithms, we tested and compared Naive Bayes, linear and RBF Support Vector Machines, and implemented AdaBoost-SVM with good results. Our final system consistently achieves an F1 score of over 0.65 for most tags.

References

[1] Eibe Frank, Gordon W Paynter, Ian H Witten, Carl Gutwin, and Craig G Nevill-Manning. Domain-specific keyphrase extraction. 1999.

[2] Xuchun Li, Lei Wang, and Eric Sung. Adaboost with svm-based component classifiers. *Engineering Applications of Artificial Intelligence*, 21(5):785–795, 2008.

[3] Brian Lott. Survey of keyword extraction techniques. 2012.