Tag Recommender for StackOverflow Questions

CSE 255 Assignment 1

Manindra Kumar Moharana,
University of California, San Diego
mmoharana@cs.ucsd.edu PID: A53068263

Recommending tags is a one of the more challenging problems in data mining of text. Tagging lets users explore related content, and is very useful on question answers sites like Stack Overflow, Stack Exchange sites, Quora, internet forums, etc. In this paper, I tackle the problem of suggesting tags for StackOverflow website by analysing the title and body of the question. I use perceptron, multinomial naive bayes and linear svm classifier for prediction and compare their performance.

Categories and Subject Descriptors: H.5.2 [**Data Mining**]: Recommender Systems, Machine Learning—

1. INTRODUCTION

Tags are keywords or terms associated to a piece of information/data that act like meta-data. They are useful in describing and locating the piece of information. In 2003, Delicious, the social bookmarking website, allowed its users to add tags to their bookmarks to group related bookmarks. Flickr added a similar feature for tagging images, which greatly improved image search. The success of tags started a trend where more and more sites began to use them to identify related content. Today tags can be found on a variety of websites like Twitter, Youtube, internet forums, etc.

In this paper, I focus on StackOverflow and the tags related to questions posted on the site. StackOverflow is the most popular site available today for software developers to post technical questions and get answers from the community. StackOverflow uses tags to suggest users related questions. My objective is to build a model that can accurately recommend tags for a new question, given a large corpus of questions with existing tags.

2. DATASET

The dataset of questions is obtained from a Kaggle competition [FB-Kaggle 2013] . The training set is 7 GB in size. The dataset contains over 6,034,195 questions and 42,049 unique tags. The test set is 2 GB in size and contains over 2 million questions. Each question in the training set contains four data points - id, title, body and tags (figure 1).

Tags	c#	java	php	javascript	android	jquery	c++	python	iphone	asp.net
Occurrence	7.721%	6.7976%	6.5184%	6.0624%	5.3306%	5.0346%	3.3182%	3.1192%	3.061%	2.927%

Table I. : Top 10 tags

The distribution of tag frequency, shown in figure 2, seems to follow the power law. The top 10 tags and their percentage occurrence is shown in table I. The top 20 tags were {c#, java, php, javascript,

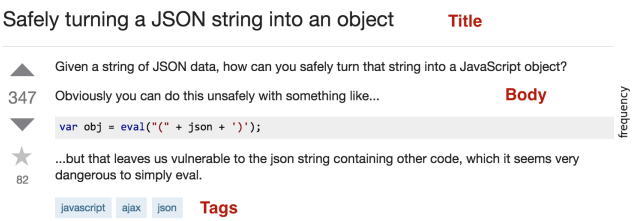

Fig. 1: One of the training examples

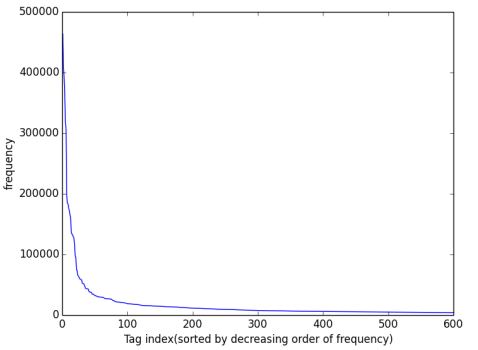

Fig. 2: Tag Frequency distribution(top 600 tags)

android, jquery, c++, python, iphone, asp.net, mysql, html, .net, ios, objective-c, sql, css, linux, ruby-on-rails, windows}. The majority of the most frequent tags were programming languages. In the unprocessed data, the total number of unique words was 33,235,535.

3. PREDICTION TASK

The objective is to predict the tags for a question by analysing the title and body of a question. Given the large size of the dataset, I use the first 500K data points, and predicted the top 20 tags for each question. The top 20 tags were identified from the training set. I used supervised learning, with a 75%:25% split between training and test set.

To evaluate the performance of a model, I calculated the error rate and F_1 score for prediction on the test set. The F_1 score is given by

$$F_1 = \frac{2 \cdot \text{precision} \cdot \text{recall}}{\text{precision} + \text{recall}}$$

I decided to use the perceptron algorithm as the baseline.

4. PREVIOUS WORK

There has been some prior research on predicting tags for StackOverflow. [Kuo 2011] used a co-occurrence model that predicts tags based on token in the question title and body, and their co-occurrence to tags. One of the tasks in the ECML PKDD Discovery Challenge 2009 was online tag recommendations (but not on the StackOverflow dataset). The winning solution [Lipczak et al. 2009] of the challenge used a 3 stage tag recommender with the second and third stages consisting of modelling title $\leftrightarrow$ tag and tag $\leftrightarrow$ tag relationships using graphs. The tags were predicted after combining scores from the three stages. [Stanley and Byrne 2013] used a ACT-R based Bayesian probabilisitic model. The tag probability depends on likelihood of retrieving a tag, given the tag's prior and by updated by context. They added an offset term which was not part of the original ACT-R retrieval equation.

5. PREPROCESSING

Given the raw, unprocessed nature of the data, some preprocessing was needed. The text was converted into lower case. All numbers in the text were trimmed out. Single letter words were trimmed out, ex-

cept for 'c' (to make sure C programming language tag was not affected). Stop words were removed using the python NLTK library. Words which occurred only once were removed from the vocabulary. Both common english words and non-common frequent words(' I', ' ', '</code></pre>') were removed. I tried using word stemming to reduce words to their basic form using the NLTK [Loper and Bird 2002] Snowball stemmer, but didn't use it in the end due to slight decrease in cross-validation accuracy. The body often contained multiple XML tags like <a>, <code>, <img> and <ul>. I used the Beautiful Soup python library [Richardson 2009] to remove these tags. With all of this preprocessing, the vocabulary size was finally reduced to 73,788 (from the original 33 million) for the training set of 375K samples. Once the data was preprocessed, it was saved to disk to avoid preprocessing on every run. This helped reduce training time.

6. FEATURES

The title and body of a question was combined into a single string and a bag of words representation was used to model the text. Essentially, the frequency of each word of the vocabulary in the document is used as a feature. To improve memory efficiency, sklearn's Hashing-Vectorizer was used. This vectorizer does not make use of a dictionary to store the index of words in the vocabulary. Instead, it directly computes the hash of each word and uses it as the index. Therefore, vocabulary word to index mapping is no longer required.

I also experimented with a second vectorizer – Tf-idf (term frequency – inverse document frequency). Tf-idf is a method to measure the importance of a word in a document based on how frequently it appears in multiple documents. The motivation behind tf-idf is that if a word occurs frequently in multiple documents belonging to different categories, it is not very useful in distinguishing between the documents and therefore its tf-idf score is low. The tf-idf weight W of a term t in a document d (from N total documents) is given by:

$$W_{t,d} = (1 + log(\text{tf}_{t,d})) \times log_{10}(\frac{N}{\text{df}_t})$$

where $\text{tf}_{t,d}$ is the frequency of term t in document d and df_t is the document frequency of documents containing the term t.

I also observed that a 56% percent of the questions contained the <code> tag, i.e., they contained some code fragment. I decided to use this as a feature instead of discarding it completely. I tried to determine the language given a fragment of code. I used the pygments python library's lexer to guess the programming language. Pygment's lexer can guess 90+ languages. The lexer returns a list of detected languages (multiple languages were returned when the lexer wasn't confident about a single language). I created a bag of words representation using the detected language names for each example. This representation was added as a feature.

Finally I added 2 other binary features based on whether the question text contains a link (<a> tag) and code(<code> tag). The has-code feature was added on top of the bag-of-words code-language feature because the pygments code lexer was not always able to identify the language. The final feature vector consisted of:

$$x = \{\text{bag-of-words/tf-idf vector, has-url, has-code, code-language bag-of-words}\}$$

7. MODEL

I tried three different classifiers for the prediction task - perceptron, multinomial naive bayes and linear SVM. I trained 20 classifiers – one for each of the 20 tags. This was required because the tags are not mutually exclusive. Multiple tags could be associated with each question. To determine the tags for a question, its content was run through each of the 20 tag classifiers and the result was combined. The training set was split 75%:25% into training and cross-validation set. The validation set proved useful in tuning the parameters of the models. The final error rate and F_1 score were calculated by taking the mean values of error rates and F_1 scores of the 20 tags.

7.1 Perceptron

Perceptron model was chosen as a baseline. It is a simple model that makes the classification decision by considering only the base dimension of the feature vector. Perceptron algorithm is one of the earliest machine learning algorithms. It works by learning a set of weights, which are combined with the feature vector. The prediction is based on the dot product of feature vector x and weight vector w:

$$f(x) = \begin{cases} 1 & w \cdot x + b > 0 \\ 0 & otherwise \end{cases}$$

I used sklearn's implementation of Perceptron classifier. l_2 penalty gave better cross validation performance compared to l_1.

7.2 Multinomial Naive Bayes

Multinomial Naive Bayes is a basic yet surprisingly accurate classifier for document classification tasks. Multinomial NB assumes each feature f to be linearly independent and calculates the probability of a class by:

$$c_d = \arg\max_{c \in C} \left[P(c) \prod_{f_i \in d} P(f_i|c) \right]$$

sklearn's MultinomialNB implementation was used for the model. Smoothing parameter $\alpha = 0.1$ was chosen using cross-validation.

7.3 Linear SVM

Linear SVM is one of the best suited classifiers for this classification task because of its ability to perform well with high dimensional data (using the 'kernel' trick). SVM algorithm works by finding the maximum separating hyperplane between points of 2 classes in high dimensional space. Another advantage of Linear SVM is that it scales linearly to the size of training data. For the Linear SVM classifier, I used sklearn's LinearSVC class. I used squared hinge loss function as it gave best accuracy on the cross-validation set. The tolerance for stopping criteria was set to 0.001.

8. RESULTS

The results from the three classifiers are shown in the tables II and IV. Perceptron had the lowest accuracy while SVM had the highest accuracy for both representations. It is worth noting that all the

three algorithms performed better with tf-idf representation, reinforcing the claim that tf-idf is well suited for text classification tasks as it down weights frequently occurring words in documents.

	Error		F_1 Score	
Features →	Text Only	Text + Meta	Text Only	Text + Meta
Perceptron	4.209%	3.865%	0.3611	0.3711
Multinomial NB	3.535%	3.232%	0.07060	0.3378
Linear SVM	2.467%	2.435%	0.5135	0.5374

Table II. : Results with bag of words representation

Tag	Text + Meta	Meta
c#	6.986%	7.064%
java	4.274%	4.294%
php	3.38%	3.574%
javascript	4.219%	4.264%
android	1.162%	1.864%

Table III. : Error Rate for top 5 tags using Linear SVM with tf-idf

	Error		F_1 Score	
Features →	Text Only	Text + Meta	Text Only	Text + Meta
Perceptron	4.050%	**3.2177%**	0.3659	0.3795
Multinomial NB	3.213%	**3.183%**	0.3624	0.3500
Linear SVM	2.448%	**2.422%**	0.5358	0.5448

Table IV. : Results with tf-idf

The use of meta features(hasCode, hasURL, codeLanguage) showed a slight improvement in error rates – SVM (+0.026%), MNB(+0.03%) and Perceptron(+0.02%) for both tf-idf and bag-of-words. This is evident even more if we compare error rates for individual tags as seen in table III.

Perceptron performs worse than Multinomial NB maybe because the training data isn't linearly separable and contains noise, therefore perceptron can't find a clear separation. The bag-of-words representation doesn't take into account the relation between words, therefore treats them as independent. This could be a possible reason for Multinomial NB being able to achieve good results. Linear SVM outperforms Multinomial NB because of the very high dimensionality of the feature space. It calculates the best separating hyperplane and gets the best result.

But high accuracy comes with a time-tradeoff. Linear SVM has the highest training time of 15 minutes. Multinomial NB has a training time of 12 minutes for the same training set. This small difference in training time becomes significant if we consider the entire training data of 6 million samples and predicting top 200 tags – SVM would take 6 hours longer to train.

I ended up not using stemming since it gave a worse error rate during cross validation. This is probably because some words lose their feature importance for distinguishing a class when reduced to their base word.

To scale this model to the entire dataset of 6 million training set and 42000 tags would require overcoming more challenges. Since the training set would be too big to fit in memory, training will have to be done in batches, by loading chunks of the training data that would fit in the memory. Linear SVM using stochastic gradient descent would be a good model for that size of data.

9. CONCLUSION

Tag recommendation on a large dataset is an open ended problem. I tried three different machine learning algorithms and evaluated their performance under different feature selection. Linear SVM using tf-idf representation with meta features gave the best performance with an accuracy of 97.58% and F_1 score of 0.5448. The state-of-the-art model (#1 on Kaggle contest's public leaderboard) has a F_1 score of 0.81, which is much better than my model. Training on the complete 6 million examples would probably help reduce this gap, but there is room for more improvement.

10. FUTURE WORK

If I had more time, I would have experimented with assigning different weights for words in the title and body. I could have also tried using LDA to select a smaller set of useful features. I could have tried working on a larger dataset with Linear SVM + SGD to build a scalable solution. In the future I would try to implement a multi-stage graph based algorithm that takes into account title $\leftrightarrow$ tag and tag $\leftrightarrow$ tag relationships. During my research, I also came across the Vowpal Wabbit algorithm, which is well suited to this kind of task. VW has an algorithm that shows the relative importance of words in a documents corpus. This would be very useful to select good features.

REFERENCES

FB-Kaggle. 2013. Kaggle - Keyword Extraction. (2013). Retrieved February 23, 2015 from https://www.kaggle.com/c/facebook-recruiting-iii-keyword-extraction/data

Darren Kuo. 2011. *On Word Prediction Methods*. Technical Report UCB/EECS-2011-147. EECS Department, University of California, Berkeley. http://www.eecs.berkeley.edu/Pubs/TechRpts/2011/EECS-2011-147.html

Marek Lipczak, Yeming Hu, Yael Kollet, and Evangelos Milios. 2009. Tag sources for recommendation in collaborative tagging systems. *ECML PKDD discovery challenge* 497 (2009), 157–172.

Edward Loper and Steven Bird. 2002. NLTK: The Natural Language Toolkit. In *Proceedings of the ACL-02 Workshop on Effective Tools and Methodologies for Teaching Natural Language Processing and Computational Linguistics - Volume 1 (ETMTNLP '02)*. Association for Computational Linguistics, Stroudsburg, PA, USA, 63–70. DOI:http://dx.doi.org/10.3115/1118108.1118117

Leonard Richardson. 2009. BeautifulSoup - A program designed for screen-scraping HTML. (2009). Retrieved February 23, 2015 from http://www.crummy.com/software/BeautifulSoup/

Clayton Stanley and Michael D Byrne. 2013. Predicting tags for stackoverflow posts. In *Proceedings of ICCM*, Vol. 2013.