Predicting Tags for StackOverflow Questions

Sebastian Schuster **Wanying Zhu** **Yiying Cheng**
{sebschu,wanyingz,yiycheng}@stanford.edu

Abstract

We present a system that is able to automatically assign tags to questions from the question-answering site StackOverflow. Our system consists of a programming language detection system and a SVM using content-based features. When testing on an unseen test set, we achieve a mean F1 of 0.41 on this task.

1 Introduction

The question-answering site StackOverflow allows users to assign tags to questions in order to make them easier for other people to find. Further experts on a certain topic can subscribe to tags to receive digests of new questions for which they might have an answer. Therefore it is both in the interest of the original poster and in the interest of people who are interested in the answer that a question gets assigned appropriate tags.

StackOverflow allows users to manually assign between one and five tags to a posting. Users are encouraged to use existing tags that are suggested by typing the first letter(s) of a tag but they are also allowed to create new ones, so the set of possible tags is infinite. While the manual tagging by users generally works well for experienced users, it can be challenging for inexperienced users to find appropriate tags for their question and by letting users add new tags it is likely that different users use different orthographic versions of tags that mean the same thing such as "php5" and "php-5". For these reasons it is desirable to a have a system that is able to either automatically tag questions or to suggest relevant tags to a user based on the question content.

In this project we are developing a predictor that is able to assign tags based on the content of a question. More formally, given a question q containing a title consisting of n words $a_1, ..., a_n$ and a body consisting of m words $b_1, ..., b_m$, we want to assign $1 \leq k \leq 5$ tags $t_1, ..., t_k$ from a limited list of tags T.

2 Related Work

2.1 Automatic Content Tagging

There has been some work done on automatic tagging of web content. Mei et al. (Mei and Zhang, 2008) trained language models using web document content and tag logs to suggest tags to users of the social bookmarking service del.icio.us. Sood et al. (Sood et al., 2007) developed a tag suggestion system for blog posts using information retrieval methods that was based on finding similar blog posts and suggesting some of their tags. Liu et al. (Liu et al., 2011) modified the IBM Model 1 word alignment algorithm that is predominantly used to train machine translation systems to generate alignments between short descriptions and texts and used this information to suggest new tags. Stanley et al. (Stanley and Byrne, 2013) propose a Bayesian probabilistic model to predict tags for StackOverflow posts. For each tag they compute an activation score given the words in the title and the body that mainly depends on co-occurrence statistics.

2.2 Programming Language Detection

Very little work has been done so far on the automatic detection of programming languages. GitHub developed a library[1] that is able to detect programming languages but it mainly relies on the file extension and only uses a probabilistic model for ambiguous cases such as files ending in ".h" that could be C, C++ or Objective-C header files. Instead of using a multi-class Naive Bayes Classifier, Klein et al. (Klein et al., 2011) collect some specific statistics of code in different languages such as statistics on certain keywords or punctu-

[1] https://github.com/github/linguist

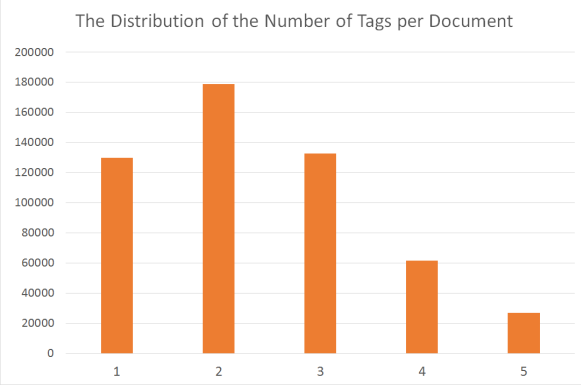

Figure 1: Distribution of the number of tags per document

ation marks to predict the used programming language.

3 Data

We use the data provided for the Kaggle competition on the automatic tagging of StackOverflow posts[2]. The data set consists of 6,034,196 StackOverflow questions. Each question consists of a title, the HTML markup of the question body and the tags of the question. As working with such a huge data set entails many computational limitations, we decided to use only a subset of the data. First, we reduced the tag space by only considering documents tagged with the most 1,000 frequent tags. From this data, we sampled 529,588 documents for our training set, 1,000 documents for our development set and 5,321 documents for our final evaluation set.

Table 1 shows the most frequent tags in our corpus. Here we can see that the most common tags are almost all names of programming languages. Figure 1 shows the distribution of the number of tags per post. Here we can see that more than 50% of the posts have either two or three tags.

c#, java, php, javascript, android, jquery, c++, python, iphone, asp.net, mysql, html, .net, ios, objective-c, sql, css, linux, ruby-on-rails, windows

Table 1: 20 most frequent tags

[2]http://www.kaggle.com/c/facebook-recruiting-iii-keyword-extraction

4 Methods

4.1 Evaluation

We evaluate our system by computing the mean recall, mean precision and mean F1 over all documents, i.e. we compute recall, precision and F1 for every document and then take the mean of all these values.

4.2 Baseline

As a baseline, we implemented a Naive Bayes classifier trained on the words in the title of a question. We tokenize the title using the sentiment-aware tokenizer by Christopher Potts[3] that is well suited for tokenizing web text with a lot of punctuation. For new questions, we assign the 3 (the median number of tags per question) most probable tags. By doing that we get a precision of 0.33 and a recall of 0.45.

4.3 Programming Language Detection

Motivated by the observation that a lot of posts contain code snippets and that the programming language is often used as a tag, we implemented a programming language detection system. We identified a list of 28 very common tags referring to programming languages. Then, we went through our training data and considered all questions with exactly one of these tags. Code snippets are encapsulated in specific HTML-markup so it is easy to extract only code snippets from a question. We tokenize these snippets by splitting on a character level for all non-alphabetic strings and on a word level for alphabetic strings, so the string "int count=0" would be split into "int", "count", "=", "0". Then we filter out all tokens that occur less than 20 times in our corpus based on the assumption that these are variable or function names or parts of comments that contain little information on which programming language was used. The remaining tokens are then used to train a multinomial Naive Bayes classifier that can return the most probable programming language given a code snippet.

4.4 Content-Based Classifier

As a second system, we implemented a SVM based classifier. We treat our problem as a binary classification problem that predicts for each tag $t \in T$ whether it is a tag for the given document

[3]http://sentiment.christopherpotts.net/

or not. We construct our training examples the following way: For each observed tag-document pair, we compute a feature vector and use these examples as positive training examples. Additionally, we sample for each document x random tags, where x is the number of tags assigned to the document and compute again a feature vector for each of these tag-document pairs and use them as negative training examples. By doing that we get the same number of positive and negative training examples. These examples are then used to train a linear SVM classifier[4]. To assign tags to a new document, we go through all tags and for each tag we compute the tag-document feature vector and let our classifier determine whether the given tag is a tag for the document or not. In case more than N tags get assigned to a document, we assign the N ones having the highest decision function value. We treat N as one of our hyperparameters.

We preprocess all documents by 1) stripping the code fragments and any HTML code from the question body and 2) by tokenizing the title and body. In order to tokenize we use a modified version of the sentiment-aware tokenizer that does not split frequent tags such as *c#*, *c++* or *.net*.

We implemented the following six features for a document-tag pair:

- **Exact Title**: Is the tag one of the words in the title.
- **Exact Body**: Is the tag one of the words in the body.
- **Relaxed Title** and **Relaxed Body**: Are all tokens that are obtained by splitting tags at hyphens contained in the title resp. body. (e.g. the tag *machine-learning* would be contained in *Which machine learning algorithm should I use?*)
- **Title PMI** and **Body PMI**: We loop over all words in the title resp. body and compute the sum over all pointwise mutual information (PMI) values for a word-tag pair. PMI is defined as following:

$$PMI(t, w) = \log \frac{p(t, w)}{p(t)p(w)}$$

We estimate the probabilities needed to compute the PMI by computing the maximum likelihood estimates of the training data. In case we did not observe a certain document-tag pair, we set its PMI to 0.

[4]We use the LinearSVC implementation in scikit-learn.

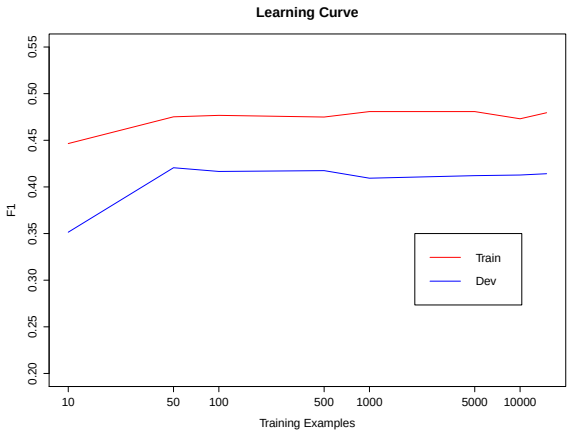

Figure 2: Learning Curve on training and development set.

4.5 Hybrid System

Finally, we combine the two described systems to create a hybrid system. For the hybrid system we predict the programming languages of all code snippets in a question (if there are some present) and up to N tags using the SVM described above. The output of the hybrid system is then the union of the two tag sets that were output by the respective systems.

5 Results

5.1 Training Set Size

Given that we have a model with only six features, we assumed that we need only a small fraction of the training data to obtain a good model. Figure 2 shows the learning curve of our model. We can see that our assumption was right and by using only 50 training examples we already get our highest F1-score on the development set. For this reason we use only 50 training examples for all further experiments. Note that while we use only 50 training examples to train our SVM, we still use the entire training set to train our code classifier and to estimate the PMI values.

5.2 Maximum Number of Tags

We observed that in many cases our predictor predicted too many tags. As described above, we use the N tags having the highest decision function value. Figure 3 shows the influence of varying N on the mean F1 on the development set. As we can

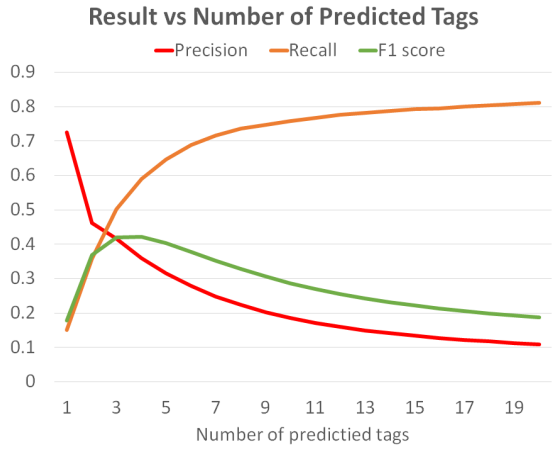

Figure 3: Influence of the maximum number of predicted tags on the performance of our model on the development set.

see in the figure, we get the best results by setting N to 3, why we used this parameter for all further experiments.

5.3 Feature Selection

In order to check whether all our features actually improve our system, we did an ablation test on the development set. We noted that leaving out any feature lowered the F1 score. For this reason we included all our features in our final model. For the sake of brevity, we omitted the exact results of the ablation test, but it showed that especially the PMI Title feature was very important as leaving it out led to a drop of 0.04 of the F1 score (0.381 vs. 0.42).

5.4 Final Results

Figure 4 shows the results of all our models on the held-out test set. We can see that in terms of precision all our models outperform the baseline. We can also see that the programming language predictor is able to predict a tag with a remarkably high precision. In terms of recall our SVM-based model and our hybrid model outperform the baseline and in terms of F1 we see a clear improvement of using our SVM based and an even further improvement using our hybrid system. Our test results are also comparable to the results on our development set suggesting that we chose reasonable values for the hyper parameters.

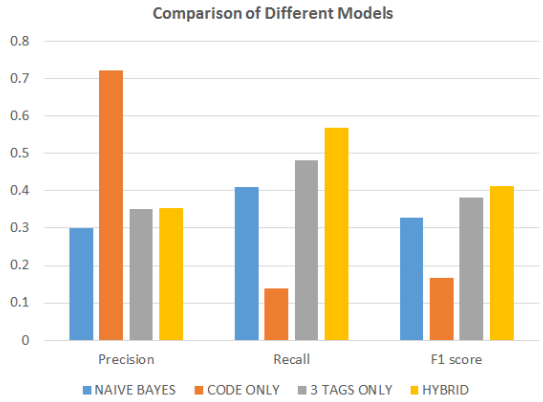

Figure 4: Results of the different models on the held-out test set.

6 Discussion

We presented a fairly simple model that is able to predict StackOverflow Tags. While a F1 of around 0.4 indicates that there is still a lot of room for improvement, our hybrid system clearly outperforms our Naive Bayes baseline and also the solely content-based or the solely code-based systems. One very nice property of our model is that it requires very little training data compared to classical text classification models and we only need one model compared to a model for every tag if we did a one-vs-all classification for all tags. However, one major drawback we noted is that we need to compute a feature vector for each document-tag pair, so given our tag set size of 1,000 we have to compute 1,000 feature vectors to make one prediction which turns out to be very slow compared to one feature vector needed for a classical one-vs-all text classification model that uses the same bag-of-words or tf-idf feature vector for all 1,000 classifications. Nevertheless, we only use 6 features and a very simple model which at least results in very fast predictions once we computed all the feature vectors.

By varying the number of predicted tags as shown in Figure 3 we also showed that by only predicting 10 tags for each question we achieve a recall of around 0.8. Considering that one possible application of our system is providing tag suggestions to a user, this indicates that our model would be very well suited for that task as choosing between 10 suggestions seems to be easily accomplishable by users.

Compared to the current leaderboard on the Kaggle competition, our results seem to be very

bad, as other teams seem to achieve F1 scores beyond 0.75. However, we noted that around 75% of the test data are also contained in the training data so one could easily achieve a mean F1 of 0.75 by just looking up the questions in the training data. As we split the original training data into non-overlapping train, development and test sets, our numbers should give a better estimate on how well such a model actually performs on unseen data.

6.1 Error Analysis

Given our results, it is clear that there are still errors that remain. We identified the following three common error categories.

- Our system predicts too many/too few tags:

  This problem is caused by the fact that our SVM tends to predict too many tags and we use a cut-off for the maximum number of tags that get predicted. As a user can choose between 3 and 5 tags and we typically predict 3 tags and potential programming-languages, we only get that right in the majority of the cases, but not if the user specified less or more tags. An alternative to using a fixed number would be to predict tags whose decision function value is above a certain threshold, but by examining some examples, there does not seem to be a threshold that would give an improvement over predicting a fixed number of tags.

- Predicted tag is contained in the title or body but is not a real tag to the question:

  In some cases we observed that the tag is contained in the question title or body but the user did not assign that tag. This error seems very hard to fix, as generally this feature is a good indicator for a tag-document relation. However, it seems that in many cases the wrongly assigned tag is actually a valid tag for the document and it is just the case that the user did not use it. Thus it is still a valid prediction although it does not conform to the gold standard.

- The detected programming language is not a real tag:

  This error occurs quite frequently, as people often use more specific tags such as *jquery* compared to *javascript*. However, if we assume that the number of tags is not limited, then assigning this additional tag should not harm the user experience as if the used programming language is really *javascript*, then this seems to be an appropriate tag.

6.2 Conclusion and Outlook

In this paper we described a simple classifier that has the capability of predicting tags to StackOverflow questions given only the question title and body. Besides engineering more sophisticated features, future work should focus on optimizing the runtime of our model, as it is currently too slow to be used in practice. We also assume that incorporating more data to estimate the PMI values, could further improve recall as it seems that our model heavily relies on that feature. Further, one might want to investigate whether co-occurence statistics of tags could further improve the model, as there are a lot of tag pairs such as *javascript* and *jquery* that co-occur very often.

References

David Klein, Kyle Murray, and Simon Weber. 2011. Algorithmic programming language identification. *CoRR*, abs/1106.4064.

Zhiyuan Liu, Xinxiong Chen, and Maosong Sun. 2011. A simple word trigger method for social tag suggestion. In *Proceedings of the Conference on Empirical Methods in Natural Language Processing*, pages 1577–1588. Association for Computational Linguistics.

Qiaozhu Mei and Yi Zhang. 2008. Automatic web tagging and person tagging using language models. In *Advanced Data Mining and Applications*, pages 741–748. Springer.

Sanjay Sood, Sara Owsley, Kristian J Hammond, and Larry Birnbaum. 2007. Tagassist: Automatic tag suggestion for blog posts. In *ICWSM*.

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