Predicting Tags for Stack-overflow Questions

By
Asish Gaddipati – Chinmaya Naguri

Introduction

In Stack-overflow website while a user asks a question, one will be given an option to choose utmost 5 tags regarding what he is asking. User has to select these tags manually. Tags helps the website to show appropriate questions to those who are interested in that particular topic *[Tags define a topic of interest. For E.g. java – about java language, multithreading – about multithreading independent of the language one is using etc.]*, thus targeting the right people who could answer it. As the user is in control of the tags, he may choose tags which may or may not be relevant to a question. New users may not know how to tag a question to reach maximum audience or sometimes a user may choose a tag which is relevant but there could be a tag which is more appropriate which he didn't know existed *[There are nearly 40K tags in Stack-overflow]*. So, to help a user in making his question to reach wide & appropriate audience and to increase the probability of getting it answered we can predict tags while a user is writing it. Thus making the user aware of tags available and giving him a chance to select could help the website community in total. Our project is trying to address this issue by predicting the tags for a given question.

Our approach has two models. One is to use a SVM with Quadratic Kernel (a supervised learning method) and other to use Clustering (an unsupervised learning approach).

Problem Definition and Algorithms

Task Definition: If given a question which is about to be posted our program should predict tags which are relevant.

Dataset: First we have to get the dataset for training the models. Stack-overflow does not give any API to get its questions and answers. So, we wrote a java program using Jaunt API – A Web Scrapping & Automation API. This program grabs the questions, associated tags and saves them to a Database. We have to train our models for each tag, so that it can decide how close or related a question is to a particular tag. So, we need a set of questions related to each tag in our database for learning. Therefore we need a list of tags which we are interested in. So, we grab the tags from http://stackoverflow.com/tags.

For this project we have taken 144 top used tags. Now, for each tag we grabbed 210 questions for training. So, in total we have got over 30K questions. As for every question we download there will be at-least a tag to utmost 5 tags. Thus on a whole we have got 7900 unique tags. We get these questions from the below URL replaced for every important tag we noted.
http://stackoverflow.com/questions/tagged/<tag_name>. We go recursively over this page till we get 210 questions. While storing the question information we will remove words which are frequently used in English literature using a Stop words List. We had to induce a time gap of 740ms in between every request to the website or else our IP gets blocked by their server. So, it takes almost 6-7 hours to just acquire the data. With this huge database we can train models for tags which has atleast 20/30/150 questions depending on our experiment.

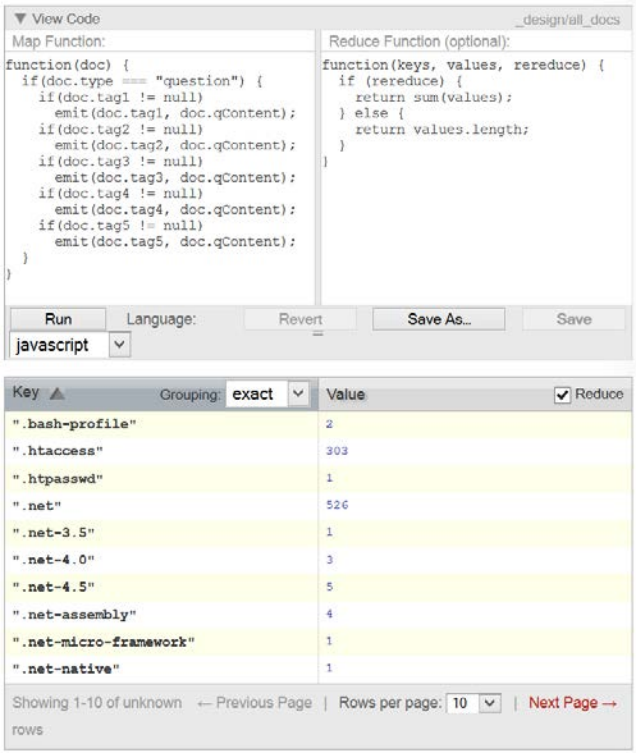
Using same method we have downloaded more 30K questions in to a different database for testing.

Database: As our data is mostly text which are documents, we chose *CouchDB* - a database that uses JSON for documents storage. This is a No-SQL database. We chose this as this has a feature called replication which synchronizes with any other CouchDB and maintains data integrity between databases. This is useful for us to work on data using multiple systems and store its results into a local database, which replicates itself onto a master database, thus making the master database an aggregation of results from all the systems used. This makes learning & testing process much faster. A typical DB entry looks like this:

Field	Value
_id	"9c591c5a7c9341b39fc87bb499ac4de8"
_rev	"1-d3620f759a43aee355b00a0709d85876"
qContent	"10 documents collection now want update documents count means updating documents will look like know update set method don know use update d..."
tag1	"python"
tag2	"mongodb"
tag3	"collections"
tag4	"documents"
type	"question"

We can specify the contents of JSON depending on our usage. There is no strict restriction or table structures to follow. So, we have used this DB to store all different kinds of data, like question information, feature vectors, feature word IDF values, results information (predicted tags) etc.

We can retrieve data from the database using MapReduce JavaScript functions stored in the database. We wrote many Map functions for different purposes like retrieving all the questions, questions vs tags, IDF value list etc. A typical MapReduce function and result is as below:



We used *LightCouch API* to interact with the database from Java programs. This gave us an easy way to save, retrieve, run MapReduce functions & update database entries.

Feature & Feature Vectors: From the data we have to extract features which should be useful for prediction. We have selected words as features but limiting them by a threshold on its occurrences. For example, all the words from the questions which has occurred atleast *N* times are our features. N could be 6, 20, 30 etc. As the questions asked may include code blocks, we added these code words with *'@'* in the beginning and ending of each code word. I.e. *@class@, @function@*. This makes easy to distinguish between literature words & code words. Thus formulating the feature vector both dependent on question description as well as the code.

In both of our methods we are using TF-IDF feature vector methodology. TF means term frequency i.e. it tells us no. of times a word has occurred in a given document and IDF means Inverse document frequency i.e. it tells us no. of documents contain a given word. Different forms of Tf-Idf can be seen in the Wikipedia page [5].

We have used log normalization for TF, $log(1+f_{t,d})$ and inverse frequency for IDF, $log(N/n_t)$.

TF	IDF
$log(1+f_{t,d})$	$log(N/n_t)$
$f_{t,d}$ - frequency of term 't' in document 'd'	n_t – no. of documents having term 't'. N – Total no. of docs

For SVM model each document is a question both in training and testing time. In Clustering method a document is a question in test time but during training, the aggregation of all the questions for a particular tag is a single document. This is because we have to generate a vector for each tag – a cluster.

Tag Selection: As mentioned above we have nearly 8000 unique tags in our database. So, for making a good prediction, we need ample set of questions for each tag. If we say we will learn for tags which have atleast 20 questions then we will have 480 tags. Or say atleast 100 questions then we get 150 tags. In our tests we chose different values for comparison.

After selecting the feature words & final tags for learning, we can start learning on the given dataset using following methods:

Unsupervised Learning: *(Clustering with Cosine similarity for comparison)* Now, we have features to formulate a feature vector & we have tags to form different clusters. Each cluster's data is the aggregation of the questions of a particular tag. Now, we train our model to build a feature vector for each Cluster/Tag:

1. Compute IDF values for all the features, assuming each cluster as a single document.
2. *For Each Cluster*: compute Tf values for each feature and simultaneously multiply its corresponding IDF value for all the features, i.e. tf * idf value.
   E.g: Java Cluster:
   Feature Vector: *[@throw@, @contrib@, threads, timer, flow, @methods@, ….]*
   Tf-Idf vector: *[5.7071*0.4056, 2.4512*4.3175, ….]*
3. After formulating all these vectors save them into DB for future retrieval.
4. In Testing, we given a question we will compute the feature vector of it. *(We already have IDF values for all the features from learning, now compute Tf values for the features from the given document and multiply accordingly to form the vector).*
5. Using this feature vector of the question we compute the Cosine similarity between all the tags.
6. As we want to predict top 5 tags we save the top 5 highest similarity valued tags.

SVM: Training in SVM is similar to the above method. We have features list and tags list already computed. Here we train our models for each Tag using binary classification. i.e. all the training examples which belong to a particular tag are given +1 and -1 otherwise. Similarly we train models for all the tags.

1. Compute IDF value for all the features and save them for future.
2. Now, compute and save the feature vector for each question. i.e. compute Tf values for each feature and simultaneously multiply its corresponding IDF value for all the features.
3. We are using SVMLight for computation with Quadratic Kernel having parameters: Type of Kernel, *t=1*; Order of Polynomial, *d=2*; *s=1* & bias, *c=1*. So, we need to create sparse data files for each tag for training.
4. Now, we train the SVM using these input files created in step 3 & generate corresponding .model files.
5. For testing we repeat step 2 on test data to get feature vectors for the test examples. Then create files accordingly as in step 3.
6. If we train & test 100 tags we get 100 result files. Out of them each line is a prediction value for each question. i.e. 1st line in all the result files is the predicted value for 1st question by each model. As we want to predict top 5 tags we save the 5 highest valued tags for each question.

Related Work

1. One of the challenges in ECML PKDD Discovery 2009 was to recommend online tags (but not on Stackoverflow website). The best solution is presented by Lipczak [6], where they used a 3 stage tag prediction with the second and third stages consists of *title <-> tag* & *tag <-> tag* relationships using graphs. Tags are recommended after combining the scores from each stage.
2. Stanley & Byrne, 2013[7] proposed a Bayesian probabilistic model to recommend tags for Stackoverflow questions. They computed an activation score depending on the words in the title and body which in turn depends on the co-occurrence statistics.

Experimental Evaluation

Methodology: To understand how well our models predicted we need some numbers. We suggest or predict 5 tags for every test question. But, a user has the liberty to choose any number of tags from 1 to 5 while posting the question. From these two sets i.e. true tags & predicted tags for each question in the test dataset we express accuracies on the whole as below:
a_n = No. of questions with atleast *n* tags predicted correctly.

b_n = No. of questions with atleast *n* true tags from the test dataset.
Now, Accuracy for atleast *N* tags predicted correctly is 100 * a_n/b_n where *n* = 1 to 5.
For example, if *n* = 2:
a_2 = No of questions with atleast 2 tags predicted correct. I.e. it includes 2, 3, 4, 5 tags predicted correctly = 9314
b_2 = No of questions with atleast 2 system tags in the test dataset. I.e. it includes questions which have 2, 3, 4, 5 tags = 29075
So, Accuracy for Atleast 2 tags predicted correctly = a_2/b_2 = 100 * 9314/29075 = 32.03%

Results: Our test dataset has 30159 questions in total. Table below shows the distribution of questions to count of tags.

Tags Count	1	2	3	4	5
Question Count	1084	5022	8753	8111	7189

We tested our models with different feature vector lengths and cutting short tags (depending on the number of questions a tag has in the training data).

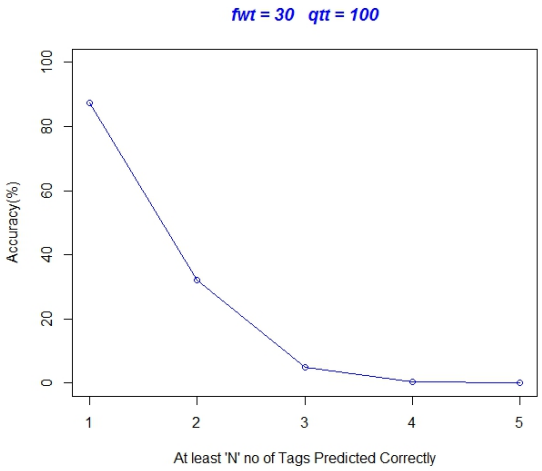
Feature Vector Length: Selection of features can vary depending on the threshold we keep on the occurrences of words in the whole data set.
i.e. *fwt* = 6 or 30 or 100 etc.
(*fwt* = feature word threshold)
Following table shows feature vector length for different threshold word occurrences (*fwt).*

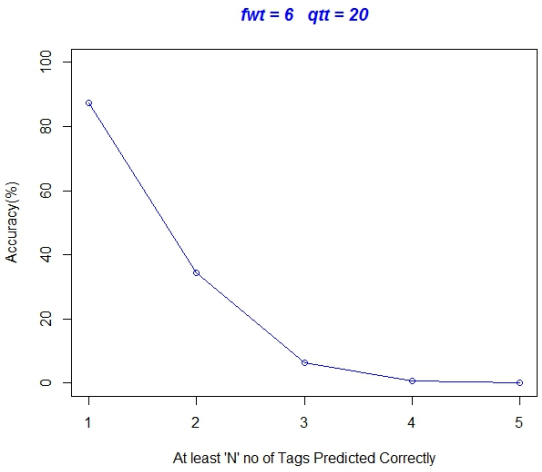
Word Threshold	6	15	30	50	100
Features Count	39989	18493	11074	7710	4741

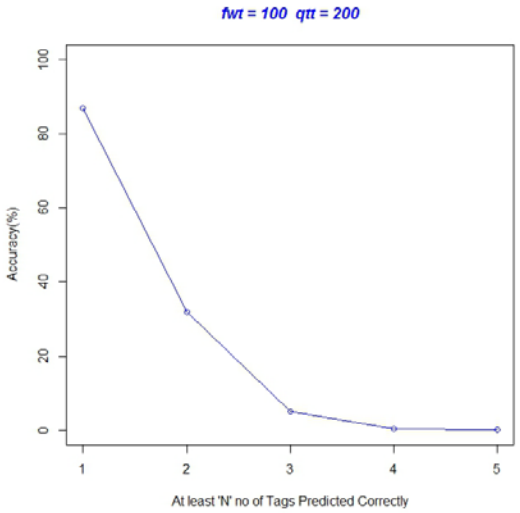
Tags Selection for learning: As mentioned in the dataset description our database has many tags but we have to choose only few tags which have ample questions for training that particular tag. i.e. say, a tag should have atleast 50 questions to qualify for learning by the model. i.e. *qtt* = 20 or 100 or 200
(*qtt* = questions threshold for tags)

Min. Questions	20	100	200
Tags Count	476	150	144

SVM Model Results: Following graphs show the percentage accuracies for no. of tags. Each graph is a different test, i.e. with different feature words count & tags threshold.







SVM model Analysis: From the graphs we can see that our SVM model has predicted with good accuracy (~87%) for atleast one tag. i.e. atleast one predicted tag is present in the actual tags. The accuracy value for 2 is 32%. There after the values are less. The fall in accuracy levels is based on our observation that the predicted tags are closely related to actual tags but they differ in version numbers or contextual names. For example html – html5, C – C99, C# - C#-4.0 are considered different by our current algorithm.

For now there is no way to check the relation between the tags. So, if we can formulate a tag tree structure to understand which tags are more closely related or to see which tag pairs occurs more often, we could easily predict tags which make more sense. It would be even helpful if we estimate tags based on the prior predicted tags (i.e. using 1st, 2nd predicted tags to predict the remaining tags) could lead to more satisfactory results.

We can observe from the graphs that even after changing the feature vectors length (i.e. using more prominent features) there is not much of difference in prediction accuracies (very less ~ 1%). This is a known SVMs behavior, that it can handle features which are not relevant to the classification.

Unsupervised Learning Results: Accuracy for this model is very poor. It is not detecting atleast 1 correct tag for any question. Tables below gives a glimpse into this for different tests, i.e change in feature words frequency and tags question threshold:

fwt = 15 qtt = 100

Atleast '*N*' tags predicted	Predicted Qs Count	Accuracy (%)
1	12	0.04
2/3/4/5	0	0

fwt = 30 qtt = 100

Atleast '*N*' tags predicted	Predicted Qs Count	Accuracy (%)
1	11	0.03647
2/3/4/5	0	0

fwt = 100 qtt = 200

Atleast '*N*' tags predicted	Predicted Qs Count	Accuracy (%)
1	20	0.06632
2/3/4/5	0	0

Unsupervised Analysis: The most pressing reason that we could think of, why unsupervised method did not yield good results is 'irrelevant features'. When we take a look into the features after selection there are still many features which are irrelevant. Most of these are generalized words and don't constitute for classification. Eg: compile, converted, means, @iwc@, @cart@ etc. Due to these features the cosine similarity between vectors resulting in higher value for unrelated clusters/tags. Whereas SVM results in good accuracies due to the fact that SVM can handle irrelevant features.

And second important reason is that in some cases the predicted tags are in fact related to the question and closely related to the true tags as well. As our comparisons are precise tag comparisons we can't see them in the results. For example a question has tags – Web-development, html. But one of our tag prediction has CSS. So, we can't see this in a tag to tag comparison unless we have some related tags datasets or some tag based tree to show the estimate the relation in between the tags. We have mentioned this in Future works section as well.

To improve the performance of this model we have to choose some robust feature selection technique. Along with this if we combine dependent and related tag identification to choose and evaluate tags, then we could see some fruitful results.

Future Works

Our results are really poor. We understood that there is lot to tune in different aspects of our methodology to increase the accuracy. Following are the ones we thought about:

- Create Tag trees. I.e. Tags which are related to each other and occurs frequently. E.g.: HTML - CSS – PHP – JavaScript – JQuery etc. Java – Swing – AWT – Hibernate etc.

  Now we can use the top two tags to predict other three tags from the tag tree structure we generated.
- It is clear from our results that the features we have selected are not good enough for prediction. Feature Selection should be more robust than only considering a threshold occurrences of a word. We can observe some keywords for each tag and give them more weight in the vector (like multiplying its tf-idf value by K-times etc.).
- We have observed some features like *sq, dummy, showing, enough* etc., are in the features even after using Stop Words & Occurrences Threshold. Should include more words like these into Stop words or formulate some mechanism to eliminate such words.
- Could use a separate language prediction algorithm for the code blocks in given questions. Thus its child tags or related tags can be give more weightage in prediction.
- Words from the post title & post description are giving same importance. From our observation post title can describe a lot about the question type & info. So, words from title could be given more weightage.

Conclusion

We conclude that Quadratic kernel SVM is able to predict very well compared to Clustering and Feature Engineering is an important aspect to yield better results with any learning method used. There is a lot to be done as discussed in this report to make the unsupervised learning method to yield good results.

Bibliography

1. Jaunt-API: Java Web Scraping & Automation - http://jaunt-api.com/
2. Stackoverflow – For Data Acquisition: http://stackoverflow.com/
3. CouchDB – NoSQL Database: http://couchdb.apache.org/
4. LightCouch - CouchDB Java API http://www.lightcouch.org/
5. Tf-Idf – Term frequency & Inverse Document Frequency. http://en.wikipedia.org/wiki/Tf%E2%80%93idf
6. Lipczak - ECML PKDD Discovery 2009 - (page 157) http://ceur-ws.org/Vol-497/proceedings.pdf
7. Stanley & Byrne - http://libflow.com/d/je2sdtys/Predicting_Tags_for_StackOverflow_Posts