A discriminative model approach for suggesting tags automatically for stack overflow questions

Authors: Avigit K. Saha, Ripon K. Saha, Kevin A. Schneider
Implemented By: Ankita Singh, Harsh Kundnani

GitHub: https://github.com/hkundnani/stack-overflow-auto-tag-suggest

Dataset: Stack Overflow PostgreSQL Dump

Language Used: Python

Libraries used for handling PostgreSQL Dump: psycopg2, pandas, sqlalchemy, datetime

Libraries Used for Processing and Modeling: Pandas, Numpy, Sklearn, NLTK, Collections

I. INTRODUCTION

Motivation: In Stack Overflow platform, all questions are associated with a few tags. Each question can have a minimum of 1 tag and maximum 5 tags. Tags can be user created (user need to have some reputation to create his own tag). When we tried to find a few relations as mentioned in the paper we got following results:

Proportions of Questions by Number of Tags:

Number of Tags	Number of Question	%
1	438,475	12.70
2	887,037	25.68
3	997,906	28.89
4	693,234	20.07
5	437,090	12.66

Table 1. Number of questions in each group (Groups- Number of tags)

The above table have numbers exactly same as in the original paper. With more than 87% of the questions having less than 5 tags.

Average views per question for the number of tags:

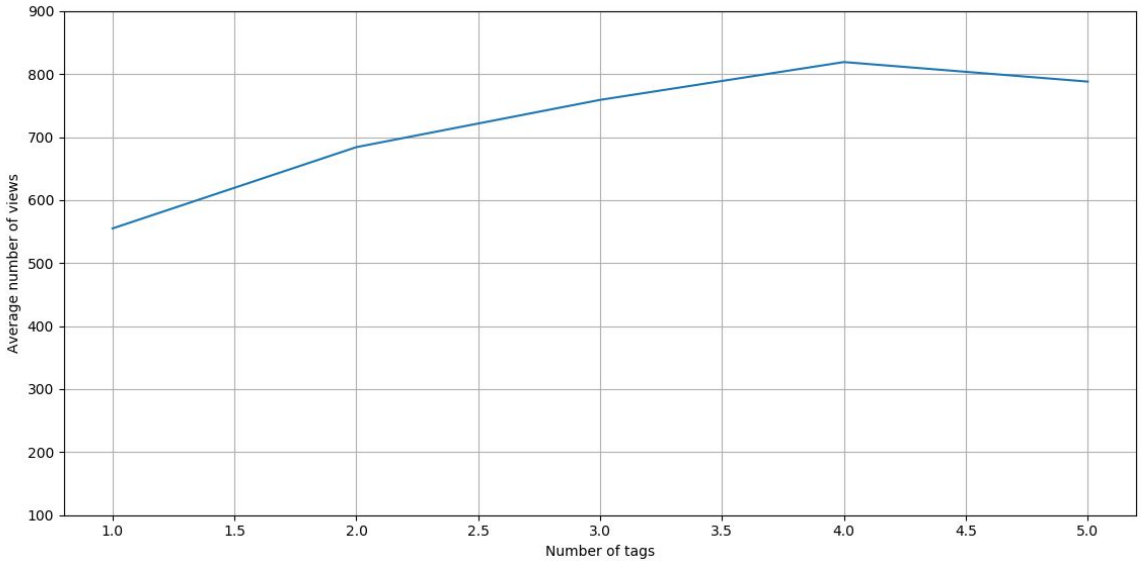

Fig 1. Our graph

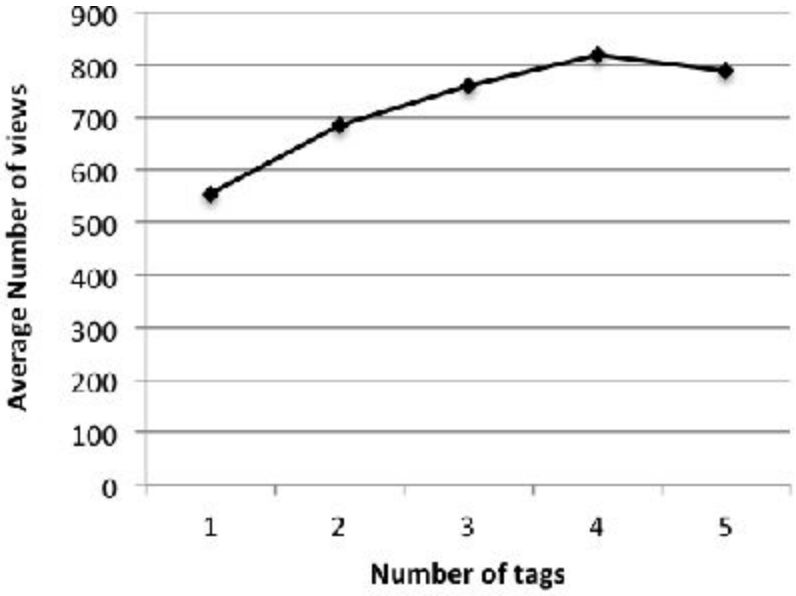

Fig 2. Original Graph

The above graph shows the relation between average number of views of a question for the number of tags. As can be observed from the graph, with increase in tags questions got more views.

So, we worked on implementing a model that automatically suggest question tags to help a questioner choose appropriate tags for eliciting a response.

We worked on following questions similar to the original paper:

1. How accurately do models work for each tag?
2. How well it can suggest the missing tags?
3. Can it suggest missing tags ?

II. METHODOLOGY

1. **Load Data:** The first step was to load the PostgreSQL data in a format which we can use for our project.

   For this we filtered the data in PostgreSQL to obtain only desired rows and columns.
   Rows: All rows with postypeId=1 (all questions)
   Columns for prior analysis: Question Id, Body, Title, Tags, Number of Answers, Number of Views
   Columns for further processing and making model: Question Id, Body, Title, Tags

   The obtained dataset is then converted to a .csv file for further use. The .csv file contained ~4 million questions.

2. **Finding Most Popular Tags:** The second step was to find most popular tags.

   Original Approach: Most popular tags are tags that appeared in more than 2000 questions.
   Total 834 such tags were found. 834 discriminative models were made (one per tag).

   Our Approach: 850 tags were obtained which appeared in more than 1500 questions. Wth the original criteria we were getting only 678 tags. To match the number of tags found in the original paper we changed the criteria from 2000 questions to 1500 questions.

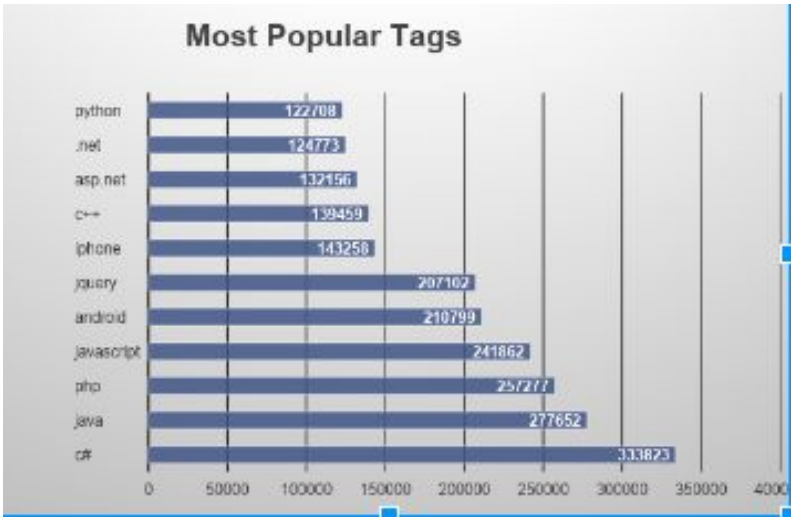

Fig 3. Top 11 tags with number of questions in increasing order.

3. **Creating Training/Test Dataset:** In a study [1] referred in the original paper it was observed that a discriminative model is best trained for a class that has 60 positive and 1500 negative documents.

   Training Dataset:
   For each tag we randomly selected 1500 negative questions i.e. questions that does not contain that tag and 60 positive questions i.e. questions containing that tag.

   For 850 tags we got ~1.3 million questions in our training dataset.

   Test Dataset:

   For each tag we randomly selected 10 negative and 10 positive questions.
   For 850 tags we got 17000 questions in our test dataset.

4. **Converting questions into vectors:**

   (i) Breaking questions into tokens
   (ii) Removing punctuations, URLs and other unwanted characters
   (iii) Removing stop words,
   (iv) Finding document term matrix:

   Original Approach: Used Mallet which is a java package for natural language processing and text mining.

Our Approach: Used CountVectorizer from sklearn library of python which creates a document by total vocabulary matrix and assigns term frequency in each cell.

Train Dataset Matrix (~1.3 million X ~5 million)

Test Dataset Matrix(17000 X 143130)

5. **Building discriminative Model:**

We built 850 classifiers for each tag. For this we used SVM as classifier using both linear and rbf model and got separate results.

Each classifier was a type of binary classifier predicting whether the question belong to a particular class (Class 1) or not (Class 0)

Original Approach: Used SVMLight for creating models.

Our Approach: Used SVM model implemented in python sklearn library.

6. **For proposing tags, we used the following algorithms as in the original paper**

Algorithm 1: Propose Probable Tags:

Input: Q(A new question) M: (All models)

Output: a list of suggested tags for Q

Body:

for each model m(from M) do

Similarity = predict(Q,m)

Add <m, similarity to list_of_tags>

end for

Sort list_of_tags in descending order of similarity

return sorted list_of_tags

Algorithm 2: Calculate similarity between Q and a model

Procedure: predict

Input: Q: a new question; m: a model

Output: max: similarity between Q and model m

Body:

Max =0

Vectors = featureVector(Q)

Similarity = SVMPredict(vectors, m)

Return MAX(max, similarity)

In the above algorithms, while testing, for each question similarity is obtained (Using algorithm 2) between model Q and question. This is achieved by testing the question against all 850 models and obtaining similarity for each of them. Then sorting the similarity between question and each model in descending order (using algorithm 1). The sorted list is then returned to the user.

III. RESULTS

The results obtained are as follows:

1. How accurately do models work for each tag?

   For each model accuracy is calculated. For example, if for a model related to particular tag, out of 20 questions (10 positive, 10 negative) 12 questions are predicted correctly then accuracy of that model is 60%. In this way 850 accuracies were obtained which is averaged to find the final accuracy.

Classifier	**Kernel**	**Training Dataset**	**Original Average Accuracy**	**Our Accuracy:**
850 binary SVM classifiers	Linear	60 positive 1500 negative questions	~68%	56%
850 binary SVM classifiers	rbf	750 positive and 1500 negative questions	-	53%

While using the original approach we got 56% accuracy in comparison to original ~68% accuracy. This accuracy does not conclude to a good implementation as it is almost similar to random guessing.

We tried a few different approaches like creating a single classifier using OnevsRest multi-classification approach where question belong to a particular class (Class= Tag) is given label 1 and for all other classes (tags) it is given label 0 and then model is trained.

This approach suffered from class imbalance problem as for each tag there were only 60 positive questions and all other questions (~1.3 million -60) did not belong to that class (tag).

This approach performed very poorly (~9%).

Other approach we took was to increase number of positive questions from 60 to 750 but this also performed bad (~53% accuracy)

2. How well can our system suggest tags for a given question?

   Below is a list of some questions with original and suggested tags.

   This shows that the a few tags out of all the suggested tags were really appropriate for the question.

Id	**Original tags**	**Suggested Tags**
9895904	Ios, ios5, xcode4	deployment tfs2010 clojure statistics full-text-search columns pyqt compiler-errors numpy 3d
3949060	windows, webkit, gtk, pygtk	gtk dialog bitmap deployment tfs2010 clojure statistics serialization animation outlook

1079809	sql, oracle10g	gtk oracle10g bitmap deployment tfs2010 graph uml views books animation
3460363	objective-c, iphone, ios-simulator, try-catch, nsexception	oracle10g bitmap deployment tfs2010 paypal statistics input ios-simulator books event-handling
8455168	flash, animation, animate	gtk bitmap tfs2010 uml associations clojure icons input animation casting

3. Can our model suggest missing tags?

   Though the models did not predict the exact tag, but it gave related suggestions for questions. A few examples are in the table below.

Question Id	Original Tags	Missing Tags
4520865	Algorithm, math	Input, statistics, clojure
10313601	Android, android-listview	Gtk, bitmap, icons, columns
10866522	Webbrowser, webclient	Webkit, dialog, gtk
11500320	Svg, png	Views, input, gtk
5972478	Jquery, jquery-ui, jquery-datepicker	Treeview, icons

Table 4. Missing Tags

IV. CONCLUSION

1. As can be seen from the prior analysis tags are really important to get more views and responses for a given question.
2. The results obtained are very different from the original paper. We got ~56% average accuracy in comparison to original ~68% average accuracy
3. We also observed that our model worked for very specific tags like internet explorer but not for general tags like c#. This is mentioned by authors as well in the paper.
4. Our model did not give good accuracy, but after analyzing how well it suggests missing tags, we found that for specific tags it was able to give nearly related tags.

Why differences?

1. Since the selection of questions were random there is a possibility of having a completely different dataset or dataset with very few similarities. Since the models were trained on different dataset and different accuracies.
2. Tags obtained were different as we changed the criteria to get the most popular tags. So were training models for different tags which could be the possible reason of different accuracies.
3. Use of different tools and packages (Mallet vs CountVectorizer, SVMLight vs SVM) might have caused building of different models.

V. TEAM CONTRIBUTIONS

	Work Description	**Done By**	**Associated Files on GitHub**
1.	Load data from PostgreSQL dump to database and then to .csv file	Ankita	connection.py
2.	Generate the graph and table for proportions of questions by no. of tags	Harsh	data_model.py
3.	Finding most popular tags	Both: Ankita: written initial code to find most popular tags Harsh: fixed errors in the code to generate most popular tag list	most_popular_tags.py
4.	Generating Test/Train Dataset	Harsh	generate_test_data.py generate_train_data.py
5.	Preprocessing	Ankita	tokenize_data.py
6.	Creating Model	Both created together along with discussion. Initially Ankita wrote the code to generate the term frequency, then Harsh wrote the code for training and testing model. We tried different variations of this code later to test for different use cases like OnevsRest and more positive questions.	dtm.py
7.	Code to save models into files	Harsh	dtm.py
8.	Evaluating Model	Ankita	accuracy.py
9.	Readme file	Harsh	Readme
10.	Project Report	Ankita	project_report.pdf

References:

[1] J. Wang and B. D. Davison, “Explorations in tag suggestion and query expansion,” in Proceedings of the 2008 ACM workshop on Search in social media, ser. SSM ’08. New York, NY, USA: ACM, 2008, pp. 43–50. [Online]. Available: http://doi.acm.org/10.1145/1458583.1458592