Predicting Stack Overflow tags

Burghelea Zaharia
zaharia.burghelea@ulbsibiu.ro

Abstract

This article proposes to predict some of the most popular programming language tags from Stack Overflow and to observe how different parts of the text such as the title or various important words influence the prediction. The focus is on the Natural Language Processing side rather than on Machine Learning and it's classification algorithms, where mostly Support Vector Machines will be used. The NLP task consists of feature extraction and feature selection and aims to map the unstructured data which can be found in raw text on Stack Overflow to a structured form that can be fed into the classification algorithms.

1 Introduction

Since you are reading this most likely you already know what **Stack Overflow** is about. I tend to browse Stack Overflow and other Stack Exchange sites quite often and one thing that I am used to do instinctively is to look at the title and classify mentally to which tag a question would belong to, as most of the time this would work out, it's natural to assume that the title is perhaps more important than other parts of the text. A question that arises now is how does a computer behave regarding this and would it classify better if we would tell it to give more importance to the title or other relevant parts of the text. This paper aims to answer this question.

We will be working with the **StackSample** dataset from where we will extract the question (body + title) and the tags from those posts that contain some of the most popular programming language tags such as Python, Java, JavaScript, Ruby and so on.

Title	Body	Tag
How should I unit test a code-generator?	<p>This is a difficult and open-ended question...	c++ python unit-testing code-generation swig
Arrays of Arrays in Java	<p>This is a nasty one for me... I'm a PHP gu...	java php jsp tomcat
Acts-as-readable Rails plugin Issue	<p>I'm using Intridea's <a href="http://www.in...	ruby-on-rails ruby plugins
Notification of drop in drag-drop in Windows	<p>My C# program has a list of files that can ...	c# c++ windows winapi com

Figure 1: The extracted data from StackSample

The whole project can be structured into two parts: Preprocessing and Classification. In the former we will map the items (the question and it's title) to the vector space model and for the latter we will mainly use Support Vector Machines to predict the tags. In order to evaluate our results we will use two common metrics, namely the Accuracy and the F1 Score.

Because it was desired to implement everything from scratch, the programming language chosen was **Racket** (being my favourite), but in order to check the results with algorithms implemented in a standard library, most of the code was also translated into Python as an alternative. Also everything that is described bellow can be found **on GitHub**.

The rest of the paper is structured as follows: Section II describes our work related to the preprocessing part and how we brought the items to the vector space model, Section III explains how the classification was performed and Section IV discusses the results and draws the final conclusions

2 Text Preprocessing

2.1 Feature Extraction

In order to start, for every item we are going to concatenate into a string the title and the question body, then remove everything that can be found in the HTML code tags. The numbers alongside the punctuation will also be removed (however the "c++" and "c#" words will be kept separately as those 2 are quite relevant).

Next, the tokenization will take part, meaning that we will split the words by space, but also separate them taking in account camelCase, snake_case or kebab-case. Furthermore we will also count the frequency of each token and build up a global vocabulary which will be used later in order to convert the items into a vector: each word will have an unique id associated with it's frequency.

After tokenization there will remain a lot of words which aren't bringing any useful information, so in order to reduce this volume we will eliminate the stop-words (the most frequent words that appear in every post) and use stemming (bring the words to a morphological variant of the original one).

In this process we also give the option to increase the importance of the title (repeat it a couple times), to increase the frequency of the tags that appear as tokens (in this case the programming languages used) and to increase the frequency of some specific words that were manually selected (this consists mostly of libraries, IDEs, special words etc).

2.2 Feature Selection

Since there will still be many words left, we will compute a score for every token that represents their importance in order to reduce the number of words, or alternatively, we keep the relevant words so that the classification algorithms will have an easier time. For this, we will use Information Gain which gives a small value for tokens that appears in almost every class and a larger value for those that represents a specific class well.

Before we arrive at information gain we have to compute the entropy for

the entire dataset, namely:

$$\texttt{Entropy(D)} = -\sum_i p_i \log_2(p_i)$$

Here D represents the dataset, the sum will loop through every tag (i, which takes the value of each programming language) and p_i is the proportion of the items tagged as i from the dataset D.

The entropy represents the amount of information the possible outcomes give, this value attains a minimum at 0 if the items belongs to a single class. The maximum is obtained when each class got the exact same number of items and that values is $\log_2(|D|)$, where $|D|$ denotes the number of items in the dataset D.

After this, we want to know how much the entropy would decrease if we would group the items according to a given token (or feature). Mainly for every token we will compute the information gain:

$$\texttt{IG(D,A)} = \texttt{Entropy(D)} - \sum_s \frac{|D_s|}{|D|}\texttt{Entropy}(D_s)$$

Here beside what was already explained above, A represents an attribute (or token) and s represents a set of possible values for the attribute. Usually s takes every possible values for the token (so if there are 10 different frequencies for a specific token, then there will be 10 sets), however for our project we will consider only 3 possible sets. The first set would be made up with the tokens that their frequency is 0 and for the other two sets we will compute the average for the remaining token's frequencies and take whatever is bellow the average as a set and what is above the average as another set. After we computed the information gain for all tokens we will select only a percentage of tokens that gives the best information gain in order to be used later.

To make it more explicit, up till now for every item the tags were also appended, but in feature selection (and in what's to come for prediction) since we needed the vectors to have exactly one class we kept only one tag.

3 Classification

3.1 Normalization & Split

Furthermore, before we get to the classification algorithm, we are going to apply a sum one normalization for each token in every vector, namely:

$$TF(v,t) = \frac{n(v,t)}{\sum_i n(v,t_i)}$$

Here v represents a vector from the dataset and t is a token from that vector. The sum from the denominator sums up every frequency and basically the goal is to normalize each token so that their sum would be equal to one.

Also the dataset will be shuffled and splitted into two parts: 70% of the items will be used for training the used algorithm and the remaining 30% will be used for testing. As a remark, usually it's necessary to perform the normalization after splitting the train and test set as we have to assume that we never saw the test set before, however using this specific normalization which is a specific to a single vector, this is not a problem.

3.2 Support Vector Machines

For classification we tried several algorithms initially, including kNN, Decision Trees or Naive Bayes, however looking at the accuracy Support Vector Machines gave way better results, so we moved on with them.

Two implementations of SVMs with the same parameters were used here: one implemented from scratch based on the SMO algorithm, the details of the implementation can be found **here** (in Romanian though). The other implementation comes from **sklearn**.

3.3 Parameters used

Since the dimensionality of our data is extremely high (there are many tokens), this motivates us to choose a SVM with Gaussian kernel (commonly also called **radial basis function kernel**) as we don't expect the data to be linearly separable (although this doesn't seems to be the case for our dataset as a linear SVM performed quite close to the one mentioned previously).

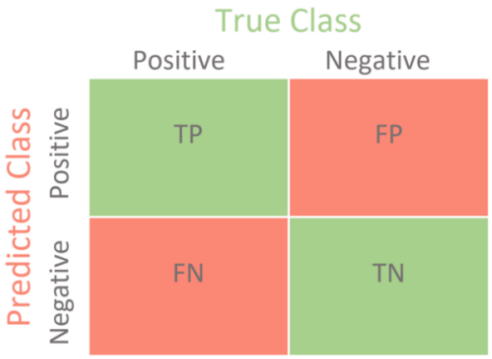
Next, as we described in the Feature Selection section we are going to use only a part of our tokens, initially we take 10% of the total words and then modify two of the Support Vector Machines parameters, namely C and γ in order to find their optimum for our dataset. In this case $C = 5$ and $\gamma = 1$ was found to be the best. Furthermore we also vary the amount of tokens that we'll select using information gain, 7.5% was surprisingly quite decent, however using 15% tokens gave the best accuracy.

3.4 Metrics

In order to evaluate our model we're going to use as metrics the Global Accuracy alongside the F1 Score. The first one is straight forward since we just need to compute number of correct predictions divided by the number ot toal predictions made:

$$\texttt{Acuracy} = \frac{\texttt{\#Correct Predictions}}{\texttt{\#Total Predictions}}$$

The F1 Score can be found as the harmonic mean between the precision and the recall, those two rely on the confusion matrix shown bellow.



One confusion matrix is built for every possible class (tags in our case), a single matrix containing 4 counters as shown in the above image. Basically

for every item in the test set one counter is increased as follows: TP if the prediction was the same as the real class, FP if the class was predicted but this wasn't the right choice, FN if the class wasn't predicted even though it should have been and TN if the class wasn't predicted and this is the correct choice. It's worth mentioning here for clarity that every single confusion matrix is updated for every item (mostly the TN will be increased if there are many classes). And from this we can extract the metrics as:

$$\texttt{Precision} = \frac{TP}{TP + FP} \; ; \; \texttt{Recall} = \frac{TP}{TP + FN};$$

$$\texttt{F1} = 2 \cdot \frac{\texttt{Precision} \cdot \texttt{Recall}}{\texttt{Precision} + \texttt{Recall}}$$

4 Results & Conclusions

At first, many simulations were done just to find the optimal parameters and the amount of tokens to be selected after computing the information gain, those being $\gamma = 1$ and $C = 5$ for a Gaussian (RBF) kernel in SVM. Also out of the total tokens only 15% were selected in feature selection.

Next, we will focus mostly to show the results for what was described in the Introduction. But first, in order to see why we have chosen to work with a SVM (with Gaussian kernel) and not with other classifiers we can take a look at Figure 2. While kNN and Decision Trees gave quite poor results, a SVM with a linear kernel was not that bad. However, we sticked with the one mentioned initially. Those results from bellow were obtained using the implementations from sklearn.

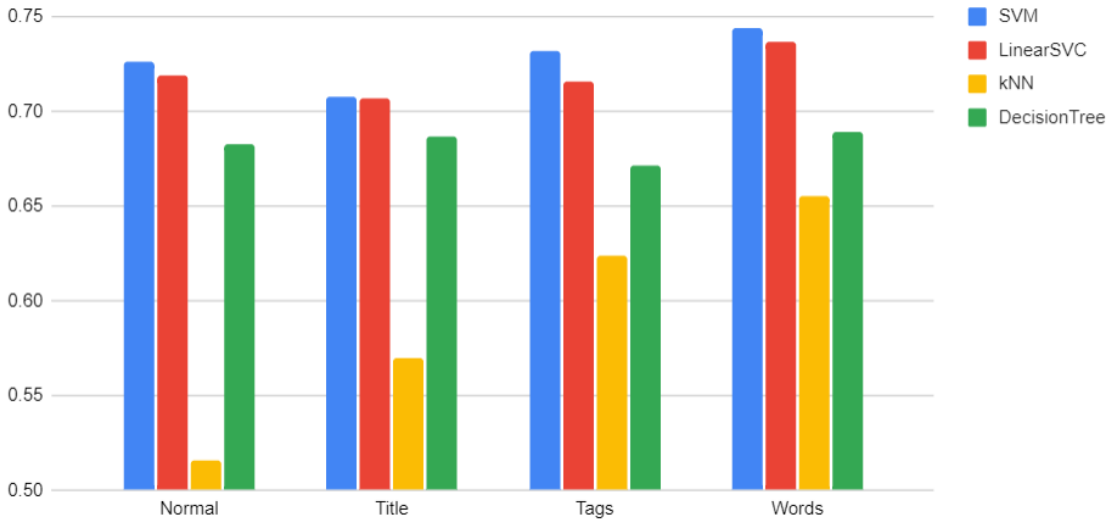


Figure 2: Accuracy for some different classifiers

In order to describe the legend from above, ***Normal*** means that nothing was modified in feature extraction, ***Title*** means that the title received more importance (in terms of frequency) being repeated a couple times. Similarly for ***Tags*** the name of the programming languages found in the text were multiplied in order to increase their frequency. For ***Words***, more importance was given to a manually selected list of words (this includes names of

some frameworks, libraries and so on). The same will apply to the upcoming charts.

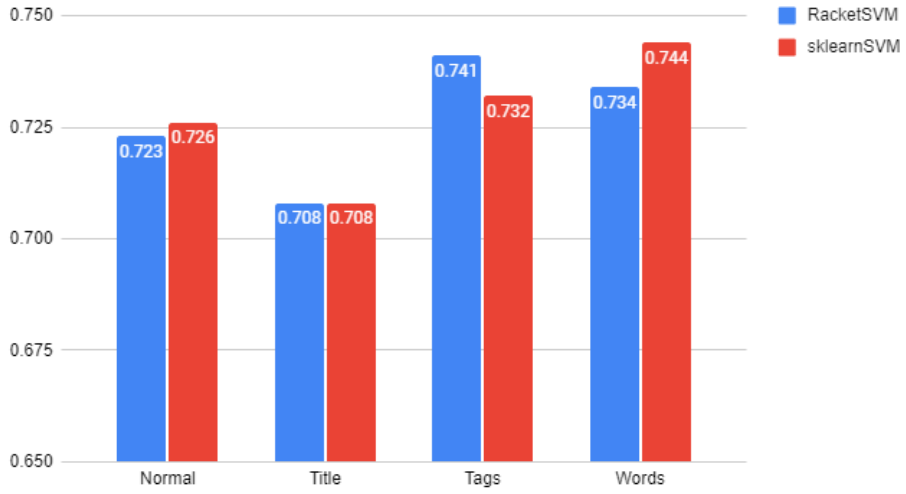


Figure 3: Accuracy for the two implementations of SVM

In Figure 3 it's shown the difference in accuracy between our implementation of SVM in Racket and using the one found in sklearn, although the results are quite similar, using something already implemented from a library it was shown to be way faster (RacketSVM took about an entire hour for the whole process, whereas the other one under 5 minutes). It's worth pointing out that this was performed only on 7546 items as RacketSVM would take too much time for a larger dataset.

The same parameters were used for both SVMs ($\gamma = 1,\ C = 5$), however according to the documentation from sklearn the multiclass support is handled with a one-vs-one scheme, whereas in our implementation a one-vs-the-rest scheme was used, so this might give the tiny difference.

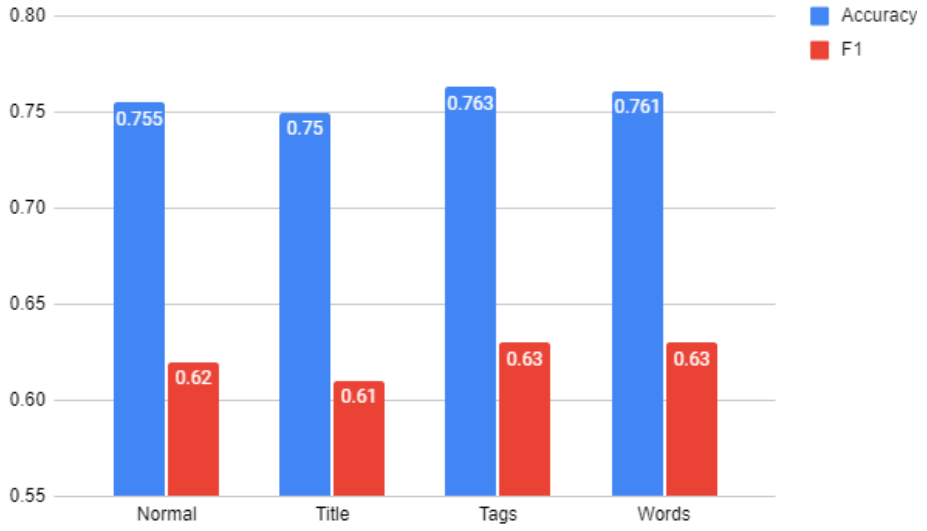


Figure 4: Accuracy and F1 Score for a larger dataset

However, since it would be better to draw conclusions from a larger dataset, we also performed the same classification with 29192 item (choosing all possibles posts from a list of 8 programming languages), this is shown in Figure 4.

Surprisingly, we can observe that even though we expected the title to be quite important (as it is from a human perspective) it turns out that it actually downgrades the performance. In contrast, for a computer, specific words tends improve the accuracy, but not the whole title.

Finally, we can draw a couple conclusions out of this paper:

- It's not worth reinventing the wheel without a specific reason in mind as the computation time can get extremely high, especially when there exists libraries which already implemented more optimally the same algorithms.
- Support Vector Machines performs quite good compared to other classification algorithms in NLP (at least in this case).
- Even though for a human when reading a post it might look like the title is more important than other stuff from the posts, this is not a case for a computer.

As a future development, trying other classification algorithms and enlarging the dataset with more tags would be a great idea in order to strengthen those conclusions.

5 Bibliography

[1]: Daniel Morariu, PhD thesis: Contribution to Automatic Knowledge Extraction from Unstructured Data, 2007
[2]: Jose Portilla, Natural Language Processing with Python (Udemy course)
[3]: Jan Zizka et al. Text Mining with Machine Learning Principles and Techniques, 2020
[4]: D. Jurafsky and J. Martin, Speech and Language Processing, 2021
[5]: Towards Data Science (for the confusion matrix)
[6]: Kaggle's StackSample dataset