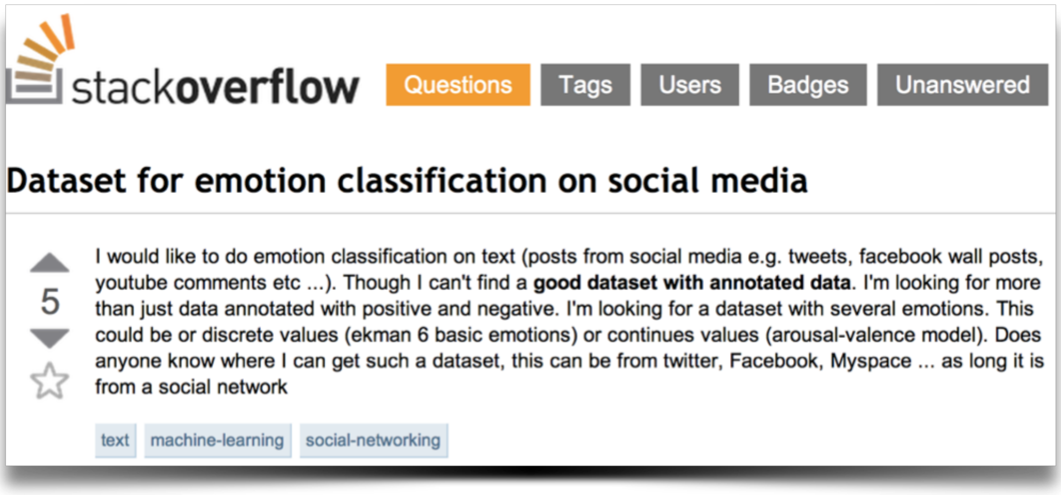
Predicting Stack Exchange Tags

Rose Perrone[*]

(Dated: December 14, 2013)

This paper shows a method of predicting the tags assigned to Stack Exchange questions, given only the title and description of millions of Stack Exchange questions.



Dataset for emotion classification on social media

I would like to do emotion classification on text (posts from social media e.g. tweets, facebook wall posts, youtube comments etc ...). Though I can't find a **good dataset with annotated data**. I'm looking for more than just data annotated with positive and negative. I'm looking for a dataset with several emotions. This could be or discrete values (ekman 6 basic emotions) or continues values (arousal-valence model). Does anyone know where I can get such a dataset, this can be from twitter, Facebook, Myspace ... as long it is from a social network

text machine-learning social-networking

I. INTRODUCTION

Stack Exchange is the a popular question-and-answer website. When someone asks a question, they must provide a title, description, and tag the question with 1-5 tags. Note the three tags on the question above.

Answerers use the tags to find questions on topics in their expertise. This task of predicting the tags is a Kaggle competition. The competition provides 7GB training data and 2GB testing data.

This was a fun learning project becasue I tried out different algorithms, features, and tuning parameters and the results of changes often surprised me. To get a much higher F1 score, however, I would use a deep learning method like a neural net.

[*] rose@cs.stanford.edu

II. METHODS

My latest algorithm has four major steps, which I'll discuss in turn.

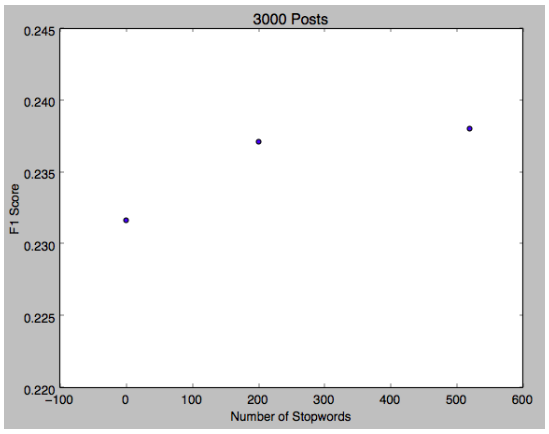
1. Data purification
2. Feature extraction
3. Logistic regression
4. Tag count selection

Note that when running experiments to tune parameters whose results you see in the plots below, I assigned three tags to every post, because there are an average of three tags per post. Each plot is titled with the number of training examples. I typically tested on 1/8 that number. I found that the F1 score plateaud at around 5000 traning examples.

The only algorithm I didn't implement myself was cosine similarity, for which I used scikit, because finding the cosine similarity was the most time intensive step in my algorithm, and scikit uses a c compiler.

A. Data purification

I stripped out HTML markup, unimportant punctuation, capitalization, and stopwords. I hand-selected over 500 stopwords from the list of most frequent words that weren't themselves tags. Below is a plot of the number of stopwords vs. F1 score.



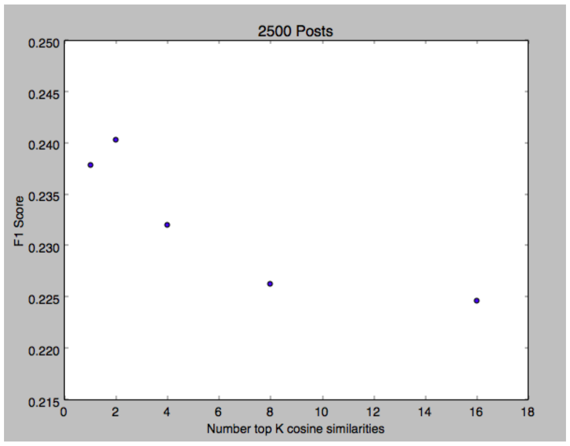
B. Feature extraction

I tried out a variety of features, but kept only those that my optimization algorithm gave significant weight. These four features are described in the following sections. The following weights were learned by logistic regression.

Feature	Weight
Probability of the text given the tag	0.07
Frequency of tag in title	0.17
Tag popularity	0.18
Cosine similarity score	0.15

1. *Cosine similarity*

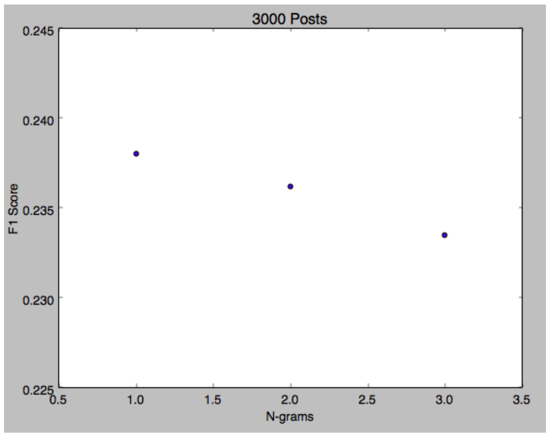
I generated what I call a *cosine similarity score* for each tag by converting each document to a tf-idf vector and finding the two documents that are most cosine similar, and then for each tag, I summed the cosine similarity scores of the documents to which it was assigned. I chose the two most similar documents because of the following finding.



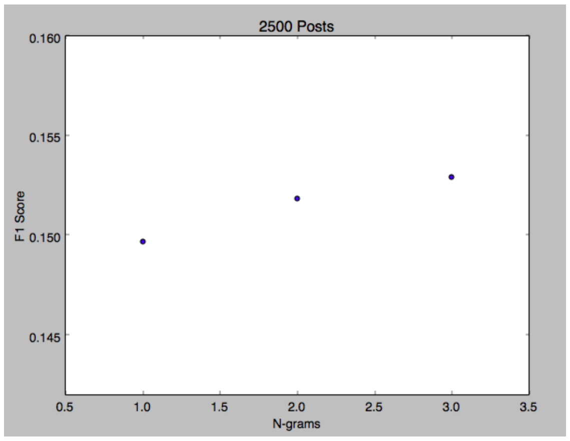
I also found that tag prediction based only on cosine similarity using only one most cosine similar document gave an F1 score of 0.17, compared to my algorithm's F1 score of 0.25.

2. *Probability of the text given the tag*

I computed the probabilitiy of each ngram found in the posts given each tag. I summed these probabilities for each post per tag, giving me an estimation of the probability of the text given my tag. This feature is similar to the Naive Bayes score, except that I include the probability of the tag in a separate feature. Unigrams gave a better F1 score than bigrams and trigrams, as shown in this plot.

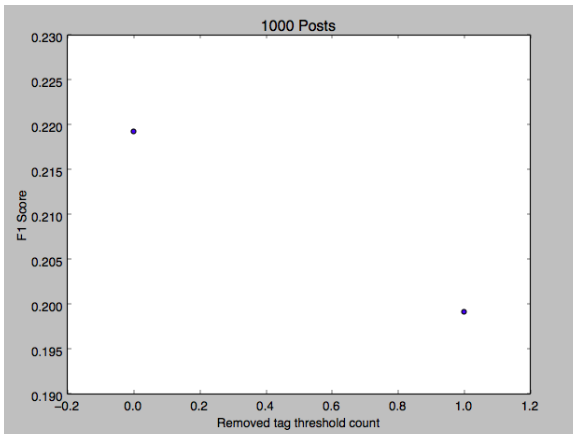


There is a discrepancy between this trend and the success of trigrams when I run my implementation of Naive Bayes by itself. I have not resolved this discrepancy. When I run Naive Bayes, I assign three tags per post, as I do in my full algorithm. When I run Naive Bayes on unigrams, bigrams, and trigrams, I get the following results. The best result, an F1 score of 0.15, is still much lower than the best F1 score I get on the full algorithm, which is 0.25.



3. *Popularity of the tag*

The popularity of the tag is the number of times it is assigned to training samples. I was curious if removing the least popular tags from my candidate set would improve the F1 score, but it didn't, as shown below.

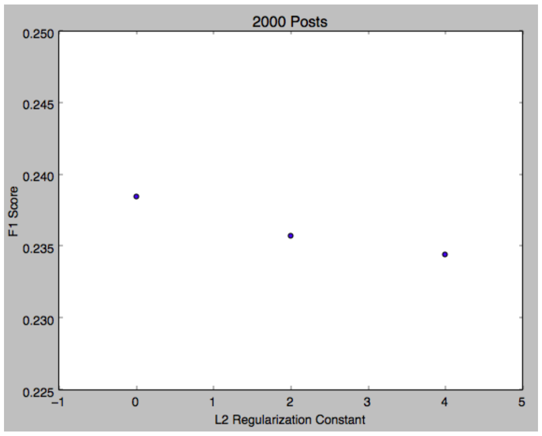


4. *Tag frequency in the title*

This feature is simply the number of times the tag appears in the title of the post.

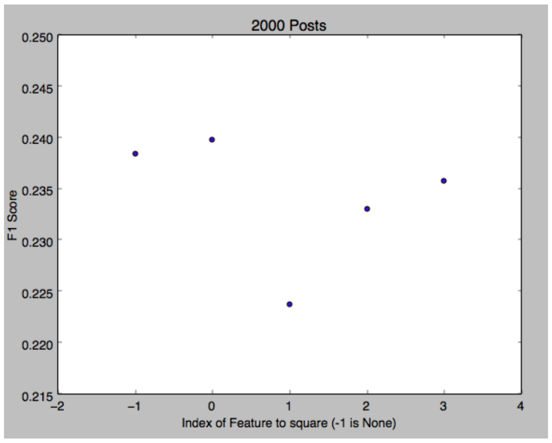
C. Logistic regression

I normalized the design matrix to mean zero, standard-deviation 1 for each feature. I gave each feature vector a supervised output of 1 if the tag was assigned to the post on which the feature vector was generated, and 0 otherwise. Because I pared my number of features down to 4, my algorithm was tolerant to changes in L2 regularization as shown below.



I got to see how weights and covergence changed on each iteration of gradient ascent when I varied the learning rate, convergence constant, data size, and different features. I set the convergence constant to 1e-12 and the learning rate to 1e-5.

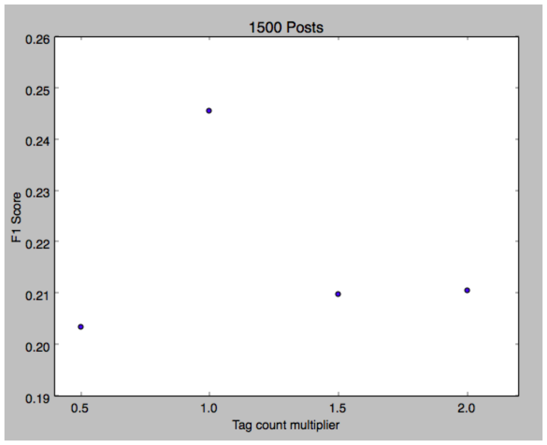
I tried squaring each feature, testing the effect of squaring a single feature one feature at a time, but I found no squared feature significantly improved the F1 score, as shown below. The -1 is the baseline where no features are squared.



Key
x=0 shows probability of the text given the tag
x=1 shows tag frequency in the title
x=2 shows tag popularity
x=3 shows cosine similarity score

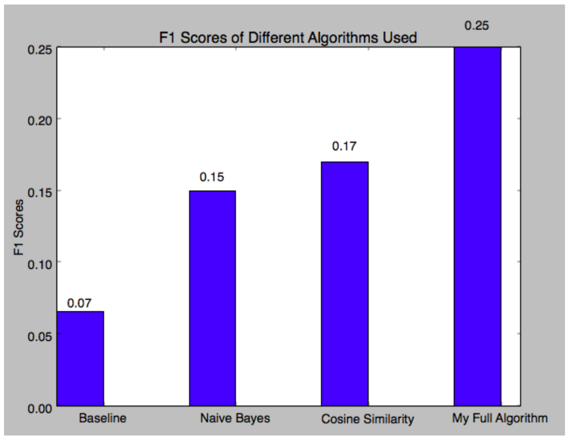
D. Tag count selection

I could choose to assign between 1 and 5 tags to each post. To choose that tag count that would most improve the F1 score, I kept an online average of the top five tag scores per post, where a tag score is the dot product of the tag's feature vector and the learned weight vector. For each post, I assigned the top k tags, where $k = max(1, \text{number of tags with a score above } 1.5 \times \text{(average tag score)})$ I chose 1.5 because of the following result.



III. RESULTS

The baseline was assigning the three most popular tags to every post. That gave an F1 score of 0.066. My full algorithm gave an F1 Score of 0.25. A comparison of the algorithms I used is shown below.


F1 Scores of Different Algorithms Used
F1 Scores
Baseline
Naive Bayes
Cosine Similarity
My Full Algorithm