

# Applications of Machine Learning on Keyword Extraction of Large Datasets

Meng Yan

*my259@stanford.edu*

## Abstract

Given a large text dataset composed of greater than 200k training sets and multiple classifications, various machine learning algorithms were used to train and predict tags and keywords. This project also explores various techniques in pruning and managing a large, unwieldy dataset in order to produce a practical training point. Naïve Bayes and SVM are the algorithms focused on in this project, with various contours of the dataset tested to examine the practical effects of dataset manipulation.

## 1 Introduction

Keyword extraction is a common problem that exists today due to the rapid growth of data available online and in databases. In order to make such a large body of information readily accessible to the general public, it is pertinent that accurate and efficient text classification algorithms are used to sort and index this information.

For this project, I utilize a large set of data provided by Kaggle from their competition “Facebook Recruiting III – Keyword Extraction”. This competition involves training on an 8 GB training set to predict the tags for a 2.3 GB testing set. Broken down, the data consists of roughly 6 million training examples with 2 million testing examples, and a total of 42000 different tags, with the possibility of multiple tags associated with a single example. As such, the data management and computer workload management is a nontrivial task. For the purposes of this project, I decided to limit myself to 200k examples from the data set, training on the 10 most prevalent tags.

First, the raw data used in this project will need to be processed and pruned into a manageable and useable format. This will involve applying various techniques learned in class, such as stemming, stop words, and filter feature selection. After parsing the data, Naïve Bayes and Support Vector Machine algorithms will model the data and be compared against each other for both accuracy and computation time. Finally, different contours of the data will be explored to see the effects on accuracy and computation time, in hopes of finding an approach that can be extrapolated to the entire 10 GB data set.

## 2 Dataset Manipulation

Each of the training sets contains four components: index, title, body, and tags. The tags are based on information derived from the title and body, which are plaintext components of arbitrary length. Based on a rough analysis of only the first 50,000 training examples, there are around 1.6 million unique words in their titles and bodies alone, with 13,000 unique tags. Clearly, the data must be pruned in order to create a usable dataset.

In this project, only the first 200k training examples were used and categorized based on the

43 top ten most used tags {c#, java, php, javascript, android, jquery, c++, python, iphone,  
 44 asp.net}. Simple cross validation was used, with a 75%/25% split between training and  
 45 testing examples. The idea is that the time to test each model for the full set of data will be  
 46 roughly proportional to the smaller test set used here. The time to train the models, however,  
 47 will drastically increase with the size of the dictionary. Fortunately, the actual act of training  
 48 has to be done only once, and can be prepared in advance. As such, computation time will  
 49 examine mainly testing time with the models as opposed to training time.

50 The following pruning techniques were applied to create a manageable dataset.

- 51 - Only title information was used, and body information was ignored as it drastically
- 52 increased the dictionary size.
- 53 - Non alphanumeric characters were removed.
- 54 - All characters became lower case.
- 55 - Repeat instances of a word in a tag or body were removed, as there are no plans for
- 56 multinomial event modeling.
- 57 - Solely numeric words were removed (ex. cs229 still remains a valid word, but 2013
- 58 is pruned).
- 59 - Word stemming was applied when applicable.
- 60 - Stop words were pruned away from the title and body.

61 To handle stemming and stop words, the Natural Language Toolkit (NLTK 3.0) for python  
 62 was used. Stop words provided through the NLTK libraries were used as a baseline, and  
 63 additional words were added to the list to handle word and symbol fragmentation due to the  
 64 initial parsing (ex. <p> is pruned away to p).

65 Filter feature selection has the potential of greatly reducing the dictionary of unnecessary  
 66 words, as is easily applicable. For instance, in the set of 6 million training examples, there  
 67 are instances where words in the dictionary are not used more that 10-50 times. A  
 68 straightforward way to parse the dictionary would be to set a lower threshold in which the  
 69 occurrence of a word must exceed before being used as a feature. In depth analysis is  
 70 required to understand what a reasonable threshold would be. However, without filter feature  
 71 selection, using the title and body information, even with the pruning, will still produce a  
 72 dictionary that is too large to process. As such, two contours of the data will be explored:  
 73 one with no filter feature selection (dictionary size of 39965 for the 200k data set), and one  
 74 with a filter feature selection threshold of 25 (dictionary size of 20858 for the 200k data set).

### 75 3 Naïve Bayes

76 Using NLTK 3.0 Naïve Bayes Classifier, with a 150k training set and 50k test set and no  
 77 filter feature selection, I obtained the following results for the top ten most used tags.

78

79 Tag	Positive Examples	Negative Examples	Accuracy	Testing Time (seconds)
80 c#	15438	184561	0.90368	3.778
81 java	13632	186367	0.94208	4.114
82 php	12938	187061	0.94372	3.651
javascript	12179	187820	0.92216	3.751
android	10622	189377	0.96922	3.972
jquery	9906	190093	0.94228	3.356
c++	6716	193283	0.94550	3.992
python	6244	193755	0.97118	3.643
iphone	6107	193892	0.95984	3.968
asp.net	5835	194164	0.94828	3.890

83 *Table 1 - Naive Bayes Baseline, no Filter Feature selection, 39965 dictionary size*

84 The total training time for Naïve Bayes without filter feature selection is 91.642 seconds.

85 Note a separate model per tag was needed, as we cannot apply the discretized Naïve Bayes  
 86 algorithm in this instance. That is, the tags are not mutually exclusive; a training example  
 87 has the possibility of being tagged with multiple tags. A cursory glance at the results  
 88 suggests that naïve bayes is a good predictor even with only using the title as the dictionary  
 89 (and completely ignoring the body information). Below are the results after applying filter  
 90 feature selection (note that the number of positive and negative examples remains constant).  
 91

Tag	Accuracy	Increase in Accuracy (%)	Testing Time (seconds)	Speed up (%)
<b>c#</b>	0.90792	0.4691926	3.835	-1.50873
<b>java</b>	0.94628	0.445822	3.783	8.045698
<b>php</b>	0.94768	0.419616	3.956	-8.35388
<b>javascript</b>	0.9269	0.5140106	4.202	-12.0235
<b>android</b>	0.97216	0.3033367	3.95	0.553877
<b>jquery</b>	0.947	0.5009127	4.216	-25.6257
<b>c++</b>	0.9539	0.8884188	3.317	16.90882
<b>python</b>	0.97856	0.7599003	3.869	-6.20368
<b>iphone</b>	0.96634	0.6771962	3.735	5.871976
<b>asp.net</b>	0.9569	0.9090142	3.78	2.827763

92 *Table 2 - Naive Bayes, Filter Feature Threshold 25, 20858 word dictionary*

93 The total training time for Naïve Bayes with a filter feature selection threshold of 25 is  
 94 76.902 seconds.

95  
 96 It is surprising to see that accuracy slightly increased with filter feature selection, which  
 97 implies the low frequency words did not have any prediction value (at least for the selected  
 98 tags) and noise was effectively removed by filter feature selection. As for testing time and  
 99 total training time, the timing methodology was to simply compute the execution time to call  
 100 the classification function of the Naïve Bayes Algorithm. Though the same machine was  
 101 used for all runs, not all transient variables were accounted for; as such, the compute time  
 102 comparisons serve only as a rough estimate. Therefore, between the two different Naïve  
 103 Bayes contours, their testing times were roughly equivalent, though the smaller dictionary in  
 104 the filtered Naïve Bayes run did produce a noticeably faster training time compared to the  
 105 baseline.  
 106

#### 107 4 SVM

108 Below are the results from the baseline SVM results with no filter feature selection. The  
 109 total training time for SVM with no filter feature selection was 680.23 seconds. The data is  
 110 compared against the Naïve Bayes baseline.  
 111

Tag	Accuracy	Increase in Accuracy (%)	Testing Time (seconds)	Speed up (%)
<b>c#</b>	0.94236	4.2802762	0.117	3129.06
<b>java</b>	0.96638	2.5793988	0.105	3818.095
<b>php</b>	0.96886	2.6639257	0.104	3410.577
<b>javascript</b>	0.9622	4.3419797	0.134	2699.254
<b>android</b>	0.9862	1.7519242	0.1086	3557.459
<b>jquery</b>	0.97878	3.8735832	0.096	3395.833
<b>c++</b>	0.97872	3.5134849	0.101	3852.475
<b>python</b>	0.98944	1.880187	0.099	3579.798
<b>iphone</b>	0.98394	2.5108351	0.106	3643.396
<b>asp.net</b>	0.98164	3.5179483	0.096	3952.083

112 *Table 3- SVM baseline, no Filter Feature Selection, 39965 word dictionary*

113 To perform SVM, the liblinear library was used (v1.93). This first involved converting the

python generated datasets into a Matlab friendly format, and then formatting the data into sparse matrixes to pass into the train and predict functions. While the conversion to python to matlab format is not counted against the training time for SVM, the creation of the sparse matrixes is. As such, there is a dramatic increase in training time compared against the Naïve Bayes baseline (almost 7 times as long), mainly due to the un-optimized generation of the large sparse input matrixes.

However, the testing times are orders of magnitude faster than that of Naïve Bayes. Granted, some of the speed increases may be due to the nature of the python Naïve Bayes and C based SVM implementations, but in general, the efficiency of SVM can clearly be seen in the comparison. Extrapolate the results of the baseline algorithms to the full data set, assuming Naïve Bayes' average test time is 3.812 seconds and SVM's average test time is 0.101 seconds, for 42000 separate tags/classifications, it would take Naïve Bayes roughly 2 days (44.47 hours) to classify all tags while SVM would only take 70.7 minutes. Note that this is only on 50000 test points too; for the full test set of 2 million data points, the actual testing computation time may be drastically higher, so much so that Naïve Bayes is infeasible. As such, for very large data sets, SVM is clearly the optimal solution, regardless of its increased training time.

Filter feature selection was also performed for SVM. Below are the results. The comparisons are against the SVM baseline. The total training time for filter feature selection with a threshold of 25 is 622.3057 seconds.

Tag	Accuracy	Increase in Accuracy (%)	Testing Time (seconds)	Speed up (%)
<b>c#</b>	0.9271	-1.6193387	0.1073	8.290598
<b>java</b>	0.93576	-3.1685258	0.0979	6.761905
<b>php</b>	0.94182	-2.7909089	0.099	4.807692
<b>javascript</b>	0.94336	-1.9580129	0.112	16.41791
<b>android</b>	0.9503	-3.6402352	0.1004	7.550645
<b>jquery</b>	0.95276	-2.6584115	0.0985	-2.60417
<b>c++</b>	0.96764	-1.1320909	0.1084	-7.32673
<b>python</b>	0.96908	-2.0577296	0.1016	-2.62626
<b>iphone</b>	0.97236	-1.176901	0.0977	7.830189
<b>asp.net</b>	0.97194	-0.9881423	0.0981	-2.1875

Table 4-SVM, Filter Feature Threshold 25, 20858 word dictionary

As opposed to the Naïve Bayes results, the filter feature threshold of 25 word occurrences caused a decrease in SVM's accuracy compared against the SVM baseline. However, SVM with filter feature selection is still more accurate than Naïve Bayes. The testing times roughly stayed the same. As such, adding filter feature selection for SVM decreased the training time by roughly 10%, but also decreased accuracy by roughly 2% overall. Given the situation and circumstances, the drop in accuracy may be reasonable for the faster model generation.

## 5 Future Improvements

The use of filter feature selection at a word frequency threshold of 25 reduced the dictionary size roughly by half. However, a dictionary of 20858 words is still very large for a training set of 150k examples. Also, as seen with filter feature selection on Naïve Bayes, a higher threshold may still be used to further reduce the dictionary size without any negative impact on accuracy. This implies that further reductions of the dictionary may be desired. To achieve this, Principle Component Analysis seems like the best choice. Not only will this remove the trial and error approach associated with guessing a proper threshold for filter feature selection, but it will also remove frequent redundant features that will be ignored by solely using filter feature selection.

156 Another improvement would involve exploring ways to reduce the training time for the data  
157 set. This is especially pertinent for SVM, as the creation of the sparse matrixes for liblinear  
158 is a very time consuming process. Though this goes beyond the scope of this project,  
159 exploring efficient sparse matrix generation algorithms is a practical consideration that is  
160 definitely needed in real world usages of keyword extraction.

161

## 162 **6 Conclusion**

163 Keyword extraction on large data sets introduces a large number of practical issues which  
164 render certain machine learning algorithms more desirable than others. In particular, SVM is  
165 seen as a much faster and feasible method of keyword extraction compared to Naïve Bayes.  
166 However, this is only true for very large data sets, as the initial cost of training SVM is much  
167 greater than that of Naïve Bayes. To reduce training times, various methods of pruning away  
168 unnecessary information can be applied and formalized, such as filter feature selection and  
169 PCA. As such, with a properly managed feature set and efficient algorithm, keyword  
170 extraction can be performed accurately and within a reasonable amount of time and with  
171 fairly inexpensive hardware.

172

## 173 **7 Citations**

174

175 Bird, Steven, Edward Loper, and Ewan Klein (2009).  
176 Natural Language Processing with Python. O'Reilly Media Inc.

177

178 R.-E. Fan, K.-W. Change, C.-J. Hsieh, C.-R. Wang, and C.-J. Lin. LIBLINEAR: A Library  
179 for Large Linear Classification, Journal of Machine Learning Research 9(2008), 1871-1874.  
180 Software available at <http://www.csie.ntu.edu.tw/~cjlin/liblinear>

181

182 "Facebook Recruiting III – Keyword Extraction". Kaggle (2013). Web.  
183 <http://www.kaggle.com/c/facebook-recruiting-iii-keyword-extraction>