Alan Newman
Devin Guillory

Predicting Stack Exchange Keywords

Abstract

In large scale systems driven by user uploaded content, tagging has become increasingly popular, as it leads to efficient ways to parse, group, and annotate data dealing with similar topics or themes. This project is based on a current Kaggle competition, in which the goal is to identify keywords relevant to Stack Exchange posts. Our dataset contains six million training entries and is evaluated over two million test entries. Each given entry is associated with between one and five respective tags. By segmenting each datum into title, text, and code, and running individualized parsing algorithms on the sets, we are able to construct a series of term frequency-inverse document frequency (tf-idf) matrices. In addition to tf-idf, we were also able to implement several more advanced NLP features based on sentiment analysis and POS tagging. By then pruning the tag space to include only the most commonly occurring tags, we constructed a multiclass, multilabel classifier mapping the tf-idf features to our tag subset using stochastic gradient descent with hinge loss. After receiving the output confidence scores, we apply a logical inference step to select the top relevant tags for each document. In order to handle the scale of the dataset we utilized cloud computing via a compute-optimized Amazon ec2 instance. Our combination of NLP and ML techniques were able to generate F1 Score of 0.754, which currently places us 27th out of 276 in the Kaggle competition (note, our placement is subject to change, as the competition ends Dec. 20th).

Introduction

In recent years, the rapid growth of user-generated content on the web has contributed to a rise in human-based tagging systems. These tagging systems are aimed at annotating content with useful metadata that could be used for browsing, searching, and organizing information. These human-created tagging systems may allow either content-creators or public viewers to create tags for the post. Common examples of large scale tagging sites include social media sites such as Twitter, Instagram, del.icio.us. Other examples include blogging and forum sites like StackOverflow and Piazza. Being able to correctly predict tags could be useful for aggregating untagged data, or suggesting tags to users. It could also be useful for increasing the search space and improving browsing experiences. Moreover, these sorts of prediction systems have a wide range of applications where only some documents may be annotated with keywords--notably in scientific and academic databases (e.g. Pubmed, Scifinder, JSTOR). Further, our system could even be adapted to handle legal case documents or briefs in Westlaw or LexisNexis.

Previous Work

With recent increases in the popularity of tagging systems in web usage, there has been research in the field on tag prediction. Sood et al, proposes a TagAssist algorithm whose goal is to guide the

users in labeling their data with meaningful tags, based on the predicting commonly used and appropriate tags for the post [2]. TagAssist applies lossless compression to the set of tags, by clustering tags based on similarities in the posts. In the validates the compression by examining co-occurence of tags and observing variation from the cluster centroid. After developing these compressed clusters, tag prediction is performed by searching through the cluster space for similar posts. After establishing the top 35 matching tag prediction is determined by find the frequency of the tag occurrence within the top 35 matches, the occurrence of the actual tag in the text, the popularity of tag in the training set, and by clustering. Heyman et al, did work on social tag prediction using a large data crawl of social tagging site del.icio.us [4]. The also relied on semantic relationships between tags to generate prediction. They proposed that each post was represented by three set: the set of tags which doesn't describe the post, the latent set of tags which describes the post, and the given set of tags which annotate the post. Using these sets they were able to develop classifiers for prediction, as well as classifiers that describe the predictability of a particular post. Previous work in this field relies heavily on determining the semantic relationship between tags in a dataset. Our work does not currently attempt to perform this type of association, but could potentially benefit from it in the future.

Evaluation Metric

The competition uses an evaluation metric of mean F1. F1 measures the harmonic mean of precision and recall--that is, F1 = 2 * precision * recall/ (precision + recall). Precision = # of true positives / (# of true positives + # of false positives), and recall = # of true positives / (# of true positives + # of false negatives). In this competition, each test datum receives its own F1 score and then the average of all test data F1 scores are averaged, resulting in the mean F1. It is also important to note that tag synonyms are not taken into account--that is, even if two potential tags corefer, only the gold tag will be accepted.

Dataset Analysis

The training dataset contains 6,034,195 entries and 42,048 tags with between 1 and 5 tag(s) per datum (with an average of 2.89). The top 10 most common tags are: c#, java, php, javascript, android, jquery, c++, python, iphone, and asp.net; examples of infrequent tags (specifically those that have a document frequency of 1) include: closesocket, pushmobi, and qatar. Each entry in the training dataset contains: id (a unique identifier), title (the title of the post), body (the body of the post--includes both text and code), and the relevant tags. The test dataset contains 2,013,337 entries and follows the same format without the tags. Assigning the tags: javascript, c#, python, php, and java to each test datum results in an F1 of .07484, and the current highest achieved F1 in the competition is .81539 (although this will likely improve). It also became apparent that roughly 70% of the test set was composed of the training set. A Kaggle administrator acknowledged that utilizing this knowledge was not only acceptable, but also encouraged, as its uncovering required extensive analysis of the two datasets, which is a desirable trait in a strong competitor.

High Level Discussion of System

Our overarching assumption is that we can achieve better results with a feature-based classifier than a rule-based system. This is due to the complex nature of natural language and the consequently amorphous requirements of keyword prediction frameworks. Past research in the field of NLP has shown that comparatively simple feature-based machine learning classifiers can achieve better (and more generalizable) results than rule-based classification systems. Nevertheless, the success of feature-based classifiers is often contingent on having a sufficiently large repository of annotated data. Given that our dataset is particularly large and fully annotated, it is reasonable to conclude that a learning system could figure out significant structural patterns that can account for the application of various tags.

Another assumption that we made is that features features should be collected and localized to three spaces: title, text, and code (which requires parsing the body of each datum and pulling out the relevant code into its own category). The intuition behind this decision is that differing vocabularies, different "meanings", different stop words, and different weightings are likely to be used in the three sections. For example the term 'public' in "public static class" and "the public thinks python is better than java" has two distinct referents. Moreover, consider "python is good for you" and "for i in range(0,len(alist)):". In the former, 'for' should be ignored (that is, 'for' should be considered a stop word); however, in the latter, 'for' is essential--and can quickly demonstrate that the tag "html" is probably less likely than that of, say, "python".

Thus, the basic X input (just the tf-idf based features--more elaborate NLP features were also used and will be discussed in a subsequent section) to the classifier consists of the horizontal stacking of three tf-idf matrices. It is significant to note that each tf-idf matrix must be sparse--that is, in the context of any particular datum, only a small subset of the title, text, and code vocabularies will be represented with nonzero values. The implication is that utilizing dense (or normal) matrices, where every value in the matrix receives a bit of memory results in a ~ 6,000,000 by ~400,000 feature matrix. If we have 64-bit float values, we end up with a memory requirement of 1.92 * 10^13 bytes, or more than a terabyte. This construction is obviously impractical (and unnecessary, as the vast majority of values are 0). Thus, we instead only store in memory values that are not zero, which results in a ~ 99.8% decrease in required ram and a robust increase in efficiency as well (as zero values are simply ignored).

However, the use of sparse matrices (and sparse features at the theory level) requires a classifier that can correspondingly handle both the practical and theoretical requirements of sparse matrices. While some implementations of ensemble methods using decision trees (random forest or extremely random trees), for example, may practically handle the use of sparse matrices, at the theoretical level, sparse inputs will not lead to ideal outcomes with these sorts of methods, although it is outside the scope of this paper to delve into an even remotely thorough explanation.

Now because our task requires the assigning of one (to five) of over 42,000 tags, it was critical that we chose a classifier that can also appropriately handle a largely multiclass framework. The first choice was between one vs one (or pairwise--each combination of two tags gets a classifier) and

one vs all (or binary--each tag gets a classifier); while the former often results in better performance (as it allows for a more fine-grained level of detail in the context of choosing between any two tags), it causes an exponential explosion in the number of individual classifiers--that is, even if we disregard 99% of the tags, 2^420 classifiers would be difficult to train in the scope of this project (or at 25 seconds per classifier training, would take roughly 2.17 * 10 ^120 years to complete).

While support vector machines often boast ideal (or near-ideal) performance in this kind of text classification problem, because of the dimensionality of our input set and the number of possible output classes, we decided instead to use stochastic gradient descent, which with a hinge loss function results in a reasonable trade-off between performance and efficiency.

It is last worthwhile to note that we used a 32-core (with 108 ecus) ec2 system with 60gb of ram. In order to fully utilize our instance, we used Python's multiprocessing .pool to dirtily multiprocess a variety of tasks implemented with Python's map(). Moreover, Scikit-learn includes dirty multiprocessing built in to our stochastic gradient descent classifier, so we were able to at all times have every core doing some sort of processing.

Output Tag Determination (interpreting the classifier output)

Any singular post in our dataset may have between 1 and 5 associated tags. However, using our one-vs-all multiclass classifier results in a confidence score (signed distance to the respective hyperplane) for every tag. This is important because we there's no immediately ideal way to handle this sort of classification. That is, while we could simply assign a tag if and only if its respective score > 0, we would end up with any number of tags. Thus, intuitively it makes sense to force there to be between 1 and 5 tags. To accomplish this end, we first sweep through assigning a tag if the score is above 0. If no tags were assigned, we take the highest probability tag. On the other hand, if too many tags were assigned, we take the top 5 and cut the rest.

However, this method is obviously flawed in a few ways. First, because we're using an svm based system (or more precisely an approximation of an svm), our output scores are simply distances to the respective hyperplanes. This means that while we could normalize and scale between 0 and 1 (or perhap use softmax), we could arrive at something resembling probabilities, but they will still be flawed in one critical aspect--that is, each score is not directly comparable to another. The implication is that the only way to adequately handle the outputs would be to have a specific threshold for each tag. There are various ways to create these thresholds less painfully (using tag-clustering), but because this issue is primarily machine learning in nature (that is, far from natural language processing), we determined it was outside the scope of the project.

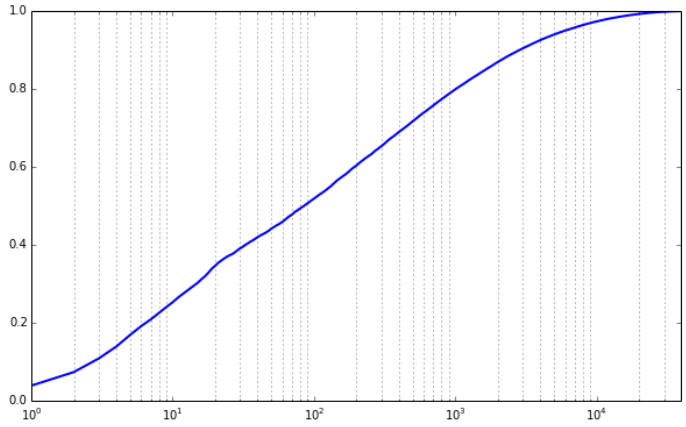
It is also relevant that, as noted in the dataset section above, the average number of tags is 2.98. Our framework ignores this point of information (as we could have scaling thresholds that encourage the indication of 2.98 tags on average). Lastly, because we're using f1, typically, it is advantageous to guess too much (instead of too little). Consider the following scenarios: (a) gold: a b c; predicted: a b; (b) gold: abc; predicted: a b c d. In (a), there are two true positives, one false negative, and zero false positives. In (b), there are three true positives, but one false positive, and zero false

negatives. The F1 of (a) is 0.8, and the F1 of (b) is 0.857, which confirms the position that guessing too much (within reason) is generally preferable. A more robust system could easily accommodate this information into a far more intelligent threshold, but again, because it is a NLP project, we chose to prioritize the enhancing of other aspects of our system.

Tag selection

We have over 42,000 possible classes. This is important for two reasons: first, computing a classifier for every tag is computationally expensive. Even on our compute-optimized ec2 instance, with our current feature-set, each tag-classifier costs roughly 30 seconds, which means that creating classifiers for every tag would take ~350 hours of training time (which is obviously unreasonable). And second, many of these tags have a document frequency of exactly one out of over 6 million documents. This means that even if we were to create a classifier for every tag, we would end up with a far noisier system (as there isn't enough data with only one positive training example to establish a classifier that doesn't make poor decisions), which would most likely result in a far lower precision for perhaps a slight boost to recall (far more false positives for slightly less false negatives).

Nevertheless, it was important to not excessively limit the possible tags--in our early tests, we ran our system with only 85 tags. Realizing this was potentially harmful but unaware of the extent, we decided that it would be fruitful to examine the maximum possible F1 score (that is, assuming our system has perfect precision and the recall is only lowered by the ignoring of some terms) on the training set at various tag counts (that is, because excluding tags will necessarily result in some amount of false negatives).



In this graph, the (log) x axis represents the number of tags used, and the y axis represents the maximum F1 score possible. This graph indicates that the benefit achieved beyond 1000 tags is quite low, and the sweet spot should be between 100 and 1000 tags. Further experimentation will clearly be required, but this analysis is helpful in that it drastically limits our required search space.

Tf-Idf Considerations

When computing the vectorization of the features to be sent to the our model for training we were faced with decisions about what type of vectorizer we should use. Our initial assumption was

that the tf-idf weighting with either linear or logarithmic term frequency would be ideal due to the fact that the weights could be normalized by their appearance in other documents thus giving high values only to terms which strongly corresponded to the particular class. However experimentation with our data showed that simply using a binary indication of whether a term was in a document in fact proved more useful than having either linear or logarithmic term frequencies. Moreover, by examining the dataset coupled with our cross-validated results, we were able to determine that this may be because of the relatively high document occurrence of discriminative terms such as “c++” and “c#” being negatively affected by their independent document frequency.

While boolean tf and ignoring idf proved ideal, we also created hand-cured stop words sets for both code and text-title (that is, we used the same stop-words set for both text and title) (reasoning and example above in the High Level Discussion of System section). It is critical to note though, that similar to the way we discounted certain tags below a threshold document frequency, we also proceeded with a similar process here--in order to reduce noise in our model. Some of the most common stemmed words found in text were: build, includ, allow, and variabl. Originally, we also limited results based on maximum document frequency (i.e. those that appear too much are also discounted), but we realized that terms like “variable” may be useful even though they appear in a large proportion of the dataset. For example, it follows that “variable” will have a low weight for tags like html, and higher for java.

Nevertheless, terms with low document frequencies, like “xoauth” or “j_usernam” seemed to simply add noise. We considered using spelling correcting prior to stemming, but concluded that given the technical nature, code-link terms (even in text), and variable names scattered around, it would not help. Thus, in order to figure out the ideal minimum document frequency, we carefully examined the minimum frequency words at various thresholds, and correspondingly used cross-validation to establish an ideal threshold as well. Ultimately, we chose 50 as a minimum document frequency, as it performed intuitively in the context of terms (roughly 10% of the bottom terms were useful, and 90% were garbage). We established this 10% rule as an ideal balance between the three competing concerns of: diminishing noise, maximizing the inclusion of potentially useful terms, and efficiency optimization.

Ngrams

We experimented with a wide variety of different ngrams for text and title (we mostly ignored code, as it’s an NLP class). While we were initially confident that our dataset, given its size, could at least in theory support higher order grams (above 1), we ultimately realized that anything above unigrams would result in too much noise--even drastically limiting the minimum document frequencies, and consequently stuck with unigrams.

Parsing

The first step in our system is to parse our dataset into three separate categories, Title, text, and code. The title for a document is simply the question that is asked as the main query. This

question is provided by the dataset. We use the python library BeautifulSoup4 to identify the code present in the body of a post. We then separate the body of the post into a section representing the code and section for the non-code text.

Instead of relying on the standard tokenizer functions we use a specially designer parser to help with tokenization. We, by hand, convert special characters that we found useful to text, e.g. '#' to 'sharp' so that the special characters are not lost in the tokenization handled by the scikit-learn library later in the system.

In order to save on memory use in training and to rapidly speed up training, we parsed our text (including stemming and lemmatization) and constructed our term to matrix transformations prior to training. This step removed roughly thirty minutes of training time, allowing us to test more models, and eased our memory cost by a factor of 2 (as we no longer had to simultaneously have in memory both the full training set csv and and the transformed matrices).

More advanced NLP Features

Originally, we wanted to try a wide variety of more advanced NLP features (than tfidf), including: multilayered POS incorporation, sentiment analysis, wordnet integration, ner tagging, coreference tagging, and others. However, after we tried single layer POS tagging and sentiment analysis, we decided to focus our efforts on other areas, as most of the literature seems to indicate that simpler features--combined with extremely careful error and dataset analysis ultimately lead to ideal systems that intelligently balance performance and efficiency.

We began by using NLTK's tagger (with TextBlob's wrapper) to tag parts of speech, thinking perhaps that nouns and adjectives may be most useful. Here, we tried two methods of incorporating this information--(a) removing undesirable parts of speech from the text and title matrix vocabularies, and (b) simply adding a new matrix with the new feature sets. While neither method was fruitful, we then hoped that simply utilizing noun phrases could be incorporated into an ideal system--again here, we did not come up with an increase in system performance (and efficiency took a robust hit). While we had wanted to stack POS tagging (that is--create features using some parts of speech followed by others), at this point, we decided our efforts would be more worthwhile elsewhere.

We had another interesting idea--that perhaps the sentiment of posts would be conveyed in the tags. For example, python or iphone supporters tend to be more "evangelical" in their support than, say, those who use html. And on the other hand, we contend that few would be excited about a post concerning sql or css. To accomplish this, we examined both polarity (positive vs negative) and subjectivity (vs objectivity). However, once again, these features weren't helpful--and actually reduced our F1.

At this point, we wanted to use NER tagging to identify any named entities in each text and title sentence, thinking that named entities are likely to refer to keywords. However, we would have to somehow train our own NER system, as none that we could find were trained on technical terms, and we aren't interested in people or places. We faced a similar problem with coreference, although suspected that it could be more worthwhile--however, here, we couldn't find a system that would work

even remotely sufficiently in the context of efficiency. There is plenty more to be done though in this area that could definitely result in an enhanced system (and will hopefully be explored in order to attempt to win the competition even after this project concludes!).

Unlike the previous work in this space we do not attempt to semantic analysis between tags. We instead utilize that the probability distribution of the tagging labels is highly skewed towards a few labels which map to large percentage of the documents. We use this fact to select only the most common tags and train over them. This both reduces the computational time by lowering the dimensionality of the target matrices, but also accounts for some of the grouping aspects of the tags.

Results

41K Subset

Method	F1	Precision	Recall
tf-idf	0.424	0.398	0.528
Binary	0.435	0.470	0.456
Count	0.396	0.430	0.41

In order to verify our design decisions we ran multiple tests over as smaller subset of data. Above are the results for the decision to use binary vectorizations as opposed to counts or tf-idf weights.

Full Dataset

Method	F1
Baseline -87 tags	0.23812
tf-idf, duplicates, thresholding, df-pruning -87 tags	0.67668
binary -87 tags	0.68543
Ngrams, Custom parsing - 350 tags	0.75393

The baseline implementation is simply the result of running our basic system on the dataset, without tuning for thresholds or pruning features based on document thresholds. The second implementation is a result of intently studying our dataset, and implementing the parameters discovered in the models discovered in the system building process. The third is the same system run with binary classification. The last result contains the 4-grams for the code section, specialized parsers for both tags and code,

and increases the possible tags from 87 to 350.

The final results were good enough to place us 27th out of 276 in the Kaggle competition. Through careful analysis of our dataset and examining our failure cases we were able to build a reliable tag prediction system for a large and complex dataset. Our feature-based classification approach was validated by the performance of our algorithm, and each of our design decision made both performance and linguistic sense. A major component which we believe contributed to our systems success was the decision to only learn classifiers for a small subset of the tags, this lead to fewer false positives that would likely occur in our model, due to lack of semantic clustering, ie “macosx” and “osx” would receive high confidence scores on the same post, but it is unlikely they should both be used as annotated tags. Our approach by default would only train on the most popular one.

Future Work

While we were able to achieve impressive results with our simple model and carefully examining the given dataset, we think we may be to improve our results in a variety of ways. In the future we would like to further explore ways to enhance tagging through the use of clustering. [4]Heyman et al talks about the three sets which define each document. The set of tags that positively describes the object (Rp). The set of tags which negatively defines the object (Rn). And the set of tags which the user annotated the object with (Ra). Due to the non-limited vocabulary in tagging systems there are several tags which can accurately describe any one post, we would like to perform clustering to be able to get a better sense of the latent set (Rp) and then use our training algorithm on the (Rp) set as opposed to the smaller (Ra) set.

We would also like to further experiment with Parts-Of-Speech Tagging. While we tried to extract only noun phrases and work on them, we think this set may have been too small for our learning algorithms to reach their full potential. We would like to consider the effects of extracting nouns, verbs, and adjectives. We hope that by running thresholding these by their document frequency we may be able to find a smaller subset of words which describe the posts, as well or better than our current implementation.

We would also like to explore Named Entity Recognition as an additional feature. Our intuition is that NER tags would be beneficial in the determining tags for programming languages, mathematical theorems, and various software packages. Successfully identifying Named Entities could help with both learning classifiers and linking related classes.

References

1.http://www.kaggle.com/c/facebook-recruiting-iii-keyword-extraction/forums/t/5605/extended-summary-stats-for-the-data-provided
2. http://infolab.northwestern.edu/media/papers/paper10163.pdf
3.http://static.googleusercontent.com/media/research.google.com/en/us/pubs/archive/41159.pdf
4. http://nlp.stanford.edu/pubs/tagpred-sigir08.pdf

5. http://iccm-conference.org/2013-proceedings/papers/0077/paper0077.pdf