CS229 Term Project

Predicting Tags for URLs Based on Delicious Data

(Jayesh Vyas & Varun Katta)

1. Motivation

Today bookmarking websites like Delicious allow us to bookmark and tag URLs. Tagging helps in labeling, organizing and categorizing URLs, and can also help in discovering new documents for topics of interest. Predicting tags for a document can have the following uses:

- Suggesting tags for a document/URL when a user is attempting to tag as untagged URL.
- By tagging large number of URLs/documents, a topic based browsing system can be built for users to serendipitously discover URLs/content on a topic of their interest.
- Invalidating query cache of a search engine when a new document is indexed - Queries which have the predicted tags of a new document are likely to retrieve that document. So when a new fresh document is indexed by search engines, they can invalidate result caches for queries which have any of the tag terms in them. This is useful for documents going through 'Fresh Documents' pipeline of search engines, which documents are refreshed much faster than time to live duration in results cache.

2. Data Processing

We got the tagged/bookmarked delicious URLs from [1]. This data is in JSON. We wrote processors to convert the Delicious JSON feed of 1 million URLs to a list of a URL to tags mappings. We sorted the tags by frequency, and picked 10 popular tags and about 5000 of these URLs which have **exactly** one of these 10 tags. We wrote a simple crawler and an HTML parser to crawl these URLs and extract text from them. The tags that we picked were: photoshop, art, education, food, research, mac, business, shopping, google, linux.

3. Feature Selection

As with most text classifiers, we stemmed the words using a standard stemmer[2] and discarded the stopwords. Single character tokens were dropped. Very long tokens (of length > 40 characters, e.g. httppagead2googlesyndicationcompageadimgadidx) which crept in because of our primitive HTML parser were also dropped and documents with very few features (< 10 words) were also dropped. After all these steps, we were left with total 17800 features (or stemmed words).

4. Multinomial Naive Bayes

The documents were vectorized and converted into sparse matrix (as in Problem Set 2). We divided the data into a test set of 500 documents, and training data of different sizes (1000, 1500, 2000 ... 5000 documents) to run Multinomial Naive Bayes.

One interesting aspect of this classification problem is that the document classes are soft, and one document may belong to more than one class, although our training data says that a document belongs to only one class. So it might be worthwhile predicting more than one tag for a document and check if it matches with the tag in the test data. The good thing about generative learning algorithms like Naive Bayes is that they build a class model out of training data, and so we can output class probabilities for every class. Thus we can output the most likely tag, the second most likely tag, the third most likely tag and so on.

So for experimentation, we compared the first and the second predicted tag against the tag in training data, and plotted the error. As expected, the fraction of documents for which even the second prediction is wrong is considerably lower than the fraction of documents for which the first prediction is wrong. These errors are plotted against the training data size in the Figure 1 below:

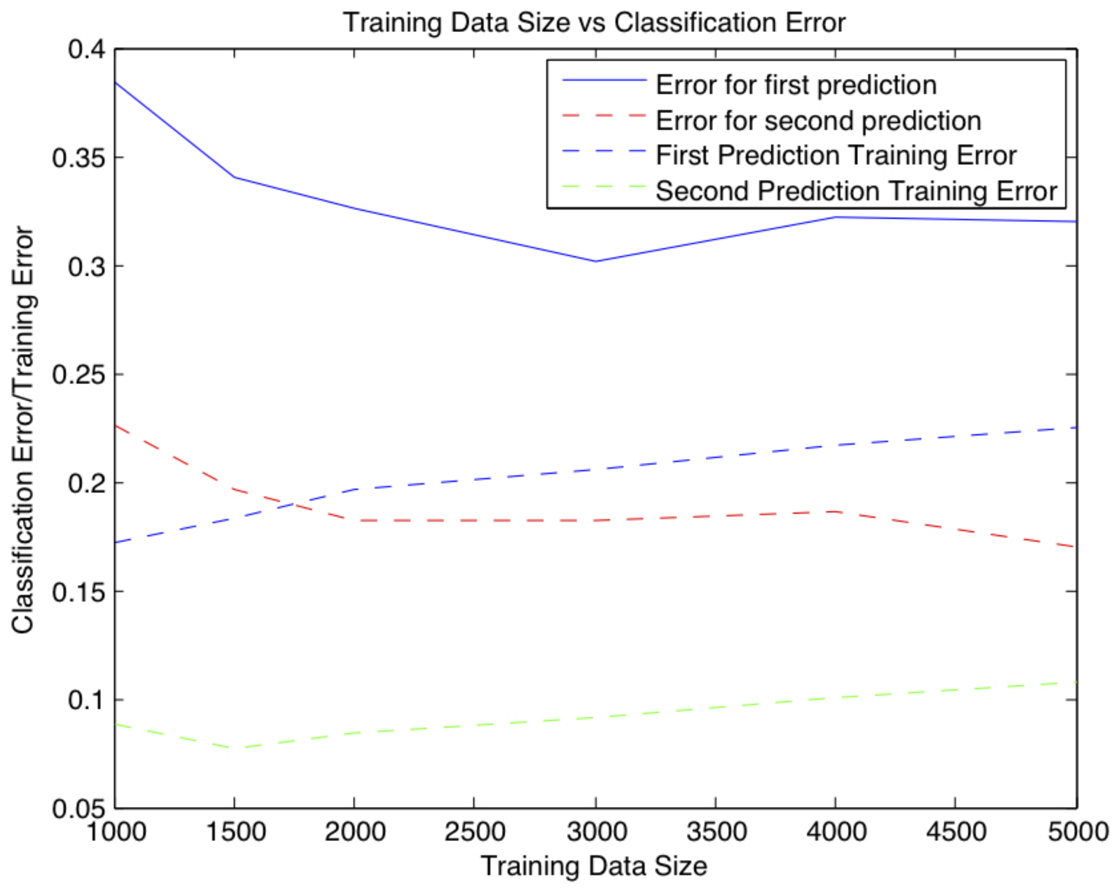

Figure 1: second prediction = Best tag + Next best tag combined for prediction and analysis

Note that “Error for Second Prediction” means that both the first and second prediction for the tag of the test document did not match the actual tag of the document.

Looking at the graph, we suspected that the model could be suffering from high variance as training errors were going down as training size increased and also due to the large gap between training and test errors. We tried the following techniques to improve the performance by increasing the training examples and reduced the number of features.

<u>Improvements:</u>

Increased crawled documents*:*

We increased the crawled documents from 5000 to 11000.

Improved Feature Selection*:*

We improved our parsing to remove features which had no value like spurious occurrences of

javascript and CSS. We built a blacklist of words we could safely drop and removed all features which belonged to the blacklist. This blacklist was based on html tag names, reserved words for javascript, CSS. We also used a whitelist of all valid English words from dictionary[3] and included only those features which belonged this whitelist. We also dropped documents which were not pure HTML like PDFs due to large amounts of meta data in these files . We computed tf-idf scores for words in each document and dropped all terms from a document with tf-idf scores below a certain threshold . We also dropped all features with very low frequency (< 10) This reduced the size of our features from ~17800 to ~7625. We used Multinomial Naive Bayes to build a model based on the new improved feature set and increased the number of training examples.
Figure 2 below shows the the plot of test error as we increased the training set size.

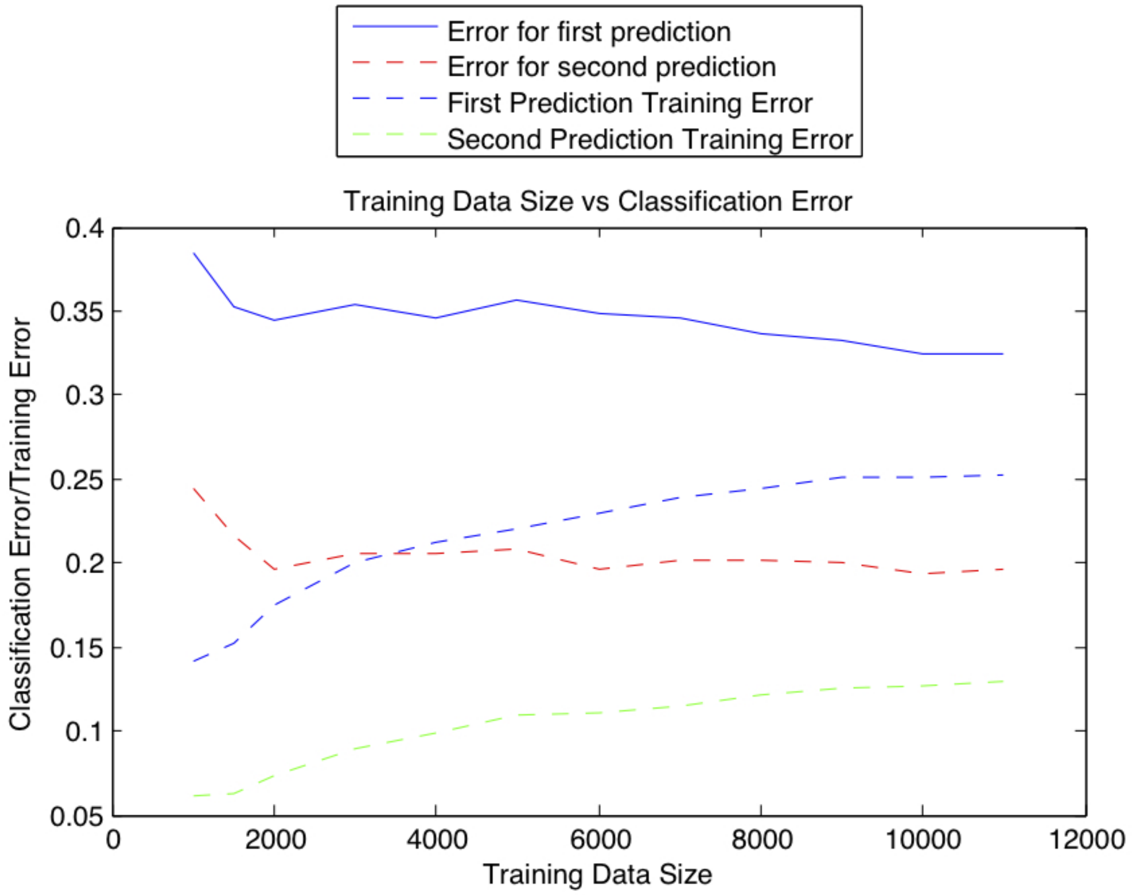

Figure 2

Though there is no perceived difference in the test error, it appears from this exercise, we got rid of all features which were not contributing to any learning. As we keep increasing the training size, the error rate decreases at a very low rate.

<u>Support Vector Machines:</u>

We also experimented with SVM based approach for multi-label classification. We used liblinear[4] for SVM based classification. For SVM training, we used the same improved feature set used for Naive-Bayes previously. We observed that the error rate improved (as excepted) as we increased the training set size. We let the regularization parameter C default to liblinear's default value. We also set the n-fold cross validation mode parameter V to 5 during SVM training exercise. This is was done to reduce the risk of over-fitting during training. See Figure 3 below for the plot of test error against training data size.

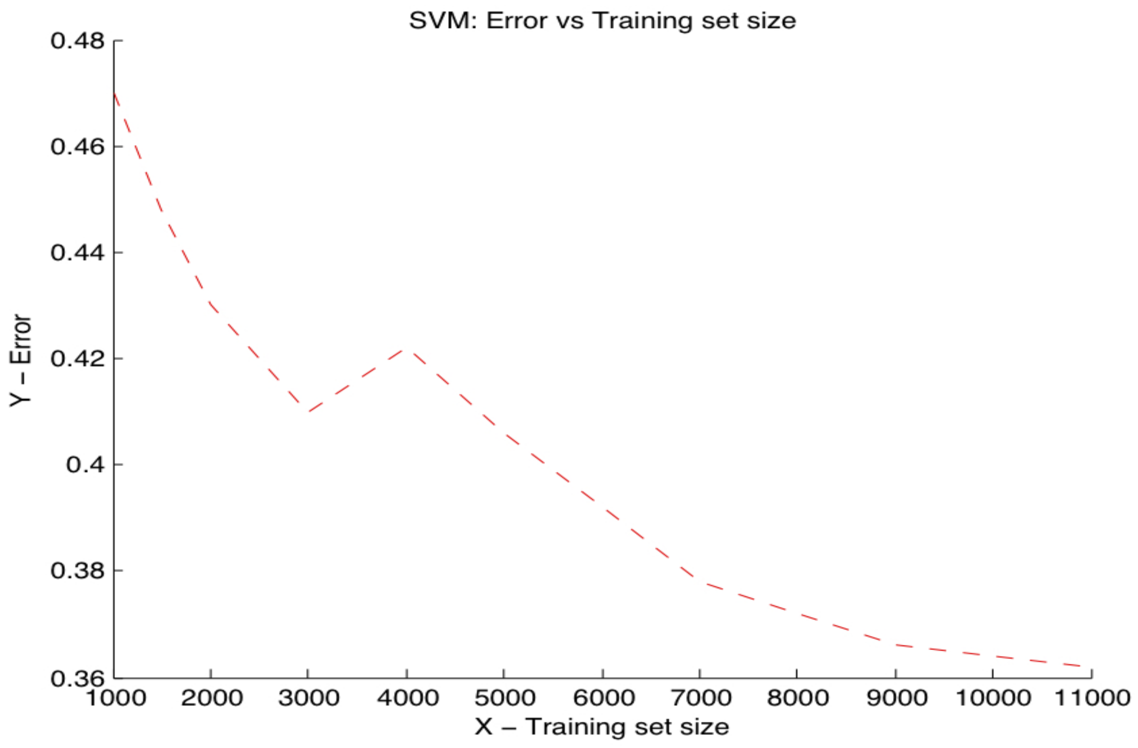

Figure 3

As a further improvement, we checked, if we could improve training error by tweaking the regularization parameter C. In particular, we used 5-fold cross validation mode and varied C from 1 to 10 for a particular training set size. We didn't observe much change in the training error by varying C from 1 to 10. Figure 4 below shows the plot of test error versus the regularization parameter C. Though our search best value of C was not exhaustive, this exercise gave us quick insight into whether we could improve the model during training.

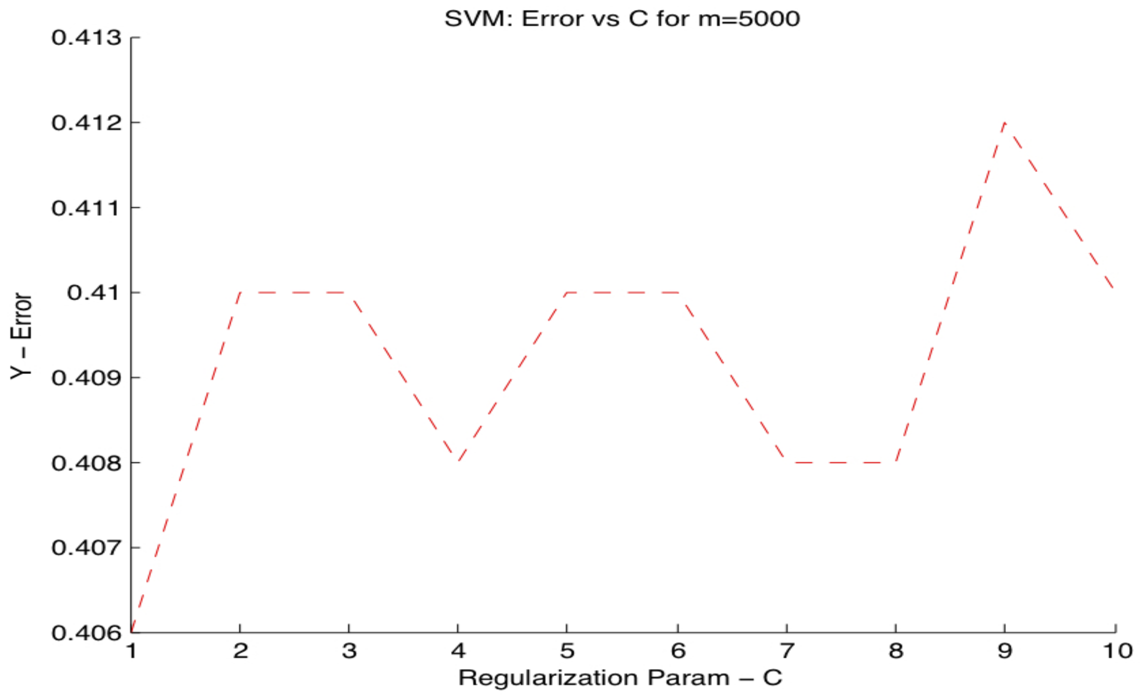

Figure 4

5. Conclusion

We tried to apply text classification techniques to auto-tag delicious URLs. We were intentionally cavalier in picking URLs, and didn't try to curate the URL set to crawl predominantly rich, text only pages. This and the fact that some of the documents could be labeled with more than 1 unique tag (soft-tagging) and noise in data (mis-labeling) could be reasons for lower precision. Combined error rate of best and next-best tag referred above as second prediction has a much lower error rate as shown in Naive-Bayes results above, and is encouraging.

Further Work:

- Extend the model to include larger number of tags (may be 100+).
- Improve the HTML parser to discard useless data and javascript.
- Feature selection: Try feature selection techniques like filter feature selection using mutual information.
- Document selection: Documents with very few tokens are likely to be misclassified (an improved HTML parser might fix this also).

6. References

1. Arvind Narayanan: Dataset of delicious.com bookmarks with 1.25 million entries (http://arvindn.livejournal.com/116137.html)
2. Lingua Stemmer: http://snowhare.com/utilities/modules/lingua-stem/
3. Word dictionaries: http://infochimps.com/collections/moby-project-word-lists
4. Liblinear: http://www.csie.ntu.edu.tw/~cjlin/liblinear/