Articles Tagging

Tuan Nguyen, Rohan Maheshwari, Xuesen Li

I. Introduction

Extracting topics from texts have wide applications on many fields. In user experience design, we can propose a set of tags for user to decide rather than make them type and choose among hundreds of tags. Also, target marketing can make more relevant advertisement if we understand which topics are more relevant to a user. We try to solve this problem by working with the New York Times database.

I. Data

1. Data Source

Newspapers from the historical archive of The New York Times are collected as our source data. The corpus includes metadata provided by The New York Times Newsroom, The New York Times Indexing Service and the online production staff at NYTimes.com. This corpus contains nearly every article published in The New York Times between January 01, 1987 and June 19, 2007. After preprocessing by The New York Times Company, Research and Development, these newspapers are in XML format and contain different fields including, Titles, Article Abstract, Headline, and Body etc. These files also contain news tags which were either produced by Indexing Service (Human labeling) or by Online Producer (Computer Labeling)

2. XML Parsing Methodology

We used lxml library in Python as our main tool for XML parsing. The lxml XML toolkit is a Pythonic binding for the C libraries. It is unique in that it combines the speed and XML feature completeness of these libraries with the simplicity of a native Python API, mostly compatible but superior to the well-known ElementTree API.

Two groups of texts are extracted from each XML articles:

- Content: the main content of the article, containing the Titles, Byline, Article Abstract and Body.
- Tags: only human labellings are selected since human labellings are the target that we are going to compare with. A label is considered as human labelling only when classifier class is indexing_service and classifier type is descriptor.
- Date: date of the newspapers are extracted for organizing purpose

3. Result

In total 253,546 XML files are parsed, in which 87,100 files in 2002, 84,783 files in 2003 and 81,663 files in 2004.

II. Parsing

1. Motivation

Before proceeding the learning process. We need a good algorithm on breaking down an article into a set of significant words. These words are then used as inputs for deciding tags. Significant words would usually be object, people or places. Therefore, we want to extract nouns and names from an article.

2. POS (Part of Speech) tagging

The objective for this module was to read text in some language and assigns parts of speech to each word (and other token), such as noun, verb, adjective, etc., although generally computational applications use more fine-grained POS tags like 'noun-plural'. For this, we first explored and benchmarked certain existing standard implementations, like those available with NLTK. But, as we started implementing our own POS tagger, we realized that perfecting this module is itself more than just a quarter long project. So, after analyzing performance of several existing tools, we picked Stanford NLP Group's Log-linear POS tagger. For analyzing performance we tagged a set of randomly selected articles from our database, and observed the errors in tagged results. After comparing the results, Stanford NLP's tagger was clearly the best choice. But, since most of our existing code was in Python, we chose the forked (https://bitbucket.org/torotoki/corenlp-python) an already forked (https://github.com/dasmith/stanford-corenlp-python) wrapper for the Stanford's Core NLP tools which are coded in Java.

We referred to several papers to get a better understanding of natural language parsing in general, of which most of them were published from the works of Stanford's NLP group. Some of the interesting ones were:

- Christopher D. Manning. 2011. Part-of-Speech Tagging from 97% to 100%: Is It Time for Some Linguistics? In Alexander Gelbukh (ed.), Computational Linguistics and Intelligent Text Processing, 12th International Conference, CICLing 2011, Proceedings, Part I. Lecture Notes in Computer Science 6608, pp. 171--189. Springer
- Kristina Toutanova, Dan Klein, Christopher D. Manning, and Yoram Singer. 2003. Feature-Rich Part-of-Speech Tagging with a Cyclic Dependency Network. *HLT-NAACL 2003*
- Kristina Toutanova and Christopher D. Manning. 2000. Enriching the Knowledge Sources Used in a Maximum Entropy Part-of-Speech Tagger. *Proceedings of the Joint SIGDAT*

3. Frequency distributions

Another part of parsing articles was to figure out how often do some words occur in article writing, and in the process prune them our for better tagging. This also enables us to figure out trending words at different periods of time. For this, we created a redis database with frequency distribution of all words in all the articles in our collected database. The choice of redis was preferred over any sql based alternative, as key-value stores are way faster than any relational database alternatives. This will help in

substantially increasing our performance in the future (and present!) when we want to quickly retrieve/update the frequency distributions of millions (soon billions!) of words!

4. Word to vector representations

This is ongoing work where we want to explore different ways to represent words as vectors in 2-D space, so as to easily compute differences or similarities between different words. Many current NLP systems and techniques treat words as atomic units - there is no notion of similarity between words, as these are represented as indices in a vocabulary. This choice has several good reasons like simplicity, robustness and the observation that simple models trained on huge amounts of data outperform complex systems trained on less data. But, in our case the similarities or differences between words (also, at multiple degrees) cam prove to be really beneficial when tagging the entire article with a tiny subset of related words. For this we referred to the paper on Efficient Estimation of Word Representations in Vector Space by Tomas Mikolov, Greg Corrado, Kai Chen and Jeffery Dean (http://arxiv.org/pdf/1301.3781.pdf). We are still trying to develop an efficient implementation of their algorithm in python for our project.

III. Learning

1. Evaluation

Before implementing an algorithm is to propose a way of measuring error for the purpose of model evaluation and comparison. Given a set of tags from a single article, we want our predicted set to be close to that set. Here, there are two types of error. One is predicting tags that are not in the article's set of tag. The other is not predicting tags that are actually in the article's set of tag. We define the score function between predicted tags and data tags as:

$$Score = \frac{2 * (P * R)}{P + R}$$

Where P is precision and R is recall. This score is derived from $1/(\frac{1}{2*P} + \frac{1}{2*R})$. For small precision, $\frac{1}{P}$ is large; and for small recall, $\frac{1}{R}$ is large. A small precision means that our algorithms overshoot with tags prediction; it predicts a lot of tags that are not in the data outputs. While a small recall shows that our algorithm are too conservative. We miss a lot of tags that are in the data set outputs. By increasing the score, we are trying to play the algorithm so that its prediction is accurate but not too conservative so that it doesn't miss so many tags in the data set.

Also, we would like to claim that our actual performance of an algorithm is always better than its maximal scores. Since there are tags that are similar in meaning but we dis-allow it in the measurement. For example, the article's tag is "Theatre", but our prediction is "Motion Pictures". In reality, we would consider this as reliable tagging, but our measurement excludes this circumstance for precise measurement.

2. Algorithm: tag-switch learning

Our learning algorithm is simply generating a sequence of individual tag switching. Given an article, we put it against each tag in our universe set of tags. For each tag, we predict a 1 or 0 depend on the content of the article and our algorithm. After running through the universe of tags, we obtain our predicted set of tags with tags that appeared 1 after the process.

For every tag, we use the same learning algorithm for prediction. Our first three candidates are Naïve Bayes, LASSO Logistic and SVM. An article will be decomposed and filtered into a set of words. This parsing algorithm is used as discussed previously in this report. Before any training process, we would like to build a universe set of tags and set of words. For tags, we only select tags that appear more than a certain threshold. For words, we want to keep words that are not too rare and not usual. On top of that, we only want to exclude words that don't have good correlation with any other tag. These are considered non-significant words. After building the universe sets, we are ready for training and testing.

Before running machine learning for each tag, we also need to decompose an article into set of words, and the dictionary of words is our feature. We use 3 types of parsing for this process.

- Word by word: we take each word after a space
- Compound parsing: we take meaningful 2-grams as additional feature. For example, "film producer" will be taken as feature instead of "film" and "producer"
- Nouns: we takes nouns only from an article

3. Results

We trained on 2000 articles within the year 2002. We keep tags that appear at least 25 times throughout the training set. Words that appear less than 15 times and more than 1000 times are excluded from the set. We train on 1600 articles and test on 400 others. Here is the F-score reported on the test set:

	Simple Parsing	Compound Parsing	Nouns Only
Naïve Bayes	0.3769	0.3502	0.2321
SVM	0.6635	0.6776	0.6561
LASSO Logistic	0.6551	0.6979	0.6554

IV. Conclusion

By observing the result, we come to have certain conclusions:

- Robust learning method such as SVM and L1-regularization are powerful
- Knowing nouns is reasonably enough for predicting content of an article. This relates to questions of Who, Where and What in journalism.
- However, for industry use where speed is necessary, we don't need advanced parsing. With simple parsing, we can have reasonable performance.