Journal of Computer and Communications, 2014, 2,157-162
Published Online March 2014 in SciRes. http://www.scirp.org/journal/jcc
http://dx.doi.org/10.4236/jcc.2014.24021

Tagging Accuracy Analysis on Part-of-Speech Taggers

Semih Yumusak[1], Erdogan Dogdu[2], Halife Kodaz[3]

[1]Computer Engineering Department, KTO Karatay University, Konya, Turkey
[2]Computer Engineering Department, TOBB University of Economics and Technology, Ankara, Turkey
[3]Computer Engineering Department, Selcuk University, Konya, Turkey
Email: semih.yumusak@karatay.edu.tr, edogdu@etu.edu.tr, hkodaz@selcuk.edu.tr

Received November 2013

Abstract

Part of Speech (POS) Tagging can be applied by several tools and several programming languages. This work focuses on the Natural Language Toolkit (NLTK) library in the Python environment and the gold standard corpora installable. The corpora and tagging methods are analyzed and compared by using the Python language. Different taggers are analyzed according to their tagging accuracies with data from three different corpora. In this study, we have analyzed Brown, Penn Treebank and NPS Chat corpuses. The taggers we have used for the analysis are; default tagger, regex tagger, n-gram taggers. We have applied all taggers to these three corpuses, resultantly we have shown that whereas Unigram tagger does the best tagging in all corpora, the combination of taggers does better if it is correctly ordered. Additionally, we have seen that NPS Chat Corpus gives different accuracy results than the other two corpuses.

Keywords

POS Tagger; Brown Corpus; Penn Treebank Corpus; NPS Chat Corpus

1. Introduction

Part-of-speech (POS) tagging is a process of classifying the words in a sentence according to their types [1-3]. POS taggers are used for different purposes. POS tagging can be used for Linguistic-text Pre-processing before semantic analysis [4,5]. POS tagging was also used for multi-lingual text analysis [6]. The tag sets and the terminology of tagging are defined by different projects such as Brown Corpus Tag-Set, Penn Treebank Tag-Set [7]. In Python programming language there is an infrastructure about natural language processing, which is called Natural Language Toolkit (NLTK). NLTK 3.0 is a code library that is used to build Natural Language Programming (NLP) in Python [8,9]. In the Python development environment, there are several tagged corpora available for installation. The available corpus names are listed by using nltk. download () method. However, not all of the corpora listed by this method are tagged. In this paper, we will be dealing with tagged corpora, *i.e.* which includes part-of-speech annotations. The tagged corpora installed for this work in the Python NLTK library are as follows; Brown Corpus [10] Penn Treebank Corpus [7] NPS-Chat Corpus [11]. In this study, we have more deeply examined the different corpuses and taggers available in NLTK and mentioned in [8]. We have first listed the methods and data used for analyzing the taggers. The Python source code for analyzing the Penn Treebank corpus is written in method and analysis section to show the exact way of how we made the analysis. While writing the code for analysis, we have inspired from the code samples used in [8]. Finally, all corpora are analyzed by using the same tagging methods and the accuracies are listed in **Table 1**.

The rest of the paper is organized as follows. Section 2 presents explanation of the materials and the method we used. Section 3 describes details of our application and reports experimental results. Finally section 4 concludes the paper.

2. Materials and Methods

NLTK 3.0 (NLTK) is a code library that is used to build NLP programs in Python. The features in the NLTK library are; sentence and word tokenization, part-of speech tagging, chunking and named entity recognition, text classification [8,9]. There are also several corpora available in NLTK library to use for data analysis. We explored and imported these corpora by using the following method, by replacing * with the name of the corpus;

*>>> from nltk.corpus import**

We have found that not all of the corpora installable are tagged. Then we filtered the tagged corpora as if the tagged words() method is reachable in a corpus, we assumed it as a tagged corpus. The assumption is made according to the NLTK web site, which is written as tagged words() method is only supported by corpora that include part-of-speech annotations [12]. As a result of this filtering, we have found that Brown Corpus, Penn Treebank Corpus and NPS Chat Corpus are available with their tagged sentences. After all, we have applied Default Tagger, Regex Tagger, Bigram Tagger, Unigram Tagger, Trigram Tagger to these three corpuses. The source code used for corpus exploration and the POS tagger applications are explained by using the Penn Treebank corpus in the following sections.

2.1. Corpus Exploration

The Python command used for checking part-of speech annotations is as follows:

>>> tagged = treebank.tagged_words()

>>> tagged [(‘*Pierre*’, ‘*NNP*’), (‘*Vinken*’, ‘*NNP*’), (‘,’, ‘,’), ...]

In order to make evaluations by using different taggers, we needed to get the tagged sentences and non-tagged sentences for tagging and evaluation. We have used tagged sents() method to get the tagged sentences for evaluation, and sents() method to get the original sentences for tagging. The sample code we used for getting the tagged and nontagged sentences are as follows;

>>> treebank_tagged_sents = treebank.tagged_sents()

>>> treebank_tagged_sents [[(‘*Pierre*’, ‘*NNP*’), (‘*Vinken*’, ‘*NNP*’), (‘,’, ‘,’), (‘61’, ‘*CD*’), (‘*years*’, ‘*NNS*’), (‘*old*’, ‘*JJ*’), (‘,’, ‘,’), ...]

>>> treebank_sents = treebank.sents()

>>> treebank_sents [[‘*Pierre*’, ‘*Vinken*’, ‘,’, ‘61’, ‘*years*’, ‘*old*’, ‘,’, ‘*will*’, ‘*join*’, ‘*the*’, ‘*board*’, ‘*as*’, ‘*a*’, ‘*nonexecutive*’, ‘*director*’, ...]

2.2. Application of POS Taggers

Since we have the tagged and non-tagged versions of the same text, we are able to evaluate the correctness of a tagger. There are several taggers in Phyton NLTK library, which we used to tag and evaluate the tagger algorithm. The taggers available in the NLTK library can be explained as follows;

Default Tagger: Applied with assigning NN (Noun) tag to each token. It basically gives the percentage of nouns, which is most likely to occur in a sentence.

Regex Tagger: The regular expression tagger is initialized with a pattern array, and tags according to that pattern array.

Unigram Tagger: It assigns a tag to a word which is most likely to occur. More specifically, it trains with a training data and calculates the occurrences of the words, then tags the test data according to the occurrence statistics in the training data.

Bigram Tagger: It applies statistically the same procedure as the unigram tagger with one specific difference.

It checks for the occurrences of the words together with one word before.

Trigram Tagger: It applies statistically the same procedure as the unigram tagger with one specific difference. It checks for the occurrences of the words together with two words before. As it is shown in **Figure 1**, the n-gram tagging is explained by using tri-gram representation as; the tag to be chosen, t_n, is circled, and the context is shaded in grey. We have n = 3; that is, we consider the tags of the two preceding words in addition to the current word. An n-gram tagger picks the tag that is most likely in the given context.

2.3. Unsupervised Taggers

In order to evaluate the taggers on the treebank corpus, we first started with un-supervised taggers. In the program code below, we applied the most frequent tag, *i.e*. the NN tag to the default tagger. It shows that 13 percent of the tags are nouns for the treebank corpus.

```
>>> defaultTagger = nltk.DefaultTagger(‘NN’)
>>> evalResult = defaultTagger.evaluate (treebank.tagged_sents())
>>> print ‘Accuracy is: %4.2f %%’ % (100.0 * evalResult)
Accuracy is: 13.08%
```

Secondly, we improved the Regular expression template given in [8] and formed a new pattern as follows:

```
patterns = [
(r‘.*ing$’, ‘VBG’),
(r‘.*ed$’, ‘VBD’),
(r‘.*es$’, ‘VBZ’),
(r‘.*ould$’, ‘MD’),
(r‘.*\’s$’, ‘NN$’),
(r‘.*s$’, ‘NNS’),
(r‘(The|the|A|a|An|an)$’, ‘AT’),
(r‘.*able$’, ‘JJ’),
(r‘.*ly$’, ‘RB’),
(r‘.*s$’, ‘NNS’),
(r‘.*’, ‘NN’)]
```

The regular expression tagger is applied as follows:

```
>>> regexTagger = nltk.RegexpTagger(patterns)
>>> evalResult = regexTagger.evaluate(treebank.tagged_sents())
>>> print ‘Accuracy is: %4.2f %%’ % (100.0 * evalResult)
Accuracy is: 20.52%
```

Regular expression tagger by itself is limited to very common language properties; therefore it is able to tag only the 20.52% of the whole corpus correctly.

2.4. Supervised Taggers

There are also some supervised taggers available in the standard installation of Python environment. We applied 3 step n-gram taggers in order to analyze their tagger performance.

First of all, unigram tagger is analyzed. Unigram tagger is designed as a simple statistical algorithm. It assigns the tags with the most probable tag by calculating the frequencies of each token [8]. In the first step it trains the data, and then tests it with a different data. Unigram Tagger is applied to the treebank corpus with the Python code below:

```
>>> corpusSize = int(len(treebank.tagged_sents()) * 0.8)
```

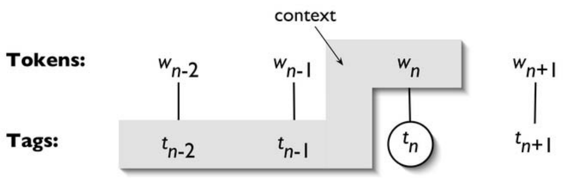

Figure 1. N-gram tagging [8].

```
>>> trainingSents = treebank.tagged_sents()[:corpusSize]
>>> testSents = treebank.tagged_sents()[corpusSize:]
>>> unigram_tagger = nltk.UnigramTagger(trainingSents)
>>> evalResult = unigram_tagger.evaluate (testSents)
>>> print 'Accuracy is: %4.2f %%' % (100.0 * evalResult)
Accuracy is: 86.25%
```

In this code, we selected 80% of the treebank corpus as the training data, 20% of the corpus as the test data. The accuracy of the tagger is 86, 25% for the treebank corpus.

Secondly, we looked for the multiple n-gram taggers. The difference between unigram and n-gram is explained in [8] Natural Language Processing with Python as follows; when we perform a language processing task based on unigrams, we are using one item of context. In the case of tagging, we consider only the current token, in isolation from any larger context. Given such a model, the best we can do is tag each word with its a priori most likely tag. This means we would tag a word such as wind with the same tag, regardless of whether it appears in the context the wind or to wind. An n-gram tagger is a generalization of a unigram tagger whose context is the current word together with the part-of-speech tags of the n-1 preceding tokens.

The bigram tagger is applied similar to the unigram tagger, the same training and test sets are used.

```
>>> bigramTagger = nltk.BigramTagger(trainingSents)
>>> evalResult = bigramTagger.evaluate(testSents)
>>> print 'Accuracy is: %4.2f %%' % (100.0 * evalResult)
Accuracy is: 13.55%
```

The accuracy of the tagger decreases dramatically according to the unigram tagger. This decrease is explained in [8] as follows; Notice that the bigram tagger manages to tag every word in a sentence it saw during training, but does badly on an unseen sentence. As soon as it encounters a new word (*i.e.*, 13.5), it is unable to assign a tag. It cannot tag the following word (*i.e.*, million), even if it was seen during training, simply because it never saw it during training with none tag on the previous word. Consequently, the tagger fails to tag the rest of the sentence.

As a third step, we applied the trigram tagger to treebank corpus. As it is seen in the following code, the same training and test sets are applied.

```
>>> TrigramTagger = nltk.TrigramTagger(trainingSents)
>>> evalResult = TrigramTagger.evaluate(testSents)
>>> print 'Accuracy is: %4.2f %%' % (100.0 * evalResult)
Accuracy is: 67.06%
```

The accuracy of the trigram tagger is better than the bigram tagger, however it is low for an application on a gold standard corpus. Thus, we applied all the taggers by combining them in an order as follows:

```
>>> defaultTagger = nltk.DefaultTagger('NN')
>>> regexpTagger = nltk.RegexpTagger(patterns, backoff = defaultTagger)
>>> unigramTagger = nltk.UnigramTagger(trainingSents, backoff = regexpTagger)
>>> bigramTagger = nltk.BigramTagger(trainingSents, backoff = unigramTagger)
>>> trigramTagger = nltk.TrigramTagger(trainingSents, backoff = bigramTagger)
>>> evalResult = trigramTagger.evaluate(testSents)
>>> print 'Accuracy is: %4.2f %%' % (100.0 * evalResult)
Accuracy is: 89.56%
```

This code does the tagging with this order; firstly look for the trigram tagger, then bigram, unigram, regex, lastly the default tagger is used for the non-tagged ones. Consequently, the result becomes better than all of the above taggers used.

3. Experimental Results

In the evaluation section, all types of taggers used in this work are applied to three corpora, which are the Brown Corpus, Penn Treebank Corpus and the NPS Chat Corpus. The combination of taggers is selected by combining taggers by every possible order starting with default and regex taggers. In **Figure 2** horizontal axis represents the combination of taggers. For instance, drubt means; default, regex, unigram, bigram, trigram taggers applied respectively. The vertical axis is the accuracy percentage. As it is seen in **Figure 2**, the best accuracy result comes with drubt ordering with a small difference from the drutb combination. Thus, we used drubt ordering for combination of taggers accuracy calculation. The results are listed in **Table 1**.

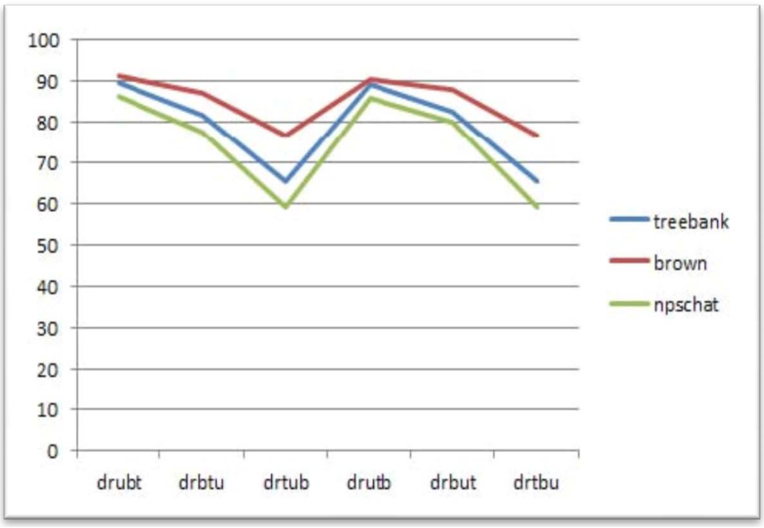

Figure 2. Combination of taggers accuracy.

Table 1. NLTK tagging accuracies.

Corpus Name	Default (%)	Regex (%)	Unigram (%)	Bigram (%)	Trigram (%)	Comb. of Taggers (%)
Brown Corpus	10.99	25.78	87.70	33.88	19.20	91.33
Treebank Corpus	14.47	21.75	86.25	11.33	6.7	89.56
NPS Chat Corpus	9.11	12.69	80.85	43.61	35.91	86.45

As seen in **Table 1**, Unigram Tagger performs better than the other taggers when applied alone on all three corpus. But the combinations of taggers' results are even better. Whereas trigram and bigram taggers give lower results in accuracy, its recall is most probably higher than other taggers. Although we are not able to calculate the recall value, we can infer the probability of getting two(bigram) words together in different tags from one sentence to another is logically lower than one(unigram) word. Thus, if we use the most precise (trigram) tagger first, then continue with more generalized taggers as bigram, unigram, regex, default taggers respectively; the accuracy becomes higher than any other individual tagger. When we look at the results vertically for the corpuses, the NPS chat corpus is tagged better than the other ones with the bigram and trigram taggers. Since a person use a limited vocabulary in a chat conversation and use usually similar patterns, the accuracies in bigram and trigram taggers seems reasonable. However, in the treebank and brown corpora, the trigram tagger does the worst, even worse from the default tagger in treebank corpus. This shows us that the variety of English words and phrases are higher than the NPS Chat corpus. Whereas, n-gram taggers in general give more accurate results than the others applied, bigram and trigram taggers need more data to train in order to give better results for the Penn Treebank and Brown Corpora.

4. Conclusion

In this work, part of speech tagging is analyzed by improving the usage of different techniques and different corpora. The differences between each corpus are analyzed according to their tagging accuracies. Whereas Brown and Treebank corpora are similar according to their accuracy values, NPS Chat corpus differs significantly. Consequently, NPS Chat corpus seems as a good reference for non standard internet data and the results might be generalized for the free text. For the future work, more advanced taggers will be analyzed such as; Brill Tagger, Hunpos Tagger and Stanford Tagger. Whereas Brill and Hunpos taggers are available in the python environment, Stanford tagger needs a reference for the tagger java file for tagging. Moreover, some more improvements will be made for the regular expression tagger by using more complex patterns. Finally, some Turkish Corpus will be applied for these taggers and be analyzed according to their accuracies.

Acknowledgements

This study has been supported by Scientific Research Project of Selcuk University.

References

[1] Brants, T. (2006) Part-of-Speech Tagging. *Encyclopedia of Language & Linguistics* (*Second Edition*), Elsevier, Oxford, 221-230.

[2] Søgaard, A. (2010) Simple Semi-Supervised Training of Part-of-Speech Taggers. *Proceedings of the ACL* 2010 *Conference Short Papers*, Uppsala, 11-16 July 2006, 205-208.

[3] Das, D. and Petrov, S. (2011) Unsupervised Part-of-Speech Tagging with Bilingual Graph-Based Projections. *HLT*'11 *Proceedings of the* 49*th Annual Meeting of the Association for Computational Linguistics*: *Human Language Technologies*, Association for Computational Linguistics, Portland, 19-24 June 2011, 600-609.

[4] Ittoo, A. and Bouma, G. (2013) Term Extraction from Sparse, Ungrammatical Domain-Specific Documents. *Expert Systems with Applications*, **40**, 2530-2540.

[5] Demner-Fushman, D., Chapman, W.W. and McDonald, C.J. (2009) What Can Natural Language Processing Do for Clinical Decision Support? *Journal of Biomedical Informatics*, **42**, 760-772.

[6] Nothman, J., Ringland, N., Radford, W., Murphy, T. and Curran, J.R. (2013) Learning Multilingual Named Entity Recognition from Wikipedia. *Artificial Intelligence*, **194**, 151-175.

[7] Marcus, M.P., Santorini, B. and Marcinkiewicz, M.A. (1993) Building a Large Annotated Corpus of English: The Penn Treebank. *Computational Linguistics*, **19**, pp. 313-330.

[8] Bird, S., Klein, E. and Loper, E. (2009) Natural Language Processing with Python. OReilly Media, USA.

[9] NLTK 3.0 Documentation. http://www.nltk.org/

[10] Brown Corpus Manual. http://icame.uib.no/brown/bcm.html

[11] The NPS Chat Corpus. http://faculty.nps.edu/cmartell/NPSChat.htm

[12] Coprus Readers-Tagged Corpora. http://nltk.googlecode.com/svn/trunk/doc/howto/corpus.html#tagged-corpora