Slovene NLTK Tagger

Niko Colnerič
Faculty of Computer and
Information Science
University of Ljubljana
niko.colneric@gmail.com

Nejc Banič
Faculty of Computer and
Information Science
University of Ljubljana
banic.nejc@gmail.com

ABSTRACT

This paper describes the process of building NLTK tagger for a custom language. NLTK stands for Natural Language Toolkit, a library for use in Python (version 2.7). Tagger is an object, which processes a sequence of words and attaches a part of speech tag to each word. Our implementation language was Slovene. We constructed sequential *Trigram tagger*, *Brill tagger* and classifier based *Naive Bayes tagger*, which differ in tagging accuracy and time consumed for tagging. All this differences are also detaily compared. We recommend using the Brill tagger.

Keywords: Tagger, corpus, training, tagging accuracy, time consumed

I. INTRODUCTION

Part-of-speech tagging is harder than just having a list of words and their parts of speech, because some words can represent more than one part of speech at different times and because some parts of speech are complex or unspoken. This is not rare in natural languages (as opposed to many artificial languages), a large percentage of word-forms are ambiguous. Native speakers of a language perform grammatical and semantic analysis innately, and thus trained linguists can identify the grammatical parts of speech to various fine degrees depending on the tagging system [9]. Although the problem is not trivial for a computer, the taggs are useful categories for many language processing tasks.

In this chapter, we provide a brief theoretical description of implemented taggers. Next we will describe the tools used in section II, which were the base of our project. In section III the process of implementation itself is explained. The example, how to use the tagger is provided, along with some discussion on encoding problems. Finally in section IV is the evaluation of accuracy and speed.

Now we begin with the description of implemented taggers.

A. Trigram tagger

To understand how *trigram* works, we should first understand *unigram*. Unigram taggers are based on a simple statistical algorithm: *for each token, assign the tag that is most likely for that particular token*. In the case of tagging with Unigram tagger, we only consider the current token, in isolation from any larger context. The best we can do is tag each word with its *a priori* most likely tag.

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, as shown in figure 1. Unigram could also be refereed to as 1-gram.

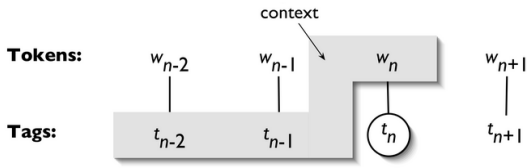

Fig. 1. Tagger Context

The tag to be chosen, t_n, is circled, and the context is shaded in grey. In the example of an n-gram tagger shown in figure 1, 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 a priori most likely tag in the given context.

Although we use the name trigram throughout this paper, we actually refer to sequential tagger, which consists of *affix*, *unigram*, *bigram* and *trigram* taggers in this order respectively. The Affix tagger learns prefix and suffix patterns to determine the part of speech tag for a word. First we build affix, which is used as a *backoff-tagger* for unigram. Unigram is afterwards used as a backoff-tagger for bigram, which is finally used for trigram.

Definitions and images in this section are summarized from [5].

B. Brill tagger

The Brill tagger is another part-of-speech tagger and was first described by Eric Brill in 1993. Before trying to explain it, we must address issues with n-gram taggers (e.g. *Trigram tagger*).

First issue with n-gram taggers is size of their n-gram table. If we want to use tagging process on mobile computing device, compromise must be taken between performance and model size.

A second issue with n-gram tagger is, that the only information it considers from prior context is tags. This is not practical because words themselves are useful information also. Therefore n-gram taggers are impractical models to be used to identify words in context.

To bypass those issues we can use *Brill tagger*. It can be summarised as an transformation-based learning and performs

very well using models that are just a fraction of the size of n-gram taggers.

Despite obvious differences between Brill tagger and n-gram tagger there is one similarity. Both of them use *supervised learning* method, since we require annotated training data to make sure whether the taggers guess is a mistake or not.

The idea behind the Brill tagger is: *guess the tag of each word, then go back and fix the mistakes*. Basically, Brill tagger can transform bad tagging into a better one.

C. Naive Bayes classifier[1]

Both Brill and Trigram are sequential backoff taggers. Meanwhile, Naive Bayes is *classifier based tagger*. Classification is a process where, for a given output, we have to choose the correct class. Basically, it is an procedure for assigning input data into one of a given number of categories.

Examples of classification tasks are:

- whether an email is spam or not.
- diagnosis to a given patient as described by observed characteristics of the patient.

The basic classification task has a number of interesting variants. For example, in multi-class classification, each instance may be assigned multiple labels; in open-class classification, the set of labels is not defined in advance; and in sequence classification, a list of inputs are jointly classified.

Every feature in naive Bayes classifier can determine which label can be assigned to a given input. Choosing a label for an input is a process, that consists of calculating the prior probability of each label (it is determined by calculating frequency in training set). Priori probability is combined with contribution from each feature. This is necessary to get the likelihood estimate for each label. Input value is then assigned the label whose likelihood estimate is the highest. The process is shown in 2.

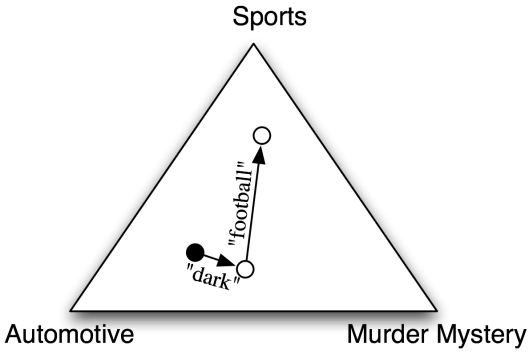

Fig. 2. An illustration of the procedure used by the naive Bayes classifier to choose the topic for a document.

Let's assume that in the training corpus most documents have automotive topic. Because of this assumption, the classifier starts near "automotive" label. Later it must take into consideration the effect of each feature. Example considers, that input document contains word "dark", which indicates for "murder mystery", but also contains the word "football", which (strongly) indicate for "sports". After every feature has made its contribution, the classifier checks which label it is closest to, and assigns that label to the input.

Features make their contribution to the decision by voting against those labels, that don't occur with this feature. Likelihood score is therefore reduced for each label by multiplying it with the probability, that an input value with that label would have that feature. Suppose that the word *ran* occurs in 12% of the sports documents, 10% of the murder mystery documents, and 2% of the automotive documents. The likelihood score for the sports label will be multiplied by 0.12, the likelihood score for the murder mystery label will be multiplied by 0.1, and the likelihood score for the automotive label will be multiplied by 0.02. The overall effect will be, to slightly reduce the score of the sports label, reduce the murder mystery label (more then sports label) and significantly reduce the automotive label. Figure 3 illustrates this example.

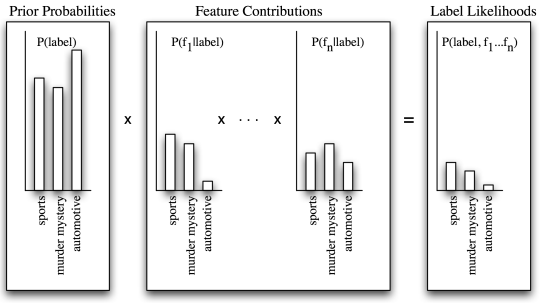

Fig. 3. Illustrating the process of calculating label likelihoods with naive Bayes.

II. Tools used

A. Nltk-trainer

The code for building the tagger was taken from *Nltk-trainer* project [3], whose author is Jacob Perkins. Its an open-source project hosting on Github, with moto to *Train NLTK objects with zero code*. We almost exclusively used the script `train_tagger.py`, which has the ability to generate NLTK taggers. This project made it a lot easier for us to train the taggers on Slovenian corpora.

B. JOS corpus

The basic component for building Slovene tagger is *JOS corpus* from JOS project [2]. It stands for: *Jezikoslovno Označevanje Slovenskega jezika* (Linguistic annotation of Slovenian language). It contains collections of various text in Slovenian language with annotation. For our project we used jos1M corpus, that contain 1 million words with it's appropriate lemmas and morphosyntactic descriptions. The JOS is compatible with Slovene *MULTEXT-East morphosyntactic specifications Version 4*[4]. It basic purpose is to define word

[1]Definition and images summarized from [5]

classes (i.e. part-of-speech). For each class there are various attributes and their appropriate values, which they can be mapped into morphosyntactic descriptions (i.e. MSD). The structure of JOS corpora is in *Extensible Markup Language (XML)*.

III. Implementation

A. Corpus transformation

We can easily manipulate XML files in various programming languages (e.g. Python with xml.dom.minidom). This is mandatory, because *nltk-trainer*[3] expects special form of input file (.pos file).

The first step for corpus transformation is to identify proper *XML tags*[6] in JOS corpora. There are several types, but we must parse only those in *sentence tag* (<*s*>):

- <*S*> - space.
- <*c*> - character i.e. punctuation marks (e.g. period (.), comma (,), colon (:), ...).
- <*w*> - word.
- <*term*> - a word or group of words designating something, especially in a particular field.

For nltk-trainer to work, we must transform `jos1m.xml` to `.pos` file, which contains words with their morphosyntactic descriptions and punctuation marks. The correct syntax is shown in the next example:

```
Britanski/Ppnmeid BBC/Slmei ,/,  brez/Dr
```

The trainer will train the tagger properly, if .pos file consists off:

- The words and their MSD are separated with slash.
- Every punctuation mark is separated with slash and then follows the same punctuation mark.
- Words and punctuation marks are between space.

If we follow these "rules", implementation is simple. We just need to go through all child tags in sentence tag and properly write them in a file. Because Slovenian language contain letters with caron, proper encoding of tags must be provided. This is simply done with function *encode()* (e.g. encode("utf-8") - this is proper encoding for Slovenian language). *Jos1M* corpus is build up with 10 XML files (e.g. `jos1M-01.xml`, `jos1M-02.xml`, ..., `jos1M-10.xml`), so we must parse all of them. File `transformJOS.py` contains the corpus transformation.

B. Training the tagger

The script `train_tagger.py` was used for this purpose. Its mandatory argument is a corpus in .pos format. There are many optional arguments. We will describe only the basic ones, for further description run script help with:

```
train_tagger.py --help
```

To generate a Sequential Tagger we use:

- a: AffixTagger
- u: UnigramTagger
- b: BigramTagger
- t: TrigramTagger

We can use any combination of the these letters, to select a sequential backoff algorithm. The default is *aubt*, which was used for our Trigram tagger, but you can set this to the empty string not to train a sequential backoff tagger. The Brill tagger is trained before the other taggers, which in our implementation was the Trigram tagger. When building a classifier based algorithm, we must select the classifier or the sequence of classifier. Following are supported: NaiveBayes, DecisionTree, Maxent, GIS, IIS, CG, BFGS, Powell, LBFGSB, Nelder-Mead, MEGAM and TADM. The time to train classifier based tagger, is much larger than, the time for a sequential or Brill.

Script also has the possibility to set a default tag. We used *-Neznan-*, which is Slovene word for *-unknown-*.

We also choose whether to evaluate the tagger or not and the percentage of sentences to be used. The default value is 1, which means that the *same* sentences are first used for training the tagger and then for evaluation. We recommend that this argument is <1. In this case, sentences are divided into two groups. The first in used for training and the second for evaluation. *Fraction* is the percentage of sentences assigned to training group. This was used for evaluating constructed taggers. For results of this evaluation see section IV.

C. Usage with NLTK

We can use the trained tagger and NLTK to see how they can be used in Slovenian language. The requirements are installed NLTK and Python 2.7. We set an example in file `example.py`. First we need to import following functions:

- *pickle* - it is a module for serialization and de-serialization a Python object structure ("pickling" - Python object hierarchy is converted into a byte stream). Pickle is used for loading classifiers such as Trigram tagger, Brill tagger, etc.

```
tagger = pickle.load( open("
    slovene_taggers/BrillTagger.pickle")
     )
```

- *PunktWordTokenizer* [7] - it is just one of many NLTK's tokenizers, that can be used to divide strings into lists of substring. Tokenizers are subclasses of *nltk.tokenizer.TokenizerI* interface, which defines the *tokenizer()* method. PunktWordTokenizer uses regular expressions to divide a word into tokens. These tokens are then used by the tagger. Following example will demonstrate, how does PunktWordTokenizer tokenize a sentence. If we have the following sentence:
  Potem se je obrnil proti nam in skozi solze izoblikoval najlepši nasmeh, kar sem jih videl v življenju.
  the tokenizer will output the following list of tokens:
  [*'Potem', 'se', 'je', 'obrnil', 'proti', 'nam', 'in', 'skozi', 'solze', 'izoblikoval', 'najlepši', 'nasmeh', ',', 'kar', 'sem', 'jih', 'videl', 'v', 'življenju.'*]
  As you can see, there is a problem with the last token. The solution we used is described below.

- *nltk.data* - it is a module that contains various functions to load NLTK files, such as corpora. We use it for *sentence segmentation*. In some cases user can input not a sentence, but a whole paragraph. Before using PunktWordTokenizer to tokenize sentence into tokens, we need to segment it into sentences. This can be easily used by including Slovene Punkt sentence segmenter. Following example shows, how does sentence segmenter work. If we have the following paragraph:
  Zdravljíca je pesem, ki jo je France Prešeren napisal novembra leta 1844. Po 6. členu ustave je Zdravljica besedilo himne Republike Slovenije. Melodija je iz istoimenske zborovske skladbe skladatelja Stanka Premrla.

  the sentence segmenter will output the following list of sentences:
  [*'Zdravljíca je pesem, ki jo je France Prešeren napisal novembra leta 1844.', 'Po 6. členu ustave je Zdravljica besedilo himne Republike Slovenije.', 'Melodija je iz istoimenske zborovske skladbe skladatelja Stanka Premrla.'*]

The first step is to use sentence segmenter, to form a list of sentences. This list is then being tokenized with PunktWordTokenizer. One drawback with this tokenizer is that it doesn't tokenize the last word of a sentence and the following period (as shown in the example). Before using the PunktWordTokenizer, we must divide them with space. After we tokenize the sentences, tagging can be used. The result is a list of tuples with words or characters and their appropriate morphosyntactic description.

We also implemented a function that returns two description dictionaries for MSD's with attribute=value expansions in Slovene and English. This way user can view which part-of-speech words belong to in Slovene or English. Following example demonstrates the expected output:

(Lep | PPNMEIN) - pridevnik
(je | GP-STE-N) - glagol
(dan | SOMEI) - samostalnik
(, | ,) - ni razlage
(vse | ZC-SEI) - zaimek
(diši | GGNSTE) - glagol
(že | L) - členek
(po | DM) - predlog
(pomladi | SOZEM) - samostalnik
(! | !) - ni razlage

We must exmplain, that here only the part of whole MSD is printed. For example the whole MSD for *Lep* shown in table I.

The first output was created with function *prettyPrintWithDescription*, full output can be created with *prettyPrintWithFullDescription*. Both function are defined in `example.py`

But there is an issue. Because Slovenian language contain letters with caron, the output will have unicode symbols for them. In the previous examples, the output is already corrected. Solution is described in the following section.

pridevnik	
vrsta	splošni
spol	moški
število	ednina
sklon	imenovalnik
živost	0
vid	0
oblika	0
oseba	0
nikalnost	0
stopnja	nedoločeno
določnost	ne
število_svojine	0
spol_svojine	0
naslonskost	0
zapis	0

TABLE I
WHOLE MSD FOR *Lep* IN SLOVENE

D. Encoding problems

Printing the result of our example is not optimal. Because we want users to know what these characters are, we implemented a solution. To resolve this, we must decode every word to *UTF - 8* (because words we tag are in UTF - 8). If a user wants to print the result in "pretty" form, he should call function *prettyPrint()* or *prettyPrintByLine()* (both are implemented in `example.py`).

IV. RESULTS

For all constructed taggers, we provide their accuracy. The results were obtained by repeating the process of building the tagger with various evaluation arguments (see script `evaluateTaggers.sh`). See Table II and Figure 4.

fraction	*Trigram*	*NaiveBayes*	*Brill*
0.75	0.8286	0.8451	0.8490
0.80	0.8304	0.8461	0.8514
0.82	0.8306	0.8465	0.8515
0.84	0.8299	0.8462	0.8508
0.86	0.8306	0.8472	0.8512
0.88	0.8299	0.8474	0.8517
0.90	0.8288	0.8467	0.8507
0.91	0.8278	0.8462	0.8489
0.92	0.8276	0.8464	0.8490
0.93	0.8281	0.8464	0.8491
0.94	0.8278	0.8461	0.8496
0.95	0.8289	0.8475	0.8510
0.96	0.8388	0.8538	0.8609
0.97	0.8401	0.8568	0.8612
0.98	0.8417	0.8588	0.8629
0.99	0.8379	0.8567	0.8578
average	**0.8317**	**0.8490**	**0.8529**

TABLE II
TAGGERS ACCURACY FOR VARIOUS VALUES OF FRACTION PARAMETER.

As seen, there are no major differences in tagging accuracy. The Brill and the Naive Bayes are about 2% better than Trigram.

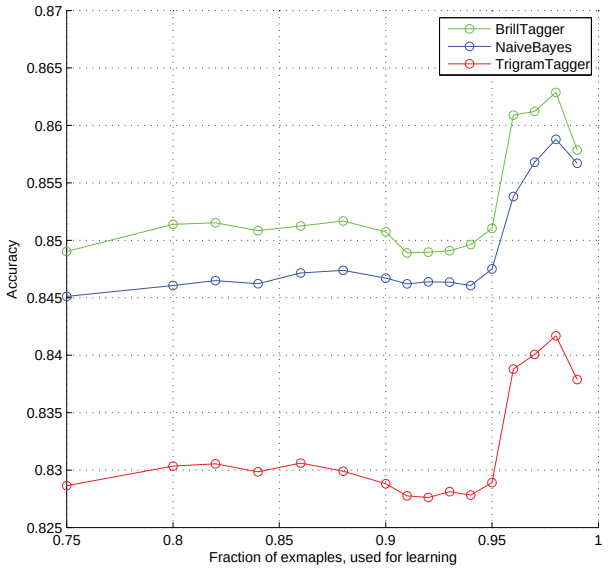

Fig. 4. Accuracy evaluation results.

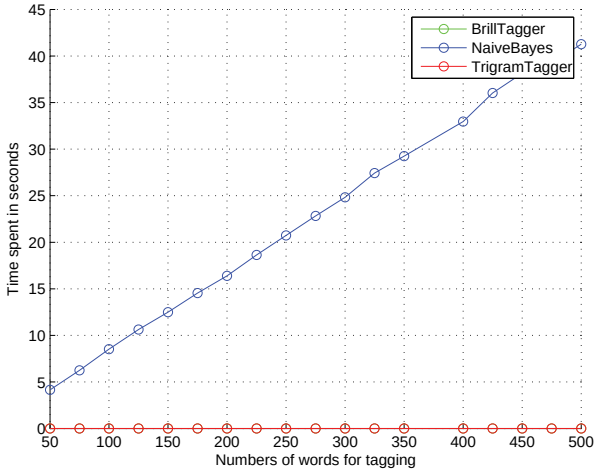

Fig. 5. Spent time measurements. NOTE: The green line is not visible because it is right under the red one.

We noticed that the time for evaluations differ drastically. The results are listed in Table III and graphically represented on Figure 5 (see script `evaluateTaggersSpeed.py`).

number of words	*Trigram*	*NaiveBayes*	*Brill*
50	0.0003	4.1543	0.0011
75	0.0004	6.2464	0.0014
100	0.0006	8.5232	0.0018
125	0.0008	10.6370	0.0020
150	0.0009	12.4937	0.0024
175	0.0011	14.5484	0.0027
200	0.0011	16.3944	0.0029
225	0.0013	18.6332	0.0032
250	0.0015	20.7357	0.0035
275	0.0017	22.8280	0.0041
300	0.0018	24.8243	0.0042
325	0.0018	27.4270	0.0043
350	0.0022	29.2458	0.0051
400	0.0023	32.9599	0.0052
425	0.0024	36.0322	0.0056
450	0.0027	38.1471	0.0062
475	0.0027	39.2278	0.0064
500	0.0029	41.2688	0.0063

TABLE III
TIME SPENT IN SECONDS FOR TAGGING DIFFERENT NUMBERS OF WORDS.

The difference is enormous. Brill and Trigram run at almost the same speed, while Naive Bayes runs much slower. This is also one of the reason why we prefer using the Brill tagger.

V. CONCLUSION

To conclude our paper, tagging in Slovene language is practical with proper use of Natural Language Toolkit and *nltk-trainer*. Time spent in seconds for tagging different numbers of words, show that Brill and Trigram ran at the same speed, while Naive Bayes ran drastically slower. There are no major differences in tagging accuracy. However Brill and the Naive Bayes are about 2% better than Trigram.

Besides these three taggers we built, others can be built without any bigger modification. All that is needed is to set some arguments when calling the script `train_tagger.py`. The main reason we built only these three is computing power. These three are the only one we could built and evaluate in some reasonable time.

Our whole project is publicly available on github [8]. We also contacted the NLTK developers group and will arrange our taggers to become the part of NLTK.

ACKNOWLEDGEMENTS

We would like to express our gratitude to all the contributors of *JOS* [2] project, *nltk-trainer* [3] project and to Prof. Marko Robnik-Šikonja, who was our supervisor at Faculty of Computer and Information Science. Also this project would not be possible without *Natural Language Toolkit* [1] and its admirable documentation.

REFERENCES

[1] (2011) NLTK web page. [Online]. Available at: http://www.nltk.org/
[2] (2011) JOS project web page. [Online]. Available at: http://nl.ijs.si/jos/
[3] (2011) NLTK-trainer project source code. [Online]. Available at: https://github.com/japerk/nltk-trainer
[4] (2011) Slovene MULTEXT-East Morphosyntactic Specifications. [Online]. Available at: http://nl.ijs.si/ME/V4/msd/html/msd-sl.html
[5] S. Bird, E. Klein, and E. Loper. (2009, June). Natural Language Processing with Python — *Analyzing Text with the Natural Language Toolkit*. (1st ed.) [Online]. Available at: http://www.nltk.org/book
[6] T. Erjavec. (2011, January 29). TEI Schema for SSJ and JOS corpora of Slovene. [Online]. Available at: http://nl.ijs.si/ssj/schema/tei_ssj_doc.pdf
[7] (2011) Tokenizers. [Online]. Available at: http://docs.huihoo.com/nltk/0.9.5/guides/tokenize.html
[8] (2011) Slovene NLTK Tagger (the source code from our project). [Online]. Available at: https://github.com/nikicc/slovene-nltk-tagger
[9] (2011) Wikipedia. [Online]. Available at: http://en.wikipedia.org/wiki/Part-of-speech_tagging