

BUILDING AN EFFICIENT, SCALABLE, AND TRAINABLE PROBABILITY-AND-RULE-BASED PART-OF-SPEECH TAGGER OF HIGH ACCURACY

by

JIAYUN HAN

(Under the Direction of Michael Covington)

ABSTRACT

This project is aimed to build an efficient, scalable, portable, and trainable part-of-speech tagger. Using 98% of Penn Treebank-3 as the training data, it builds a raw tagger, using Bayes' theorem, a hidden Markov model, and the Viterbi algorithm. After that, a reinforcement machine learning algorithm and contextual transformation rules were applied to increase the tagger's accuracy. The tagger's final accuracy on the testing data is 96.51% and its speed is about 26,000 words per second on a computer with two-gigabyte random access memory and two 3.00 GHz Pentium duo processors. The tagger's portability and trainability are proved by the tagger-maker's success in building a new tagger out of a corpus that is annotated with the tagset different from that of Penn Treebank.

INDEX WORDS: Part-of-Speech, Tagging, Markov Model, The Viterbi Algorithm, The Bayesian Theorem, Machine Learning, Contextual rules, Natural Language Processing

BUILDING AN EFFICIENT, SCALABLE, AND TRAINABLE PROBABILITY-AND-RULE-
BASED PART-OF-SPEECH TAGGER OF HIGH ACCURACY

by

JIAYUN HAN

MA, Sichuan International Studies University, China, 1999

PHD, The University of Georgia, The United States, 2007

A Thesis Submitted to the Graduate Faculty of The University of Georgia in Partial Fulfillment
of the Requirements for the Degree

MASTER OF SCIENCE

ATHENS, GEORGIA

2009

© 2009

Jiayun Han

All Rights Reserved

BUILDING AN EFFICIENT, SCALABLE, AND TRAINABLE PROBABILITY-AND-RULE-
BASED PART-OF-SPEECH TAGGER OF HIGH ACCURACY

by

JIAYUN HAN

Major Professor: Michael Covington

Committee: Paula Schwanenflugel
Alexander Williams

Electronic Version Approved:

Maureen Grasso
Dean of the Graduate School
The University of Georgia
May 2009

DEDICATION

To my lovely wife, Xianchun Huang, and my dear son, Jing Han, for their endless
support

ACKNOWLEDGEMENTS

I sincerely thank Dr. Covington for his longitudinal guidance, great patience and kind encouragement, without which this thesis could not have been done. My sincere gratitude also goes to Dr. Schwanenflugel and Dr. Williams for their insightful revision suggestions. I also deeply thank my wife and my son, who pardoned me for not being able to spare more time to enjoy life with them.

TABLE OF CONTENTS

	Page
ACKNOWLEDGEMENTS	v
LIST OF TABLES	viii
LIST OF FIGURES	ix
CHAPTER	
1 INTRODUCTION.....	1
2 SURVEY OF MAJOR EXISTING TAGGING PROGRAMS.....	6
2.1 RULE-BASED TAGGERS	6
2.2 PROBABILITY-BASED TAGGERS	7
2.3 HYBRID TAGGERS	11
3 MAJOR ALGORITHMS.....	14
3.1 BAYES' THEOREM	14
3.2 A HIDDEN MARKOV MODEL.....	17
3.3 THE VITERBI ALGORITHM	19
3.4 CONTEXTUAL ADJUSTMENT ALGORITHM.....	20
3.5 REINFORCEMENT MACHINE LEARNING ALGORITHM	21
3.6 ALGORITHM DEALING WITH UNKNOWN WORDS	24
4 DEVELOPING THE TAGGER.....	26
4.1 DATA.....	26
4.2 DEVELOPING THE RAW TAGGER	26
4.3 TRAINING THE RAW TAGGER	31
4.4 CONTEXTUAL FINAL ADJUSTMENT.....	38

4.5 BUILDING THE TAGGER-MAKER	39
5 RESULTS	42
5.1 TAGGING ACCURACY	42
5.2 TAGGING SPEED	46
5.3 PORTABILITY AND TRAINABILITY	47
6 CONCLUSIONS	49
BIBLIOGRAPHY	52
APPENDICES	55
A SOFTWARE DOCUMENTATION	55
B PENN TREEBANK TAGSET	64
C THE C5 TAGSET OF CLAWS4	65

LIST OF TABLES

	Page
<i>Table 2.1 Major existing taggers and their features</i>	<i>13</i>
<i>Table 4.1 The word-tag-frequency statistical figures</i>	<i>27</i>
<i>Table 4.2 The tag-count lookup table</i>	<i>29</i>
<i>Table 4.3 The tag-word emission lookup table</i>	<i>30</i>
<i>Table 4.4 Unique bigrams and their frequencies</i>	<i>30</i>
<i>Table 4.5 Unique trigrams and their frequencies</i>	<i>30</i>
<i>Table 4.6 The tags-tag transition probability lookup table</i>	<i>31</i>
<i>Table 4.7 The pseudo code for tagging a sequence of words</i>	<i>32</i>
<i>Table 4.8 The pseudo code for adjusting emission probabilities</i>	<i>35</i>
<i>Table 4.9 The pseudo code for adjusting tags-tag transition probabilities</i>	<i>36</i>
<i>Table 4.10 The training parameters and their values</i>	<i>37</i>
<i>Table 4.11 The pseudo code for training the tagger as a whole</i>	<i>37</i>
<i>Table 4.12 The pseudo code for building the context dictionary</i>	<i>39</i>
<i>Table 4.13 The pseudo code for contextual constraint adjustments</i>	<i>40</i>
<i>Table 4.14 The pseudo code for building a tagger out of training data</i>	<i>41</i>
<i>Table 5.1 The tagging results</i>	<i>43</i>
<i>Table 5.2 The time needed for building a new tagger</i>	<i>48</i>
<i>Table 5.3 The new tagger's accuracy results</i>	<i>48</i>

LIST OF FIGURES

	Page
<i>Figure 5.1 The tagging accuracies on the training data and the testing data.....</i>	<i>43</i>
<i>Figure 5.2 The tagging speed on the training data and the testing data.....</i>	<i>44</i>
<i>Figure 5.3 The 12 hours' learning progress.....</i>	<i>45</i>
<i>Figure A.1 The screenshot of tagging the entered text</i>	<i>56</i>
<i>Figure A.2 The screenshot of checking tagging accuracy.....</i>	<i>57</i>
<i>Figure A.3 The screenshot of tagging and Training Options panel</i>	<i>59</i>
<i>Figure A.4 The screenshot of setting the training parameters</i>	<i>60</i>

CHAPTER 1

INTRODUCTION

In its broad sense, *tagging* in natural language processing (NLP) refers to any process that assigns certain labels to certain linguistic units. In its narrow sense, as it is used in this thesis, it denotes the assignment of part-of-speech tags to texts. A computer program for this purpose is called a *tagger*. Both *word* and *part-of-speech* are used in their broad senses. The former in effect includes everything but white space and the latter is not limited to the categories listed in traditional grammar books. For example, Treebank-3 (Marcus, Santorini, Marcinkiewicz, & Taylor, 1999) uses 36 tags and the C6 tagset of the CLAWS4 tagger has 138 tags (Leech, Garside, & Bryant, 1994). As an example, (1.1) is a tiny excerpt of the tagged Switchboard, which is one of the four components of Treebank-3 (See Appendix B for the tag descriptions).

(1.1) SpeakerB3/SYM ./.

Well/UH what/WP do/VBP you/PRP think/VB about/IN the/DT idea/NN
of/IN ./, uh/UH ./, kids/NNS having/VBG to/TO do/VB public/JJ
service/NN work/NN for/IN a/DT year/NN ?/.

Almost all advanced NLP projects, such as syntactic parsing, information retrieval, text mining, speech-to-text conversion, building annotated corpora, etc., rely on tagging as the first operation. Their performances can be significantly improved by an excellent tagger. For example, one of the syntactic parsers used by the company which I am working for incorrectly parsed

sentence (1.2) as (1.4). After manually assigned the correct tags to *are*, *to*, and *use*, the sentence was parsed correctly and the parsing time was reduced by about three times.

(1.2) *Bots in Bot Colony are programmed to use spoken language when a human is present.*

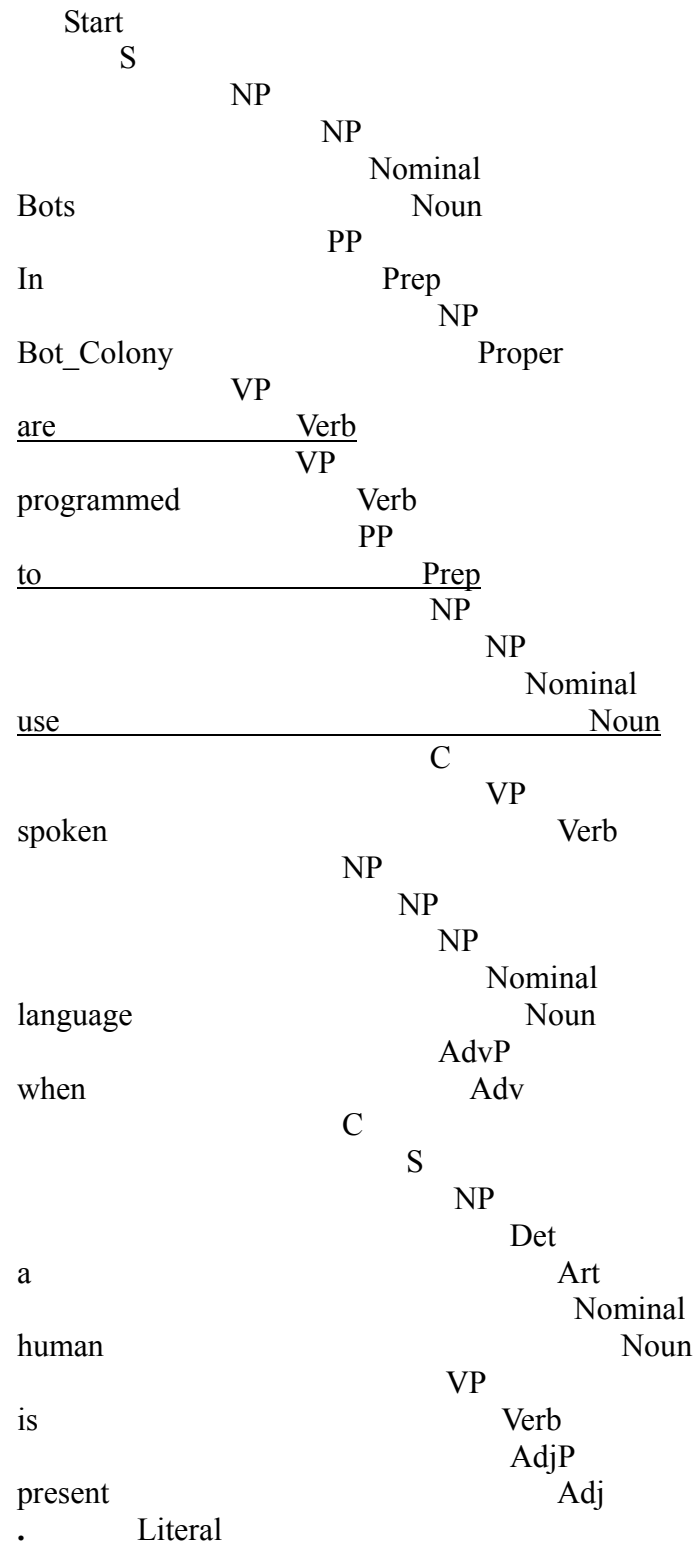
(1.3) *Bots in Bot Colony (Aux = are) programmed (Aux= to) (Verb = use) spoken language when humans are present.*

Given the important roles played by tagging in NLP, an efficient, scalable, portable, and trainable tagger with high accuracy is needed. However, almost no existing tagger possesses all of these features, as shown in Chapter 2. This motivates me to choose as my thesis topic developing a first-class tagger and an automatic tagger-maker. The resulted tagger will have the following features: a) its accuracy will be above 96% on the testing data, b) its performance will not be affected by the size of the text to be tagged, c) it can tag at least 200,000 words per second on the testing computer, and d) it can improve itself using the annotated corpus supplied by users. The resulted tagger-maker will have the following features: a) given a pre-tagged corpus of considerably big size, it will build a fast, completely scalable, and considerably accurate tagger, b) there will be no requirements for the corpus' annotation scheme, its text type, or its language, and c) the tagger-making process will be completely automatic.

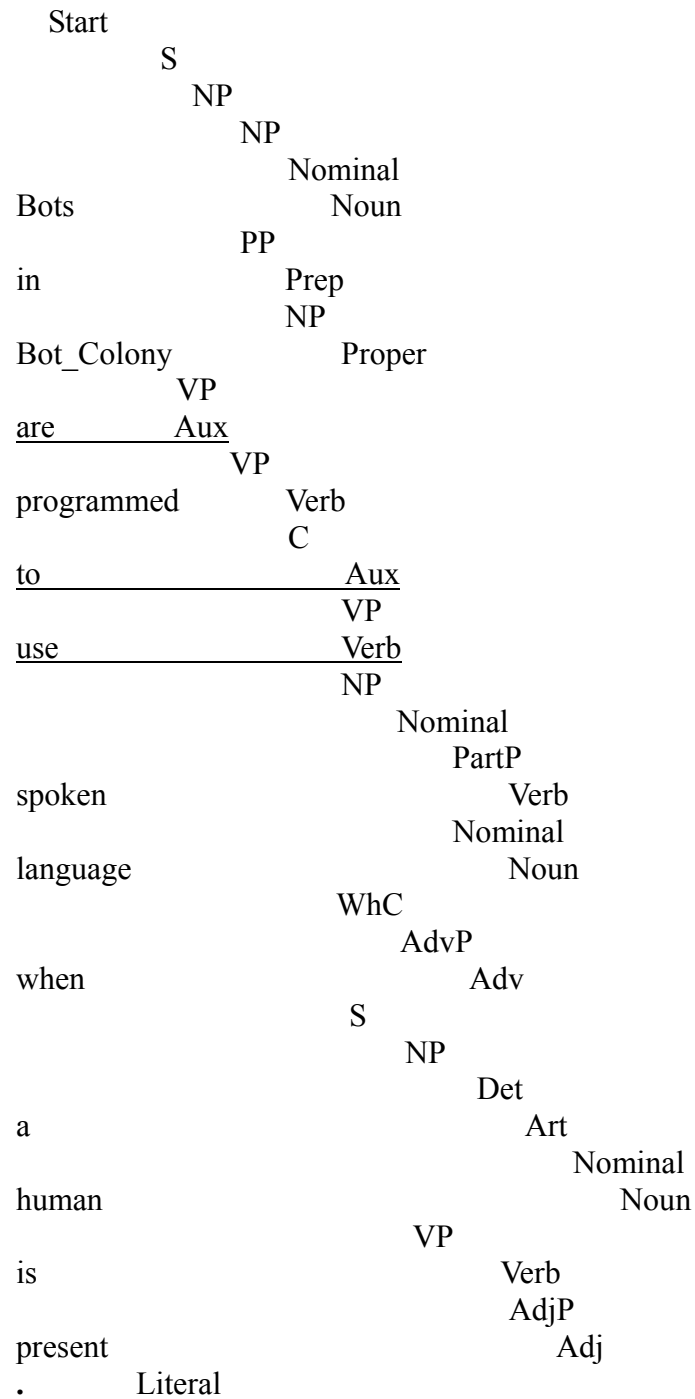
The major algorithms to be used include Bayes' theorem, a hidden Markov model, the Viterbi algorithm, a reinforcement machine learning algorithm, the contextual constraint algorithm, and the algorithm to deal with unknown words.

Written in C#, the program will provide end-users with a Windows interface and programmers with the well-documented APIs.

(1.4) The incorrect syntactic tree of the sentence *Bots in Bot Colony are programmed to use language when a human is present*.



(1.5) The correct syntactic tree of the sentence *Bots in Bot Colony are programmed to use language when a human is present.* (After manually assigning the correct parts-of-speech to the underlined words)



To avoid redundant description, throughout the thesis, the *testing computer* refers to the desktop computer that is used to test the programs. It has two-gigabyte random access memory and two 3.00 GHz Pentium duo processors.

The rest of the thesis is arranged as this: chapter 2 gives a brief survey of the major existing taggers, chapter 3 describes the major algorithms to be used, chapter 4 details the implementation of the algorithms, chapter 5 presents the tagging and tagger-making results, and chapter 6 concludes the thesis, summarizing its achievements.

CHAPTER 2

SURVEY OF MAJOR EXISTING TAGGING PROGRAMS

Tagging projects started from the 1950s. According to what they are based on, the existing taggers can be classified into three major families: those based on linguistic rules, those based on probabilities, and those based on both.

2.1 RULE-BASED TAGGERS

This type of taggers rely on contextual rules such as *if a word is an article, it cannot be followed by a base verb in simple sentences*. The rules are either written by linguists using their linguistic knowledge of a particular language or automatically generated from pre-tagged corpora. An example of the former is Klein and Simmons (1963) and an example of the latter is Brill (1995).

Klein and Simmons (1963) called the linguistic rules that they used the contextual grammar coder (CGC). The CGC system was developed empirically through manual analysis of the simple text found in a children's encyclopedia. They used CGC to tag a couple of pages of that same encyclopedia and reported that the accuracy was slightly over 90% and the speed was about 21 words per second. Of course, neither the accuracy nor the speed can be compared with today's taggers, due to the poor computers and lack of manually annotated corpora. However, their project proved that it is completely possible to use a computer to automatically assign parts-of-speech to texts.

Instead of writing the contextual rules manually, Brill (1995) generated 280 contextual rules automatically from the tagged Brown corpus, using a machine learning approach. All of the rules are of the form *A B contextual condition* where *A* is the tag to be transformed from and *B* is the tag to be transformed to. Only when the contextual condition is satisfied, can the transformation take place. For example, a rule like *vbn vbd PREV TAG np* means changing a past participle (*vbn*) into a past form (*vbd*) if it is preceded by a proper noun (*np*).

The Brill tagger is composed of three parts: 1) the lexical tagger, which, for each word, chooses among its possible tags the one with the highest frequency, 2) the unknown word tagger, which assigns the guessed tags to unknown words, and 3) the contextual tagger, which applies the transformation rules.

The Brill tagger is significant in that it proved that computer can learn to generate rules from data. His tagger's accuracy is from 95% to 97%. However, as pointed out by Roche & Schabes (1995), it is inherently slow for two reasons. First, the tagger applies all rules to each of the sentences to be tagged, matching the tags one by one. Since it does not remember which tags have already been compared, it performs a tremendous amount of unnecessary string comparison. Second, some of the rules may cancel each other, resulting in unnecessary computation.

In general, writing contextual rules by hand is a tedious process and matching them one by one is computationally expensive. This plus the availability of large pre-tagged corpora and the affordability of super computers encourages researchers to develop probability-based taggers, which, in my view, accounts for the rarity of rule-based taggers.

2.2 PROBABILITY-BASED TAGGERS

These taggers can be classified into five subtypes according to their major algorithms: 1) those based on various Markov models (Church, 1988; Kim, Rim, & Tsujii, 2003; Lee, Tsujii, & Rim, 2000a, 2000b, 2000c; Thede & Harper, 1999), 2) those using genetic algorithms (Araujo,

2002; Enrique, Luque, & Araujo, 2006), 3) those based on artificial neural networks (Roth & Zelenko, 1998; Schmid, 1994), 4) those employing statistical decision trees (Kim et al., 2003; Marquez, Padro, & Rodriguez, 2000), and 5) those relying on the maximum entropy theory (Ratnaparkhi, 1996; Toutanova & Christopher, 2000).

The common feature of the first subtype is that they are based on two types of probabilities: the probability of a word being a particular tag and the probability of a tag following or preceding a fixed number of other tags. These probabilities are calculated by counting the frequencies of the relevant items of the pre-tagged corpora. A typical example is Church (1988), which finds the best tag sequence in the following steps.

1) Calculating the probabilities of all unique words' being all possible tags and the probabilities of all unique tags' being preceded by unique bigrams. He called the former the *lexical probability*, and the latter the *contextual probability*.

2) Treating the text as though it were composed of the last word only and calculating the probabilities of all parts-of-speech of that word, each of which is the product of that word's lexical probability and its contextual probability.

3) Adding one word backward and calculating the probabilities of all possible tag combinations of those words.

4) Repeating step three until all words are processed.

5) Choosing the tag sequence that has the highest probability.

Church (1988) is one of the earliest projects that employ dynamic programming to solve the tagging problem. He reported that his tagger's accuracy ranged from 95% to 99%. However, he didn't mention how it dealt with unknown words. Though the author claimed that the computation time was linear to the number of the words to be tagged, it may be much longer

than that, since adding all tags of each preceding word increases the possible partial tag sequences exponentially.

Araujo (2002) and Enrique et al. (2006) are the first projects to use genetic algorithms to solve the tagging problem. These two papers are almost the same so I will review the more recent one only. Their basic algorithm is summarized as follows.

First, it randomly initiates a list of possible tag sequences, called *population*. Each member of the population is a *genotype* made up of *genes*. A genotype is actually a sequence of part-of-speech tags. Second, it randomly forms a number of genotype couples and exchanges the genes (part-of-speech tags) within each couple to produce new genotypes and this process is called *crossover*. Third, it randomly selects one gene of each new genotype and randomly changes that gene into another allowable tag and this process is called *mutation*. Finally, it used the fitness function $f_{genotype} = \sum_i f_{gene_i}$ to score each genotype to find n best genotypes to replace them for n randomly selected genotypes to update the population and this process is called *survival selection*. f_{gene} is the fitness score of each gene, which is calculated by [2.1], where *occ* stands for *occurrences*, *LC* for left context tags, *T* for the current tag in question, *RC* for the right context tags, and *l* is the number of the possible tags of a word.

$$[2.1] \quad f_{gene} = \log(occ(LC, T, RC) / \sum_{j=0}^{j=l-1} occ(LC, T^j, RC))$$

The entire cycle of crossover, mutation, and survival selection is called an evolutionary generation. After the preset number of generations is reached, evolution stops and the genotype with the highest fitness is chosen, which is the optimal tag sequence.

From the algorithms summarized above, we know that there are three problems in using evolutionary algorithms to tag texts. First, it is bound to be slow due to the computationally

expensive processes of crossover, mutation, and survival selection. Second, the tagging result varies from one run to another. Third, the tagger may not be scalable, i.e., the size of the text may give the tagger big trouble. The reported best accuracy was 96.41% after 5,656 generations of evolution and the reported speed was about 10,000 words per second.

An example of using statistical decision trees to solve the tagging problem is Marquez, Padro, & Rodriguez (2000). Using 96% of WSJ of Penn Treebank as the training data, the authors group all multi-tag words into 253 ambiguity classes, such as class *JJ-VBG* (e.g. *amusing*, *exciting*, etc.) and class *JJ-VBD-VBN* (e.g. *amused*, *excited*, *surprised*, etc.). The tagging problem thus becomes a classification problem. They selected the following attributes: the third tag to the left of the word in question (*tag-3*), the second tag to the left (*tag-2*), the first tag to the left (*tag-1*), the first tag to the right (*tag+1*), and the second tag to the right (*tag+2*), and the word's spelling features, such as capitalization, containing digits, etc. In other words, the best tag of a word is decided by its contextual and spelling characteristics.

They tested their tagger on the remaining 4% of WSJ and the reported accuracy was 97%. However, it was able to tag only about 300 words per second on a computer with an UltraSparc2 processor. Although, I have no idea of the performance difference between UltraSparc2 processors and Pentium processors, I believe that even if they had used today's powerful processor, the tagger would still have been very slow.

A widely used tagger is the Stanford tagger (Toutanova & Christopher, 2000) that uses the maximum entropy theory to search for the optimal sequence of tags. It is trained on sections 0-20 of WSJ of Treebank-3 and tested on sections 23-24. The reported accuracy is 96.86%. No speed is reported. The Stanford tagger is trainable in that it can produce a tagger by using a tagged corpus provided by users. Using the testing computer and the same testing data, I replicated the

tagger's performance. The accuracy is about the same as that reported but it took more than 42 minutes to tag the 111,117 tokens at the speed of 44 words per second. I then tested it on the 1-million-word Brown corpus with the result that the testing computer crashed due to an out-of-memory exception after more than 10 hours' computing. In sum, the Stanford tagger is trainable and has reached the state-of-the-art accuracy but it is unbearably slow and is not scalable.

2.3 HYBRID TAGGERS

Taggers of this family use both linguistic rules and probability-based methods. One of the most successful examples is the CLAWS project by Lancaster University (Garside, Leech, & Sampson, 1987; Leech et al., 1994; Marshall, 1983). The latest version is the CLAWS4 tagger, which is also called the BNC tagger due to the fact that it was used to tag the 100-million-word British National Corpus (BNC). Leech et al. (1994) summarized the core algorithms of the CLAWS4 tagger as follows:

- (a) segmentation of text into word and sentence units*
- (b) initial (non-contextual) part-of-speech assignment [using a lexicon, word-ending list, and various sets of rules for tagging unknown items]*
- (c) rule-driven contextual part-of-speech assignment*
- (d) probabilistic tag disambiguation [Markov process]*
- (e) output in intermediate form*

(pp. 622-623)

The probability-based algorithm used by the CLAWS4 tagger is a hidden Markov model. The linguistic rules were actually built up over 14 years and they are responsible for tagging multi-word expressions, such as *according to*, *as well as*, *kind of*, *sort of*, etc. Each word of such

an expression should be assigned the same tag as that of the expression taken as a whole. For example, in (2.1), *according* and *to* are both given the tag *PRP2*, and *sort* and *of* are both tagged as *AV02*. An ending number is used to indicate the position of the word in the multi-word expression, hence *According_PRP21* and *to_PRP22*. The definitions for the C5 tagset are given in Appendix C.

(2.1) *According_PRP21 to_PRP22 that_DT0 news_NN1 ,_, the_AT0 enemy_NN1
was_VBD sort_AV021 of_AV022 defeated_VVN ._.*

Lancaster university provides a trial tagging service at <http://ucrel.lancs.ac.uk/claws/trial.html>. It accepts up to 10,000 words for tagging every time. (2.1) is the online tagging result. To test the tagger's accuracy and its speed, I tried some files from the Brown corpus and manually checked the result. The accuracy is about 96% and the speed is about 500 words per second including the time for data transmission through the network.

That the CLAWS4 tagger successfully tagged the 100-million-word BNC indicates that it is completely scalable. The linguistic rules make the CLAWS4 tagger outstanding in handling multi-word expressions. On the other hand, we should keep in mind that as those rules were written just for the English language, they are not portable to other languages.

Table 2.1 summarizes most of the taggers mentioned in this chapter. The *speed* is measured by the number of words processed per second; *portable* indicates whether the methods can be applied to other text types or other languages, and *trainable* refers to whether the tagger can improve itself or produce a new tagger, given a user-supplied pre-tagged corpus. As shown there, almost all of the taggers have achieved very high accuracy. However, it seems that the authors had little interest in the tagger's efficiency, as most of them even didn't mention it at all. An

inefficient tagger has little hope to be scalable. Finally, no tagger except the Stanford tagger is trainable.

Table 2.1 *Major existing taggers and their features*

“Portable” indicates whether the methods can be applied to other text types or other languages. “Trainable” refers to whether the tagger can improve itself or produce a new tagger if given a pre-tagged corpus by users.

Tagger	Major algorithms	Accuracy	Speed (wps)	Portable?	Trainable?
Klein and Simmons (1963)	using manually written contextual rules	90%	21	No	No
Brill (1995)	using generated transformation rules	95%-97%	unknown	Yes	No
Church (1988)	Markov process	95%-99%	unknown	Yes	No
Enrique et al. (2006)	evolutionary algorithm	94.61%	10,000	Yes	No
Marquez et al. (2000)	decision tree	97%	300 on Ultraspac2	Yes	No
The Stanford tagger	maximum entropy	96.68%	44	Yes	Yes
CLAWS	Markov process & Linguistic rules	$\approx 96\%$	500 via internet	No	No
Banko & Robert (2004)	contextualized hidden Markov model	97.24	unknown	Yes	No
Kim et al. (2003)	variable memory Markov models	96.9%	unknown	Yes	No
Thede & Harper (1999)	2nd-order of hidden Markov model	98.04%	unknown	Yes	No
Roth & Zelenko (1998)	winnow-based network	98%	unknown	Yes	No
Schmid (1994)	neuron network	97.79%	unknown	Yes	No

CHAPTER 3

MAJOR ALGORITHMS

The major algorithms used for this project are 1) Bayes' theorem (Mitchel, 1997), 2) a Markov assumption (Brill, 2000), 3) the Viterbi algorithm (Forney, 1973), 4) a reinforcement machine learning algorithm, 5) the algorithm to deal with unknown words, and 6) the contextual transformation algorithm.

3.1 BAYES' THEOREM

In machine learning we are interested in finding from all possible hypotheses the most probable one, given the training data D . This can be done by calculating the probability of each hypothesis and choosing the one with the highest probability. The most probable hypothesis is called the *maximum a posteriori* hypothesis, notated as h_{MAP} , which is found by [3.1], where h is just one hypothesis and H is the set of all possible hypotheses and $P(h | D)$ is the probability of h given the data D .

$$[3.1] \quad h_{MAP} \equiv \arg \max_{h \in H} P(h | D)$$

Then how do we find $P(h | D)$? The answer lies in Bayes' theorem, which provides a way to calculate the probability of a hypothesis based on its prior probability, the probability of observing the data given the hypothesis and the probability of observing the data itself. Bayes' theorem is expressed as [3.2], where $P(D)$ is the probability of observing data independent of the hypothesis h , $P(h)$ is the probability that a hypothesis holds independent of the data D ,

and $P(D | h)$ is the probability of observing data D given some world in which the hypothesis h holds.

$$[3.2] \quad P(h | D) = \frac{P(D | h) \times P(h)}{P(D)}$$

Applied to tagging, a hypothesis h is simply one possible tag sequence t , and D is the word sequence W . Therefore, $P(h)$ is $P(t)$, the probability that t is the correct sequence of tags independent of W ; $P(D)$ is $P(W)$, the probability of observing W independent of t ; $P(D | h)$ is $P(W | t)$, the probability of observing W given t , and finally $P(h | D)$ is $P(t | W)$, the probability that t is the correct tag sequence given W . Bayes' theorem is thus expressed as [3.3] when applied to tagging.

$$[3.3] \quad P(t | W) = \frac{P(W | t) \times P(t)}{P(W)}$$

Since $P(W)$ is the same for all possible tag sequences, it can be dropped from the expression which simplifies [3.3] into [3.4] and [3.1] into [3.5].

$$[3.4] \quad P(t | W) = P(W | t) \times P(t)$$

$$\begin{aligned} [3.5] \quad h_{MAP} &\equiv \arg \max_{t \in T} P(t | W) \\ &= \arg \max_{t \in T} P(W | t) \times P(t) \end{aligned}$$

To get the result of (3.5), we need to calculate $P(t | W)$ for each of the possible tag sequences and find the one with the highest probability. That is, we need to know the value of $P(W | t)$ and that of $P(t)$ for each possible tag sequence. A short sentence *Bob can go* is used to illustrate how to calculate those two values, where *Bob* has two possible tags: noun and verb, *can* has three: modal, verb, and noun, and *go* has two: noun and verb. Therefore this sentence can

have $2 \times 3 \times 2 = 12$ possible tag sequences, as listed in [3.6] where *md* stands for modal verb.

Take the tag sequence t_{12} for example. $P(W | t_{12})$ is calculated by [3.7] and $P(t_{12})$ by [3.8].

$$[3.6] \quad \left. \begin{array}{l} t_1 = \textit{noun}, \textit{noun}, \textit{noun} \\ \dots \\ t_{12} = \textit{noun}, \textit{md}, \textit{verb} \end{array} \right\} \in T$$

$$[3.7] \quad P(\textit{Bob}, \textit{can}, \textit{go} | \textit{noun}, \textit{md}, \textit{verb}) = P(\textit{Bob} | \textit{noun}) \times P(\textit{can} | \textit{md}) P(\textit{go} | \textit{verb})$$

$$[3.8] \quad P(\textit{noun}, \textit{md}, \textit{verb}) = P(\textit{noun}) \times P(\textit{noun_md}) \times P(\textit{noun_md_verb})$$

Each term of the right side of [3.7] and [3.8] can be obtained by counting the frequencies of certain items of the pre-tagged training data. For example, $P(\textit{can} | \textit{md})$ is the frequency of the word *can* tagged as a modal verb divided by the total occurrences of the modal verb tag in the training data while $P(\textit{noun_md})$ is the frequency of a noun immediately followed by a modal verb in the training data divided by the total frequency of the given tag, i.e. the tag *noun* in the training data. Similarly, $P(\textit{noun_md_verb})$ is calculated by dividing the frequency of a noun immediately followed by a modal verb immediately followed by a verb in the training data by the total frequency of the leading tags, i.e. the frequency of a noun immediately followed by a modal verb in the training data. Applied to tagging texts of any number of words, [3.7] and [3.8] are generalized into [3.9] and [3.10], respectively.

$$[3.9] \quad P(W | t) = P(w_1 | t_1) \times P(w_2 | t_2) \times \dots \times P(w_n | t_n)$$

$$[3.10] \quad P(t) = P(t_1) \times P(t_1, t_2) \times P(t_1, t_2, t_3) \times \dots \times P(t_1, t_2, \dots, t_n)$$

Obviously, it is difficult or even impossible to get the reliable values of each of the terms of the right side of [3.10] when too many words need to be tagged. To solve this problem, a Markov assumption is used, which is explained in the next section.

3.2 A HIDDEN MARKOV MODEL

A hidden Markov model is a finite state machine, in which each state emits a symbol and each state also transitions to a new state. Thus for each state, there are two associated probabilities: the probability that it emits a particular symbol and the probability that it transitions to a particular state. In other words, there is a sequence of visible symbols and a sequence of hidden states. Applied to tagging, the sequence of visible symbols is the sequence of words to be tagged while the sequence of hidden states is the sequence of tags to be obtained. The tagging goal is thus to search for the sequence of tags that has the highest probability, as indicated by [3.11], which is the same as [3.5], the simplified Bayes' theorem.

$$[3.11] \quad h_{MAP} \equiv \arg \max_{t \in T} P(W | t) \times P(t)$$

As stated in section 3.1, getting $P(t)$ can be computationally intractable when the text contains too many words. To make it feasible to calculate $P(t)$ for texts of any size, a Markov assumption is made, which assumes that a state is dependent only on a small and fixed number of previous states. For example, a bigram model assumes that a state is subject to its immediate preceding state only, i.e. a tag is dependent only on the one that immediately precedes it. This assumption changes [3.8] into [3.12] for our simple sentence when $t = \textit{noun_md_verb}$ and the generalized formula [3.10] into [3.13]. The value of [3.13] can be easily calculated by counting the frequencies of certain items in the training data.

$$[3.12] \quad P(\textit{noun}, \textit{md}, \textit{verb}) = P(\textit{noun} | \textit{Start}) \times P(\textit{md} | \textit{noun}) \times P(\textit{verb} | \textit{md})$$

$$[3.13] \quad P(t) = P(t_1 | \textit{Start}) \times P(t_2 | t_1) \times P(t_3 | t_2) \times \dots \times P(t_n | t_{n-1})$$

Obviously, English does not completely follow the Markov assumption in terms of part-of-speech distributions. For example, the part-of-speech of *go* in *The **person** near the window of the library of the University of **Georgia** goes to church every Sunday* is not dependent

on the tag of its immediate predecessor, i.e., *Georgia*, but on that of *person*, the word quite far away from *go*. Despite the falsity of the Markov assumption for English, in most cases, it still produces good result.

Applied to tagging, for a bigram Markov model in which a tag is influenced by its immediate preceding tag, the prior probability of a tag sequence is the product of the probabilities of each tag transitioned from its preceding tag, as expressed in [3.14]. The probability of observing a sequence of words given a sequence of tags is the product of the probability of each word emitted by its corresponding tag in that tag sequence, as expressed in [3.15]. The posteriori probability of a particular tag sequence t given a sequence of words W is the product of [3.14] and [3.15], as expressed by [3.16].

$$[3.14] \quad P(t) = \prod_{i=0}^w P(tag_{i+1} | tag_i)$$

$$[3.15] \quad P(W | t) = \prod_{i=1}^w P(word_i | tag_i)$$

$$[3.16] \quad P(t | W) = P(t) \times P(W | t)$$

The simple Markov assumption has solved the problem of calculating $P(t)$. To get h_{MAP} of [3.11], a simple solution is to calculate all $P(t | W)$ where $t \in T$ and choose the one with the highest probability. The machine learning approach using Bayes' theorem in this fashion is called brute-force Bayes' concept learning (Mitchel, 1997). This learning method is computationally intractable when used to tag considerably long text since the run-time of getting the value of [3.11] is $O(n^{|W|})$ where n is the average number of tags of each of the words and w is the number of the words. To get this around, the Viterbi algorithm (Forney, 1973) is used, which is explained in the next section.

3.3 THE VITERBI ALGORITHM

The main idea of the Viterbi algorithm is that instead of iterating over all possible state sequences to choose the best state sequence, we iterate over all possible candidates of each state to get the best one for that individual state. The concatenation of the best individual states produces the best state sequence. Applied to tagging, this algorithm searches for the best tag for each word in order to find the best tag sequence. The best tag of a word is calculated by [3.17] where tag_w is one possible tag of the word, the plural form, $tags_w$, is all of the possible tags of that word, Tag_w is the best tag of that word to be obtained, and $tags_p$ are the small fixed number of tags preceding that word.

$$[3.17] \quad Tag_w = \arg \max_{tag_w \in tags_w} P(w | tag_w) \times P(tag_w | tags_p)$$

Using a bigram Markov model, Bayes' theorem, and the Viterbi algorithm, the procedures to find the best tag sequence of the sentence *Bob can go* are listed below.

1. Find the best tag of *Bob* which is

$$Tag_{Bob} = \arg \max_{tag_{Bob} \in tags_{Bob}} P(Bob | tag_{Bob}) \times P(tag_{Bob} | Start)$$

2. Find the best tag of *can* which is

$$Tag_{can} = \arg \max_{tag_{can} \in tags_{can}} P(can | tag_{can}) \times P(tag_{can} | tag_{Bob})$$

3. Find the best tag of *go* which is

$$Tag_{go} = \arg \max_{tag_{go} \in tags_{go}} P(go | tag_{go}) \times P(tag_{go} | tag_{can})$$

4. The best tag sequence of the sentence *Bob can go* is

$$[3.18] \quad h_{MAP} = Tag_{Bob} + Tag_{can} + Tag_{go}$$

If the average number of tags of a word is n and there are w words to be tagged, the run-time to calculate the best tag sequence is $O(n^2 \times |W|)$, which is exponentially shorter than the time needed by brute-force Bayes' concept learning.

3.4 CONTEXTUAL ADJUSTMENT ALGORITHM

Due to the fact that it is very difficult to determine the correct tag of some ambiguous words, such as *can*, *may*, *must*, *will*, *might*, *saw*, etc, the above algorithms frequently make mistakes. This project uses the contextual information to adjust the tags of these tough words as a remedial measure, based on the assumption that a word's tag is constrained by the tags of its surrounding words. Using the phrase *a full can of beans* as an example, the algorithm is illustrated as this: if we know the tags of *a*, *full*, *of*, and *beans* are determiner, adjective, preposition, and noun, respectively, which form a tag sequence of *determiner, adjective, ?, preposition, noun*, we can then search the training data for the word *can* that is preceded by an *adjective* which in turn is preceded by an *determiner* and that is followed by a *preposition* which in turn is followed by a *noun*. If this search succeeds, we can change the tag of *can* in *a full can of beans* obtained previously into the tag of *can* given by the training data in this context.

For this algorithm, there are two points to be considered. First, we know that the bigger the context size is, the more accurate the obtained tag will be. However, if the context size is too big, we may not find that tag sequence with the word in question in that particular slot in the training data. I choose the two tags preceding the word and the two tags following the word as the contextual tags. Second, whether this algorithm works depends on the accuracy of the contextual tags. For example, only after we get the correct tags of *a*, *full*, *of*, and *beans*, can it be possible to adjust the tag of *can*. This calls for a training algorithm to get the most accurate probability of a tag's transitioning to its following tag and the most accurate probability of that tag's emitting a

particular word. This is the goal of the reinforcement machine learning algorithm of this project, which is detailed in the following section.

3.5 REINFORCEMENT LEARNING ALGORITHM

As shown from [3.17], the choice of a word's tag depends on two probabilities: the emission probability, $P(\text{word} | \text{tag})$, and the transition probability, $P(\text{tag} | \text{tags})$. Therefore, we can design a machine learning algorithm, which gradually adjusts these two values to minimize the tagging difference between the tags assigned by a tagger and those by the training data. The learning algorithm is stated as follows.

For a multi-tag word, if it is assigned an incorrect tag in a particular context, then this word's probability of being the incorrect tag needs to be reduced and its probability of being the correct one needs to be increased. By the same token, this error can also be corrected by reducing the probability of the incorrect tag transitioned from its preceding tag(s) and by increasing the probability of the correct tag transitioned from its preceding tag(s). After the adjustments of the probabilities, tag the entire sequence of words again with the new probabilities. If the accuracy drops, cancel the adjustments; if the accuracy increases, save the probability changes permanently by writing them to files; if the accuracy has no change, keep the probability changes in memory without updating the files. This is because in most cases, there will be no accuracy change and updating the files frequently would greatly slow down the training process.

Let us again use the example *Bob can go* to illustrate how this learning algorithm works. Let us make the following assumptions (where *NNP* stands for *proper noun*, *NN* stands for *common noun*, *MD* stands for *modal verb*, and *VB* for *base verb*):

- 1) This sentence appears in the training data and is pre-tagged as

(3.1) *Bob/NNP can/MD go/VB ./.*

2) This sentence is tagged by our tagger as

(3.2) *Bob/NNP can/NN go/VB ./.*

3) The probabilities of *can* emitted by its various tags and the probabilities of those tags transitioned from its preceding tag, *NNP*, are obtained from the pre-tagged data as:

$$P(\text{can} | MD) = 0.02 \quad P(MD | NNP) = 0.001$$

$$P(\text{can} | VB) = 0.006 \quad P(VB | NNP) = 0.002$$

$$P(\text{can} | NN) = 0.01 \quad P(NN | NNP) = 0.003$$

The above assumptions show that 1) the word *can* that is a modal verb in this context is incorrectly tagged as a noun by the tagger, 2) given a proper noun, a common noun that follows it has the highest probability (0.003) while a modal verb that follows it has the lowest probability (0.001), and 3) the word *can* has the highest probability of being emitted by a modal verb (0.02) and the lowest probability of being emitted by verb (0.006). The tag *MD* is called the weak tag because its probability should be increased in order to be assigned to *can* while the tag *NN* is called the strong tag because its probability should be reduced in order to be removed from this word in this context.

The probabilities of *can* being emitted by *MD*, *NN*, and *VB* are calculated as follows:

$$P(MD | \text{can}) = P(\text{can} | MD) \times P(MD | NNP) = 0.02 \times 0.001 = 0.00002$$

$$P(VB | \text{can}) = P(\text{can} | VB) \times P(VB | NNP) = 0.006 \times 0.002 = 0.000012$$

$$P(NN | \text{can}) = P(\text{can} | NN) \times P(NN | NNP) = 0.01 \times 0.003 = 0.00003$$

Thus the highest probability of the tag, given the word *can* is *NN*, which is not correct, however, when compared with the training data. We can correct this problem by reducing the probability of *can* emitted by the strong tag *NN*, and that by the non-strong tag *VB* by deducting different values from their probabilities. These deductions are called taxes in this paper and are

given to the emission probability of the weak tag *MD*. We can do the same thing to adjust the transition probabilities. In this example, I will only illustrate how to adjust emission probabilities as adjusting the transition probabilities can be done in a similar manner.

Let $tax_{strong} = 0.002$ be the tax for the strong tag *NN* and $tax_{common} = 0.001$ for the tag *VB* that is neither strong nor weak. The processes of tax-collection and subsidiary-granting produce the new probabilities of *can* emitted by *MD*, *VB*, and *NN* as:

$$P(can | MD) = 0.02 + 0.002 + 0.01 = 0.023$$

$$P(can | VB) = 0.006 - 0.001 = 0.005$$

$$P(can | NN) = 0.01 - 0.002 = 0.008$$

Now we retag the sentence and get the new probabilities of *can* emitted by *MD*, *VB*, and *NN* as:

$$P(MD | can) = P(can | MD) \times P(MD | NNP) = 0.023 \times 0.001 = 0.000023$$

$$P(VB | can) = P(can | VB) \times P(VB | NNP) = 0.005 \times 0.002 = 0.00001$$

$$P(NN | can) = P(can | NN) \times P(NN | NNP) = 0.008 \times 0.003 = 0.000024$$

The new probability of *can* being emitted by *NN* is still higher than that by *MD* but the difference is considerably reduced. We continue adjusting the word-emission probabilities until $P(MD | can) > P(NN | can)$. After that, we save the new probabilities to file.

In this example, I only illustrated how to change the word-emission probabilities of one word *can*. However, in the real learning situation we must keep in mind that after *can* in this particular context is tagged correctly due to the probability adjustments, *can* in other places of the training data may be tagged incorrectly and that the word (for the bigram mode) or the two words (for the trigram mode) following *can* may be tagged incorrectly as well, since their tags depend on the new tag of *can*. Therefore, after every change of probabilities, we need to retag the

whole text and let the tagging result decide whether to cancel, to keep in memory only, or to permanently save the probability changes.

3.6 ALGORITHM DEALING WITH UNKNOWN WORDS

The above algorithms assume that every word of the text to be tagged has appeared in the training data. In reality, this is not the case. No matter how big the training corpora are, they cannot include the entire vocabulary of a language. Therefore, the tagger should be equipped with the intelligence to gauge the possible tag(s) of unknown words. The tagger to be developed achieves this by using the words available in the training corpora and the word's morphological compositions. The following steps are used to deal with unknown words:

First, restore an inflected word to its base form by removing the inflections. For example, the word *marketability* has no plural form in normal situation. However, if for some reason, it does appear as *marketabilities*, the tagger should be able to know that it is the plural form of *marketability* and tag it accordingly.

Second, if inflection-dropping fails, try removing the prefixes. For instance, most old training corpora like Penn Treebank do not have words like *epassport* or *e-saver*. But we can treat *e* as a prefix and get the base form *passport* and *saver*. If we also know that *e* as a prefix does not change the part-of-speech of the word to which it is attached, we can tag these two words as NN.

Third, if prefix removal fails, try removing the suffixes. For instance, training corpora most probably do not have the word *girlless* (as in *A girlless party is boring*). Through morphological analysis, we know that this word is composed of *girl* and the adjectival suffix *-less*, and thus we can tag *girlless* as adjective with confidence.

In many cases, all of the above three procedures will all be used, even recursively for that matter, to obtain the parts-of-speech of words like *demodernizations*, *anti-internationalization*, *sonlessness*, etc. correctly.

Fourth, if the previous means fail, we resort to the endings of the words to gauge their tags if they end with typical suffixes, such as *-ia*, *-hood*, *-ity*, *-dom*, *-age*, *-some*, etc. We need to keep in mind the ambiguity of some suffixes. For example, *-en* can be an adjective (e.g. *wooden*), a verb (e.g. *shorten*), or a noun (e.g. *garden*). If this is the case, we have to list all possible tags of a word ending with that suffix and let the other algorithms decide its final tag.

Finally, if all of the above means fail, just tag an unknown word as a proper noun and a common noun if it starts with a capital letter and tag it as a common noun only, otherwise.

But in any case, an unknown word should not be tagged as a closed-class word, such as pronoun, conjunction, preposition, etc.

CHAPTER 4

DEVELOPING THE TAGGER

Chapter 3 explained the algorithms. This chapter describes how I will implement them.

4.1 DATA

The data used for the project are the first three components of the Treebank-3, namely, the Brown corpus, Switchboard and Wall Street Journal (WSJ), of which Switchboard is made up of telephone interviews. Every sentence of Treebank is given a syntactic tree and every token is given a tag or tags. Switchboard tags *is* as BES and *has* as HVS while the Brown corpus and WSJ tag both as VBZ. For consistency, Michael Covington converted the BES and HVS of Switchboard to VBZ. This project uses the converted version of Switchboard. Covington also randomly divided the Brown corpus, Switchboard, and WSJ into the training data and the testing data at the ratio of about 98% to 2%, which are used in this project as the training data and the testing data, respectively.

4.2 DEVELOPING THE RAW TAGGER

There are two steps in developing the tagger: developing a raw tagger and training the raw tagger into a more accurate one using a reinforcement machine learning algorithm. This section describes the implementation of the raw tagger and the next section details the implementation of the learning algorithm.

The tagger to be developed uses the trigram hidden Markov model. According to equation [3.17], which is restated as [4.1] below for easy reference, the tag of a word is determined by the

probabilities of that word being emitted by each of its possible tags, $P(\text{word} \mid \text{tag} \in \text{tags})$, and the probabilities of each of those tags being transitioned from its two preceding tags, $P(\text{tag}_k \in \text{tags}_k \mid \text{tag}_{k-2} \text{tag}_{k-1})$, where tags are all possible tags of a word and tag is only one of them. Therefore, the core values we need are the emission probabilities and the transition probabilities.

$$[4.1] \quad \text{Tag}_w = \arg \max_{\text{tag}_w \in \text{tags}_w} P(w \mid \text{tag}_w) \times P(\text{tag}_w \mid \text{tags}_p)$$

4.2.1 Constructing the Tag-word Emission Probability Lookup Table

To find the *tag-word* emission probabilities, two lookup tables are constructed out of the training data. One lists all unique words, their possible tags, and the frequencies of those words being particular tags, as obtained from the training data. Table 4.1 is part of that table.

Table 4.1 *The word-tag-frequency statistical figures*

Word	All tag-frequency pairs
,	, 358007 UH 5 VBP 4 DT 3 RB 2 JJ 1 VBN 1 PRP 1 FW 1 IN 1 NN 1
...	...
a	DT 79087 SYM 15 FW 9 VBP 7 NN 6 VB 5 PDT 4 JJ 4 RB 4 NNP 3 LS 2 VBN 2 IN 1 VBD 1 VBG 1 PRP 1 , 1
...	
Abandoned	VBN 39 VBD 20 JJ 4
...	...
will	MD 7356 NN 137 VBP 1 VB 1
...	...
Zygmunt	NNP 1

As shown from the table, comma is used as a comma for 358007 times, as an UH (exclamation) for five times, as a present verb in plural form for four times, etc. The entries revealed the tagging noise of Treebank. There is little reason why the comma in (4.1) was tagged as VBP. I regard this kind of noise as tagging errors and removed them from Table 4.1. That is, punctuation marks will be tagged as punctuation marks only.

(4.1) ...if/IN you/PRP solve/VBP **/VBP** help/VB them/PRP to/TO work/VB

through/IN their/PRP\$ problems/NNS... (*Switchboard 3134*)

However, (4.2), (4.3), (4.4), (4.5) indicate that the annotators of Treebank were trying to guess the actual words in the speakers' mind when tagging their utterances. The *a* in (4.2) was tagged as VBP probably because it was believed to be the broken form of *are*. The *a* in (4.3) was tagged as VB because that particular context requires *have* or the *a* is the weakened spoken form of *have* in that context. The *a* in (4.4) was tagged as VBG because it was believed to be the stammering form of *appealing*. Finally, the *a* in (4.5) was tagged as RB (adverb) because it is believed to be the weakened pronunciation of the word *of*, which together with *sort* forms the multi-word hedging adverb *sort of*. This kind of noise is not tagging errors, though it is very hard for a probability-based tagger, such as this one, to tag words in those particular contexts in these peculiar ways. They will be removed from Table (4.1), since keeping them will not increase the tagging accuracy but considerably increase the computation time, if we recall that the tagger needs to iterate every possible tag of a word to calculate the highest probability.

(4.2) now/RB ./, watches/VBZ seven/CD point/NN two/CD hours/NNS of/IN

television/NN a/DT day/NN ./, and/CC that/IN school/NN children/NNS ./,
a/VBP ./, **a/VBP** ./, are/VBP not/RB far/RB off/IN that/DT mark/NN with/IN
six/CD point/NN eight/CD ./ (*Switchboard 2926*)

(4.3) So/UH ./, we/PRP could/MD n't/RB **a/VB** done/VBN much/RB better/RBR

than/IN **that/DT** in/IN Buffalo/NNP ./ (*Switchboard 2521*)

(4.4) And/CC how/WRB long/JJ had/VBD he/PRP been/VBN **a/VBG** ./,

appealing/VBG ?/. How/WRB long/JJ was/VBD that/DT ?/.

(*Switchboard 4856*)

(4.5) I/PRP work/VBP and/CC I/PRP live/VBP in/IN the/DT city/NN so/RB ./,
that/DT sort/RB **a/RB** kind/RB of/RB hung/VBD it/PRP up/RP ./.

(Switchboard 2562)

The other lookup table contains the unique tags and their frequencies. Table 4.2 is part of that table. Using Table 4.1 and Table 4.2, the actual tag-word emission probability lookup table, Table 4.3, is built, as illustrated below, using the word *will* acting as a noun for example.

From Table 4.1, we know that *will* appeared as a noun 137 times in the training data. From Table 4.2 we know that totally noun occurred 468466 times. Therefore, given a noun, the probability that it is *will* is $137 / 468466 = 0.000292443848646433$, which is exactly the corresponding figure listed in Table 4.3. The other values of the emission probability table are calculated in the same manner.

Table 4.2 *The tag-count lookup table*

Tag	Freq	Tag	Freq	Tag	Freq
NN	468466	POS	18190	RP	11117
MD	46400	PRP	270609	SYM	1438
NNP	208019	PRP\$	43916	TO	89991
NNPS	6441	RB	221072	VB	134624
NNS	180389	RBR	6099	VBD	128419
PDT	3503	RBS	1847	VBG	58290

4.2.2 Constructing the Tags-tag Transition Probability Lookup Table

To find the *tags-tag* transition probabilities, two auxiliary lookup tables are needed as well. One collects the unique bigrams that appeared in the training data and their frequencies; the other lists the unique trigrams and their frequencies. Table 4.4 is part of the first table while Table 4.5 is part of the second one.

Table 4.3 *The tag-word emission lookup table*

Tag	Word	Emission probability
NN (singular common noun)	will	0.000292443848646433
	William	2.13462663245572E-06
	willingess	2.13462663245572E-06
	Willingness	2.13462663245572E-06
	willingness	6.40387989736715E-05
	willow	1.92116396921015E-05
	will-to-power	2.13462663245572E-06
	Wilmington	2.13462663245572E-06
	...	..
MD	will	0.158196950472053
	willya	2.15058388352438E-05
	...	...

Table 4.4 *Unique bigrams and their frequencies*

Bigram	Freq	Bigram	Freq
NNP NNP	71739	DT NN	156733
...	...	...	...

Table 4.5 *Unique trigrams and their frequencies*

Trigram	Freq	Trigram	Freq
NNP NNP VBD	6669	NNP NNP NNP	18171
DT NN IN	46662	DT NN RB	4582
...	...	...	...

Recall that for the trigram model, the $tag_i tag_j - tag_k$ probability is the probability that $tag_i tag_j$ is followed by tag_k . This can be calculated by dividing the frequency of $tag_i tag_j tag_k$, which is listed in Table 4.5, by the frequency of $tag_i tag_j$, which is stored in Table 4.4. For example, given two consecutive NNPs, the probability that the third tag is still an NNP is 0.253293187805796 , which is the total frequency of NNP|NNP|NNP (18171, according to Table 4.5) divided by that of NNP|NNP (71739, as shown from Table 4.4). Other transition

probabilities are calculated in the same way. Table 4.6 is part of the transition probability lookup table built out of Table 4.4 and Table 4.5, using the method just described.

Table 4.6 *The tags-tag transition probability lookup table*

From-tag	To-tag	Tag-tag transition probability
NNP NNP	NNP	0.253293187805796
	VBD	0.0929619872036131
	...	
DT NN	IN	0.297716498759036
	RB	0.0292344305283508
	...	
...	...	...
	...	...

Using equation [4.3], Table 4.3, and Table 4.6, the raw tagger is built. The pseudo code for tagging a sequence of words is listed in Table 4.7.

4.3 TRAINING THE RAW TAGGER

As said earlier, the determinant values are the emission probabilities and the transition probabilities, which we have calculated and stored them in Table 4.3 and Table 4.6. Recall that they are obtained by counting the frequencies of each word's all tags, the frequency of the unigrams, bigrams, and trigrams. Most taggers based on Markov models stop at this step. However, we can try and test whether the two kinds of probability values are actually the ones needed by the program and whether there is any room for modification so that the tagger can achieve higher accuracy. This motivates me to try and adjust these probability values.

4.3.1 *Preparing the training data*

To speed up computation, the project first processes all of the training files into two big lists, one holding all the words and the other containing their corresponding tags. After that, the raw tagger tags the list of words, which gives the tagging accuracy and produces the following two sublists.

Table 4.7 *The pseudo code for tagging a sequence of words*

```
words ← a list of words to be tagged
best_tag ← the best tag of a word to be obtained
resulted_tags ← the sequence of the tags to be obtained, initially empty
emission_prob ← a word's probability of being emitted by a tag, which is listed in Table 4.3
transition_prob ← a tag's probability of being transitioned from two consecutive preceding tags,
which is listed in Table 4.6

Build a run-time dictionary
  key = a word of words
  value = the possible tags of that word. If a word exists in Table 4.1, it is retrieved from that
  table; otherwise, it is figured out by morphological analysis

for each word of words
{
  if word has only one tag
  {
    best_tag = that only tag
  }
  else
  {
    highest_prob = 0

    for each tag of word's tags
    {
      this_prob = emission_prob × transition_prob

      if (this_prob > highest_prob)
      {
        highest_prob = this_prob
        best_tag = tag
      }
    }
    add best_tag to resulted_tags
  }
}
return resulted_tags
```

The first sublist is called *unmatched_emission_items*, each member in the form of

$Word | Tag_c | Tag_w | Freq$ where Tag_c is the tag given by the training data which is treated as the correct tag, Tag_w is the tag incorrectly assigned by the raw tagger and is different from

Tag_c , and $Freq$ is the frequency of such wrong tagging. For example, the item $saw | NN | VBD | 23$ means that, for 23 times, *saw* that is supposed to be tagged as a noun is tagged as a past tense verb. The other sublist is called *unmatched_transition_items*, each member in the form of $Tag_1 | Tag_2 | Tag_c | Tag_w | Freq$ where $Tag_1 | Tag_2$ are the two consecutive leading tags, Tag_c is the tag following the leading tags given by the training data, Tag_w is the actual tag assigned by the tagger and is different from Tag_c , and $Freq$ is the frequency of such wrong tagging. For example, $DT | JJ | NN | MD | 23$ means that, for 23 times, a tag sequence of a determiner followed by an adjective that is supposed to be followed by a noun is actually given a modal verb by the raw tagger.

The frequency value, $Freq$, is called the tolerance level, which determines how strict the training is going to be. For instance, if we set $Freq$ as 30, then the above two errors will not be collected in the two lists, and they will not be treated as tagging errors.

4.3.2 Adjusting the tag-word emission probabilities

This section illustrates how the tagger learns to improve itself by adjusting the tag-word emission probabilities, using $saw | NN | VBD | 23$ as an example. The pseudo code is listed in Table 4.8.

We know that *saw* has four possible tags: NN (noun, e.g., *a plastic saw*), VB (base verb, e.g., *to saw it open*), VBP (the present verb with a plural subject, e.g. *They saw wood in the afternoon.*), and VBD (the past form of *see*, e.g., *I saw her a moment ago.*). The item $saw | NN | VBD | 23$ indicates that $P(saw | NN)$, which is the probability that that noun is the word *saw*, should be increased. Therefore, the tag *NN* of this string is called the weak tag. It also tells us that $P(saw | VBD)$ should be decreased. Therefore VBD is called the strong tag. The

other tags of this word, i.e. VB and VBP are called the common tags, which are neither strong nor weak.

There are two ways to adjust the probabilities. One is expressed in [4.2], where tax is the value to be deducted from the probability of the strong tag and added to that of the weak tag $\bar{P}$ is the post-adjustment probability and ε is the learning rate that ranges from 0 to 1 exclusive. Figuratively, it is the tax rate that the strong tag uses to calculate the tax it should pay.

$$[4.2] \quad tax = P(saw | VBD) \times \varepsilon$$

$$\bar{P}(saw | VBD) = P(saw | VBD) - tax$$

$$\bar{P}(saw | NN) = P(saw | NN) + tax$$

The other way involves changing the probabilities of the common tags as well, as expressed in [4.3], where κ is the tax rate of the common tags relative to that of the strong tag. κ ranges from 0 to 1, inclusive. If $\kappa = 0$, it means no tax whereas if $\kappa = 1$, it means it uses the same tax rate as that of the strong tag. Through experiments, I found that [4.2] is worse than [4.3] as it terminates learning too soon.

$$[4.3] \quad tax' = P(saw | VBD) \times \varepsilon$$

$$\bar{P}(saw | VBD) = P(saw | VBD) - tax'$$

$$tax'' = P(saw | VB) \times \varepsilon \times \kappa$$

$$\bar{P}(saw | VBD) = P(saw | VBD) - tax''$$

$$tax''' = P(saw | VBP) \times \varepsilon \times \kappa$$

$$\bar{P}(saw | VBP) = P(saw | VBP) - tax'''$$

$$\bar{P}(saw | NN) = P(saw | NN) + tax' + tax'' + tax'''$$

Table 4.8 *The pseudo code for adjusting emission probabilities*

```
items ← list of strings in the form of “word|weak_tag|strong_tag”, where “weak_tag” is the
        correct tag and “strong_tag” is the wrong tag
bestAccuracySoFar ← the highest accuracy so far, initially set as 0

for each item of items
{
    split item into word, weak-tag and strong-tag
    find all the tags of word
    adjust emission probabilities as described in [4.3]
    thisAccuracy ← retag the text with new probability values

    if (thisAccuracy < bestAccuracySoFar)
    {
        cancel probability adjustments
    }
    else
    {
        if (thisAccuracy > bestAccuracySoFar)
        {
            write new probabilities to file
        }
        bestAccuracySoFar = thisAccuracy
    }
}
return bestAccuracySoFar
```

4.3.3 Adjusting the Tags-tag transition probabilities

Adjusting the tags-tag transition probabilities is similar to adjusting the tag-word emission probabilities. The only difference is that the probabilities of the common tags stay intact. This choice is made to reduce the computation complexity since there are too many tags that can follow two consecutive tags. The transition adjustment is expressed in [4.4], using the transition item $DT | JJ | NN | MD | 23$ as an example. The pseudo code is listed in Table 4.9.

$$[4.4] \quad tax = P(MD | DT _ JJ) \times (1 - \varepsilon)$$

$$\bar{P}(MD | DT _ JJ) = P(MD | DT _ JJ) - tax$$

$$\bar{P}(NN | DT _ JJ) = P(NN | DT _ JJ) + tax$$

Table 4.9 *The pseudo code for adjusting tags-tag transition probabilities*

```
items ← list of strings in the form of “tag_tag|weak_tag|strong_tag”, where “tag_tag” is the two
        leading tags, “weak_tag” is the correct tag following tag_tag and “strong_tag” is the wrong
        tag following tag_tag
bestAccuracySoFar ← the highest accuracy so far

for each item of items
{
    split item into tag_tag, weak-tag, strong-tag
    adjust transition probabilities as described in [4.4]
    thisAccuracy ← retag the text with new probability values
    if (thisAccuracy < bestAccuracySoFar)
    {
        cancel probability adjustments
    }
    else
    {
        if (thisAccuracy > bestAccuracySoFar)
        {
            write new probabilities to file
        }
        bestAccuracySoFar = thisAccuracy;
    }
}
return bestAccuracySoFar
```

4.3.4 Training the tagger as a whole process

Using the training parameter values listed in Table 10, Table 4.11 lists the pseudo code for the training process as a whole by adjusting the emission probabilities and the transition probabilities alternatively. As stated in Table 4.8 and Table 4.9, after each adjustment, the tagger retags the whole text, which has two effects. One is to update *unmatched_emission_items* and *unmatched_transition_items*. The other is to report the new accuracies which may be higher than, lower than, or the same as the previous accuracy. For each training generation, the final tagging accuracy is the highest among the accuracy of the last generation, the new accuracy obtained from adjusting the emission probabilities, and the new accuracy obtained from adjusting the transition probabilities. For each generation, if the new accuracy reaches the accuracy goal,

Table 4.10 *The training parameters and their values*

Parameter	Value
accuracy goal: learning stops if this goal is reached	0.965
error tolerance: the frequency by which an erroneous pattern is regarded as an error	350
tolerance reducer: the value by which error tolerance is reduced for each training generation	20
generations: number of training cycles; learning stops if this number is reached	15
ε : tax of strong tag = its existing probability $\times \varepsilon$	0.1
κ : tax of non-strong tag = its existing probability $\times \varepsilon \times \kappa$	0.2

Table 4.11 *The pseudo code for training the tagger as a whole*

```

errorTolerance = 350;
bestAccuracy = 0.0;
newAccuracy = 0.0;
generations = 0;
limit: the number of cycles beyond which training stops
goal: the accuracy goal
emnlItems: the list of “word|correct_tag|wrong_tag” items
tranItems: the list of “tags|correct_tag|wrong_tag” items

while (generations < stopLimit AND bestAccuracy < goal)
{
    increment generations by 1
    decrease errorTolerance by 20

    newAccuracy
    emnlItems    } ← Do Tagging
    tranItems

    bestAccuracy = max(bestAccuracy, newAccuracy)
    if (bestAccuracy >= accuracyGoal OR emnlItems and tranItems are both empty)
    {
        stop
    }
    update emission probs
    accuracy1    } ← Adjust emissions probs
    accuracy2

    update transition probs
    accuracy2    } ← Adjust emissions probs
    accuracy2

    bestAccuracy = Max(bestOfAll, accuracy1, accuracy2)
}

```

training stops. Training will always stop if the number of training generations has reached the number of the preset training cycles, regardless of the training effect to avoid endless loop. The initial tolerance value is 350, high enough to ensure to focus on the most serious errors. For each training generation, the tolerance value decreases by 20 so that more errors can be corrected.

4.4 CONTEXTUAL FINAL ADJUSTMENT

In the last section, I described the implementation of the machine learning algorithm to fine-tune the tag-word emission probabilities and the tags-tag transition probabilities into the best values by using them to tag the training corpora and comparing the result with the original training data. This section illustrates the implementation of the contextual adjustment algorithm.

This algorithm is actually divided into two procedures. One is to build a context dictionary during the development of the tagger and the other is to use this dictionary during the actual tagging process.

The pseudo code for building the context dictionary is listed in Table 4.12. The dictionary thus built is a complex dictionary: the mother dictionary key is a multi-tag word and the mother dictionary value is a child dictionary. The child dictionary key is a unique context made up of the two tags preceding that word and the two tags following that word while the child dictionary value is the tag given by the training data for that word in that particular context.

The pseudo code for performing the actual contextual adjustments is listed in Table 4.13. After the tagger produced a tag for each of the words, a word-tag pair list is produced. The adjustment process goes by looking up each word in the context dictionary. If a word is in the mother dictionary, it proceeds to build the context and looks it up in the child dictionary. If it finds the context, the tag of that word will be changed to the new tag. Otherwise, no adjustment

is performed. Though it seems complicated, with the preparation of the context dictionary, the adjustment process is actually very fast.

4.5 BUILDING THE TAGGER-MAKER

This section illustrates the implementation of the tagger-maker, which is a tool that automatically builds a tagger out of a pre-tagged corpus. The most important step is building the list of token-tag pairs from the corpus. Special care should be taken to ignore the tokens that are not tagged. The pseudo code is listed in Table 4.14.

This completed the implementation of the algorithms. The next chapter will present the tagging and the tagger-making results.

Table 4.12 *The pseudo code for building the context dictionary*

words = a list of words whose tags need to be reassigned, according to their contexts
tts = a list of token-tag pairs extracted from the training data

```
Build_mother_dictionary(words, tts)
{
    index = 0;
    for each word of words
    {
        increment index
        key = word
        value ← Build_child_dictionary(word, index, tts)
    }
}

Build_child_dictionary(word, index, tts)
{
    left2right2_s ← list of contexts built out of index and tts
    for each left2right2 of left2right2_s
    {
        key = left2right2
        value = the tag of word retrived from tts
    }
}
```

Table 4.13 *The pseudo code for performing contextual constraint adjustments*

tts = the token-tag pairs given by the tagger before contextual adjustments

tt = a token-tag pair

mDict[word, cDict[context, tag]] = a contextual dictionary built as described in Table 4.12

Context_adjustment(tts, mDict[word, cDict[context, tag]])

```
{
  for each tt of tts
  {
    index = the index of tt in tts
    word ← Get_word(tt)

    if (word is in dict[word, [context, tag]])
    {
      left2right2 ← Build_contexts(index, tts)
      if left2right2 is in cDict[context, tag]
      {
        newTag = tag
        replace newTag for the tag of tt
      }
    }
  }
}
```

Get_word(tt)

```
{
  retrieves and returns the word of tt
}
```

Build_contexts(index, tts)

```
{
  constructs and returns the left2right2 contexts, using index as the anchoring point.
}
```

Table 4.14 *The pseudo code for building a tagger out of training data*

training_files: the pre-tagged training corpus

- Build a list of token-tag pairs (tts) out of training_files
- Intelligently detect the token-tag delimiter
- Find the list of words that have more than two tags, called tough_words
- Create the token_tags.txt file out of tts.
 - This file contains the allowable tags of each unique word.
 - Each line of the resulted file is in the form of “token tag1 tag2 ... tagn”
- Create emissions.txt file out of tts
 - This file contains tag-word emission probabilities.
 - Each line of the resulted file is either a tag or in the form of “word probability”.
- Create transitions.txt file out of tts
 - This file contains tags-tag transition probabilities.
 - Each line of the resulted file is either a 2-tag string or in the form of “tag probability”.
- Create the contexts.txt file out of tough_words and tts
 - This file contains the tough words and their tags given by the training files in particular contexts.
 - Each line of the resulted file is either a word or in the form of “context tag”.
 - Modify emissions.txt and transitions.txt by using algorithms stated in Table 4.11.

CHAPTER 5

RESULTS

This chapter reports the results of the project. It presents the tagging results of the built tagger at each of the following three stages: 1) before fine-tuning the tag-word emission probabilities and the tags-tag transition probabilities through machine learning (referred to as *learning* hereafter) and before applying the contextual adjustments, 2) after learning but before applying the contextual adjustments, and 3) after learning and after applying the contextual adjustments. For easier reference, the tagger at stage one is called the raw tagger, the tagger at stage two is called the trained tagger and that at stage three is referred to as the final tagger. The accuracy and the speed of each of the three taggers on the training data and on the testing data are listed in Table 5.1, where *wps* stands for the number of words that were processed by the tagger per second. For easier visual comparisons, two figures were built out of this table. Figure 5.1 compares the tagging accuracy while Figure 5.2 compares the speed.

5.1 TAGGING ACCURACY

Reading the table horizontally, we can see that each tagger produced higher accuracy when tagging the training data than when tagging the testing data. The differences between tagging these two types of data are 1.78%, 1.63%, and 1.56% for the raw tagger, the trained tagger, and the final tagger, respectively. This result is not surprising, since, when tagging the testing data, the taggers most probably had encountered the words and the tag sequences that they had not seen before. The tags of the unknown words that were obtained through morphological analyses

may be inaccurate and the new tag sequences may have prevented the application of the contextual adjustments.

Table 5.1 *The tagging results*

Stage		On training data		On testing data	
		accuracy	wps	accuracy	wps
before learning	before contextual adjustments (raw tagger)	94.08%	544,000	92.30%	424,000
	after contextual adjustments	96.89%	400,000	95.26%	276,000
after learning	before contextual adjustments (trained tagger)	96.30%	550,000	94.55%	424,000
	after contextual adjustments (final tagger)	98.07%	403,000	96.51%	251,000

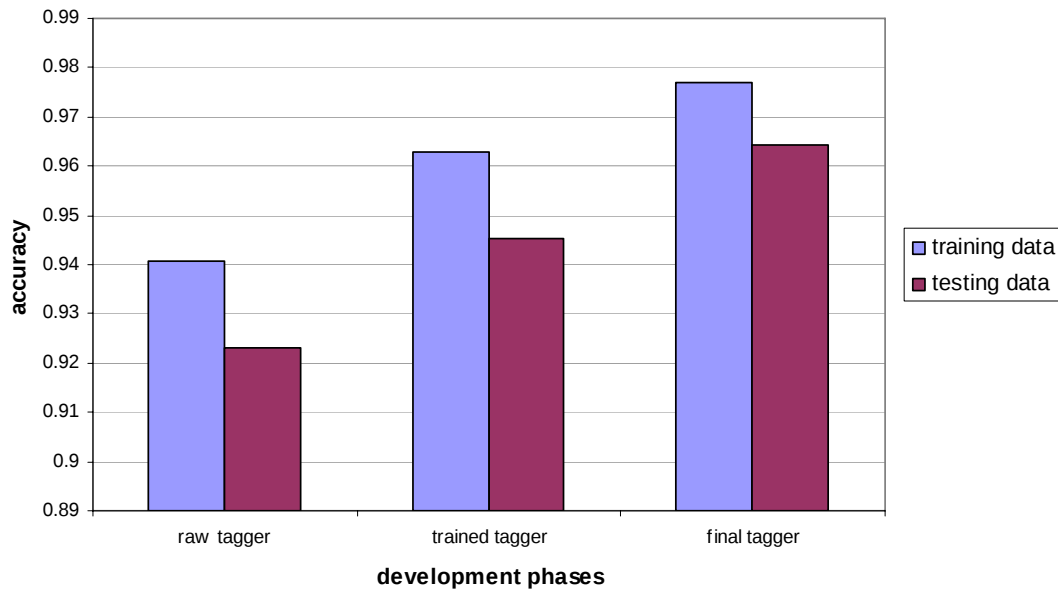


Figure 5.1 *The tagging accuracies on the training data and the testing data*

Reading the table vertically, we can have the following results. When tagging the training data, the learning process made the trained tagger 2.22% more accurate than the raw tagger and applying the contextual adjustments increased the trained tagger's accuracy by 1.77%. When tagging the testing data, the respective increases are 2.25% and 1.96%. Learning plus the contextual adjustments boosted the tagger's accuracy from 94.08% to 98.07% on the training

data, producing a total increase of 3.99%, and from 92.30% to 96.51% on the testing data, raising the accuracy by 4.21%.

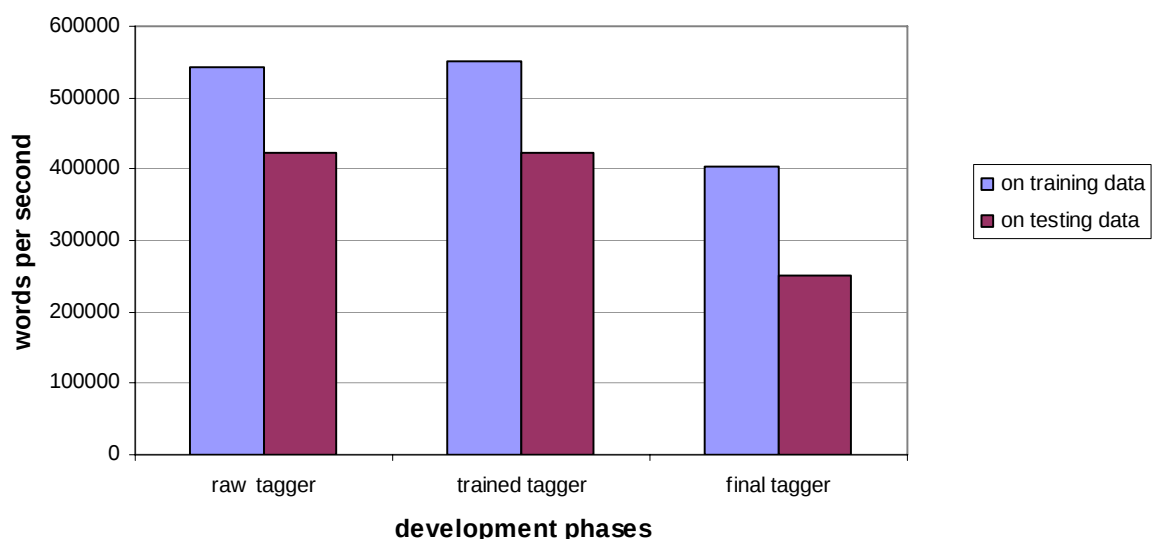


Figure 5.2 *The tagging speed on the training data and the testing data*

Table 5.1 also lists the accuracy figures of the tagger on the training data and on the testing data after applying the contextual constraints but before learning. These figures indicate that learning is necessary: the increase obtained by contextual adjustments alone is 2.81% on the training data and 2.96% on the testing data, much lower than the corresponding increases produced by contextual adjustments plus learning, namely, 3.99% and 4.21%, respectively.

The 12 hours' learning progress is presented in Figure 5.3. As shown there, most of the improvements were made during the first four hours. After that, the learning progress dropped radically. From the eighth hour on, very marginal progress was made. Recall that we excluded two types of tags from the words' part-of-speech lists: all non-punctuation tags were removed from the tag lists of punctuation marks and all tags that were given by the Treebank annotators in

an attempt to guess the semantic roles of certain words were removed from the tag lists of those words (such as the comma was annotated as the past tense of a verb and *a* was annotated as the present participle of a verb, etc.). But the tagging noise was not removed from the training data. These discrepancies decreased the tagging accuracy on the training data. For example, suppose a comma was annotated as a past tense verb and the word *a* in certain context was annotated as a present participle. Because those tags were removed from the tag list of comma and from the tag list of the word *a*, the tagger will never tag a punctuation mark as a non-punctuation mark or tag the word *a* as a present participle. The learning algorithm is designed to disallow the tagger to add new tags to a word's tag list if the training material is the Treebank corpora to prevent the tagger from learning the "bad" tagging habits. Furthermore, for the reasons I don't know, the Treebank annotators sometimes simply made mistakes such as tagging *never* as the comparative form of an adjective (presumably due to the *-er* ending of this word) but the new tagger refuses to tag those words in those incorrect ways. Taking these points into consideration, it is hard for the learner to progress further once its accuracy reached about 96.30%.

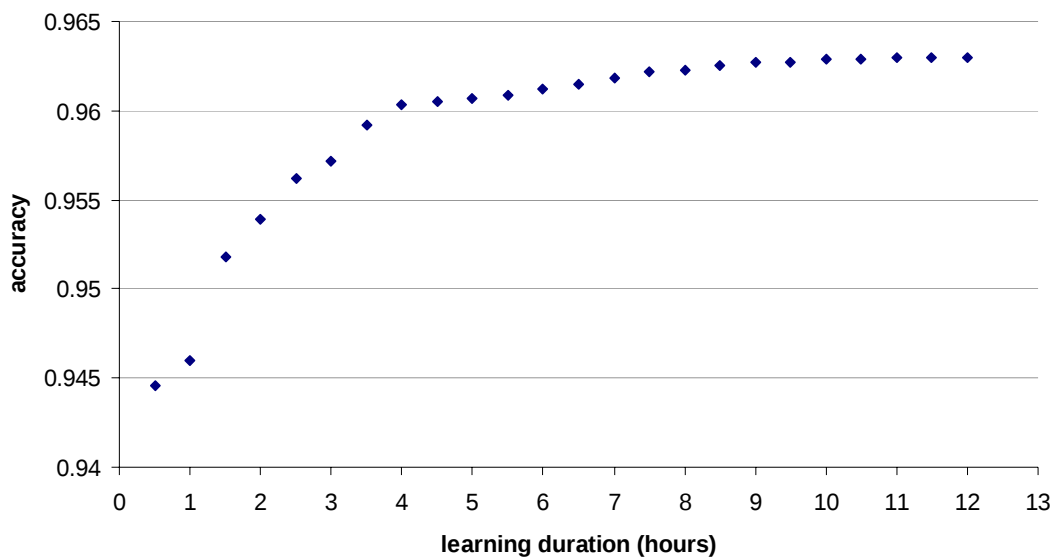


Figure 5.3 *The 12 hours' learning progress*

5.2 TAGGING SPEED

Generally speaking, a tagger's speed is less important than its accuracy. However, in some situations, such as real-time speech analysis, tagging speed is extremely important. Table 5.1 and Figure 5.2 display the following three results concerning the tagging speed on the testing computer. Note that the tagging speed is measured by the number of tokens tagged per second (abbreviated as *wps*), when the input is already a list of tokens. In other words, the time for tokenization and the time for displaying them are not included.

First, in terms of cross-data-type comparison, tagging the testing data is significantly slower than tagging the training data. Specifically, the raw tagger's speed dropped from 544,000 *wps* on the training data to 424,000 *wps* on the testing data, the trained tagger's speed dropped from 550,000 *wps* to 424,000 *wps*, and the final tagger's speed dropped from 403,000 *wps* to 251,000 *wps*. Converted to percentages, the speed of tagging the testing data is 22.1%, 22.9%, and 37.7% slower than that of tagging the training data for the raw tagger, the trained tagger, and the final tagger, respectively. The main reason is that extra computation time was spent in figuring out the tag(s) of the words that have not appeared in the training data.

Second, applying the contextual adjustments reduced the tagging speed from 550,000 *wps* to 403,000 *wps* when tagging the training data and from 424,000 *wps* to 251,000 *wps* when tagging the testing data. In other words, the tagging speed dropped by 26.7% when tagging the training data and by 40.8% when tagging the testing data because of applying the contextual adjustments.

Third, the final tagger can process 251,000 words per second on the testing data. This is the actual speed of the tagger because the real task of a tagger is to tag new text rather than tagging

the training data and tagging is an intermediate process, usually taking a list of words as input and producing a list of tagged words as output.

5.3 PORTABILITY AND TRAINABILITY

Generally speaking, probability-based taggers are portable to training data of other tagging schemes, other text types, or even other languages. A good tagger should be trainable as well. That is, given pre-tagged data of considerably big size, that tagger can learn from them to improve itself or even produce a new tagger. Building a trainable tagger is another important goal of this project.

I don't have tagged English texts of another domain or tagged corpora of another language. Fortunately, the online CLAWS tagger accepts up to 10,000 words to tag per upload (<http://ucrel.lancs.ac.uk/claws/trial.html>). Through this service, I have about 60% of the Brown corpus tagged by CLAWS4, using its C5 tagset. Since the C5 tagset is different from the Treebank tagset, the data thus tagged can be used to test the ability of the tagger-maker. I used the first 99% of the thus-obtained tagged data as the training data and the remaining 1% as the testing data. The whole tagger-making process was smooth and completely automatic. Table 5.2 lists the time needed to make a new final tagger. As shown there, it took about seven seconds to extract the data for the raw tagger, about 17 minutes to extract the contextual rules, and about four hours to complete the learning process.

The tagging speed is about the same as that of the tagger developed out of Treebank-3 (referred to as the main tagger hereafter). As shown in Table 5.3, the final accuracy on the testing data is 95.18%, which is lower than that of the main tagger. Learning and applying the contextual rules increased the accuracy by 3.34%, which is also lower than the increase produced by the same methods in the case of the main tagger. These are probably due to the much smaller

size of the training data. Nevertheless, the tagger-maker is proved working, which also shows that the main tagger is trainable.

Table 5.2 *The time needed for building a new tagger*

Procedure	Time needed
Producing the raw tagger	7 seconds
Building the context dictionary	17 minutes and 22 seconds
Learning	4 hours

Table 5.3 *The new tagger's accuracy results*

Stage	Accuracy
Before learning and before contextual adjustments (raw tagger)	91.84%
after learning but before contextual adjustments (trained tagger)	93.21%
after learning and after contextual adjustments (final tagger)	95.18%

CHAPTER 6

CONCLUSIONS

This project is devoted to the development of an efficient, scalable, and trainable tagger of high accuracy and the development of a tool that automatically turns a pre-tagged corpus into a tagger, regardless of the tag scheme, the text type or even the language of the corpus. This thesis illustrated in plain English the involved algorithms and their implementations to build a raw tagger, to train it by modifying the probability values, and to extract the contextual transformation rules to make the final adjustment. It also illustrated the procedures to build the tagger-maker.

In terms of the accuracy of the tagger built out of Treebank-3, the following conclusions can be arrived at.

First, the hidden Markov model used in this project and the Viterbi algorithm produced moderately acceptable tagging accuracy, namely, 94.08% on the training data and 92.30% on the testing data, even though both assumptions do not completely hold for the English language.

Second, there is still much room for improvement. The raw tagger's accuracy on the testing data is only 92.30%, which is not impressive at all. We must keep in mind that in most cases tagging is just an intermediate step. Other follow-up NLP activities may make errors themselves. Therefore, it is absolutely necessary that a tagger be as accurate as possible.

Third, it is completely possible to increase a probability-based tagger's accuracy by applying the contextual transformation rules, since there are always linguistic patterns that cannot be captured by probability-based methods. However, to do so, the tagger must know the

tags of the contextual words. The more accurate the surrounding words' tags are, the better result will be produced by applying these contextual constraints. One way to give the more accurate tags to the contextual words is to fine-tune the tag-word emission probabilities and the tags-tag transition probabilities through the trial-and-error machine learning approach, i.e., the reinforcement machine learning algorithm used in this project. In so doing, the probability values obtained by counting the frequencies of the tags in the training data are changed into the values that are tried in the real tagging process, which are naturally more accurate. Applying the contextual adjustments after the machine learning process boosted the tagger's accuracy by 4.21% on new texts, enabling its final accuracy to be 96.51%.

The learning algorithm is fruitful and efficient. It increased the raw tagger's accuracy by a little more than 2% during the first four hours. Considering the noise and errors existing in Treebank, no big progress can be expected once the accuracy reached about 96.30% and we should stop the learning activity to avoid the tagger from overfitting Treebank.

In terms of the tagging speed, two conclusions are reached. First, tagging the testing data is slower than tagging the training data by 22.1% for the raw tagger, 22.9% for the trained tagger, and 37.7% for the final tagger. The major reason is that it takes time to compute the tags of unknown words. Second, applying the contextual adjustments reduced the tagging speed by 26.7% on the training data and by 40.8% on the testing data. However, this loss of speed is affordable for the tagger, since it is extremely fast. The final tagger can tag about 251,000 words per second on the testing data, using the testing computer.

The tagger's trainability is proved by the fact that the tagger-maker automatically and successfully built a new tagger out of the data pre-tagged with a different tagset with good

testing result. This should be understandable if we think of the nature of the algorithms used in this project, as summarized below.

Firstly, Bayes' theorem is just a way of calculating the probability of a hypothesis, given its prior probability and the probability of observing the training data if that hypothesis holds. Therefore it is independent of the language to be tagged. Secondly, the hidden Markov model used in this project is just a finite-state machine, which is language-independent. Thirdly, the reinforcement machine learning approach is just a trial-and-error fine-tuning process, which is again independent of the features of the text to be tagged. Finally, that the part-of-speech of a word is influenced by those of its surrounding words is not peculiar to any particular language but a universal phenomenon and those contextual rules were not written by hand but obtained from the training data. These points predict the portability and the trainability of a tagger using these algorithms.

The tagging accuracy of 96.51% on the testing data indicates that the tagger is among the most accurate taggers. That it can tag 251,000 words per second on the testing data on the testing computer makes it the fastest tagger I have even seen. Finally, that the tagger is portable and trainable is proved by the tagger-maker's success. In sum, this project has achieved its goals.

BIBLIOGRAPHY

- Araujo, L. (2002). Part-of-speech tagging with evolutionary algorithms. *Lecture Notes in Computer Science*(2276), 230-239.
- Banko, M., & Robert, C. M. (2004). *Part of speech tagging in context*. Paper presented at the 20th international conference on Computational Linguistics, Geneva, Switzerland.
- Brill, E. (1995). Transformation-based error-driven learning and natural language processing: a case study in part-of-speech tagging. *Comput Linguist*(21), 543-565.
- Brill, E. (2000). Part-of-Speech Tagging. In R. Dale, H. Moisl & H. Somers (Eds.), *Handbook of Natural Language Processing*. New York: Marcel Dekker, Inc.
- Church, K. W. (1988). A stochastic parts program and noun phrase parser for unrestricted text. *Second conference on applied natural language processing*, 36-43.
- CLAWS4. Free CLAWS WWW trial service. Retrieved February 20, 2009, from <http://ucrel.lancs.ac.uk/claws/trial.html>
- Enrique, A., Luque, G., & Araujo, L. (2006). Natural language tagging with genetic algorithms. *Information Processing Letters*(100), 173-182.
- Forney, G. D. (1973). The Viterbi algorithm. *Proceedings of the IEEE*, 3(61), 268-278.
- Garside, R., Leech, G., & Sampson, G. (Eds.). (1987). *The Computation Analysis of English: A Corpus-based Approach*. London: Longman.

- Kim, J., Rim, H., & Tsujii, J. (2003). *Self-organizing Markov models and their application to part-of-speech tagging*. Paper presented at the 41st Annual Meeting on Association for Computational Linguistics, Sapporo, Japan.
- Klein, S., & Simmons, R. (1963). A computational approach to grammatical coding of English words. *J ACM*(10), 334-347.
- Lee, S., Tsujii, J., & Rim, H. (2000a). *Hidden Markov model-based Korean part-of-speech tagging considering high agglutinativity, word-spacing, and lexical correlativity*. Paper presented at the 38th Annual Meeting on Association for Computational Linguistics, Hong Kong.
- Lee, S., Tsujii, J., & Rim, H. (2000b). *Lexicalized hidden Markov models for part-of-speech tagging*. Paper presented at the 18th conference on Computational linguistics, Saarbrücken, Germany.
- Lee, S., Tsujii, J., & Rim, H. (2000c). *Part-of-speech tagging based on hidden Markov model assuming joint independence*. Paper presented at the 38th Annual Meeting on Association for Computational Linguistics, Hong Kong.
- Leech, G., Garside, R., & Bryant, M. (1994). *CLAWS4: the tagging of the British National Corpus*. Paper presented at the 15th conference on Computational linguistics, Kyoto, Japan.
- Marcus, M. D., Santorini, B., Marcinkiewicz, M. A., & Taylor, A. (1999). *Treebank-3: Linguistic Data Consortium*, Philadelphia.
- Marquez, L., Padro, L., & Rodriguez, H. R. (2000). A Machine Learning Approach to POS Tagging. *Machine Learning*(39), 59-91.

- Marshall, I. (1983). Choice of grammatical word-class without global syntactic analysis: tagging words in the Lob Corpus. *Computers and the Humanities*(17), 139-150.
- Mitchel, T. M. (1997). *Machine Learning*. Boston: WCB/McGraw-Hill.
- Ratnaparkhi, A. (1996). *A maximum entropy model for part-of-speech tagging*. Paper presented at the Conference on Empirical Methods in Natural Language Processing.
- Roche, E., & Schabes, Y. (1995). Deterministic Part-of-Speech Tagging with Finite-State Transducers. *Computational Linguistics*, 21(2), 227-253.
- Roth, D., & Zelenko, D. (1998). *Part of speech tagging using a network of linear separators*. Paper presented at the 36th Annual Meeting of the Association for Computational Linguistics and 17th International Conference on Computational Linguistics, Montreal, Quebec.
- Schmid, H. (1994). *Part-of-speech tagging with neural networks*. Paper presented at the 15th conference on Computational linguistics, Kyoto, Japan.
- Thede, S. M., & Harper, M. P. (1999). *A second-order Hidden Markov Model for part-of-speech tagging*. Paper presented at the 37th annual meeting of the Association for Computational Linguistics on Computational Linguistics, College Park, Maryland.
- Toutanova, K., & Christopher, D. M. (2000). *Enriching the Knowledge Sources Used in a Maximum Entropy Part-of-Speech Tagger*. Paper presented at the Joint SIGDAT Conference on Empirical Methods in Natural Language Processing and Very Large Corpora (EMNLP/VLC-2000).

APPENDICS

APPENDIX A: SOFTWARE DOCUMENTATION

This document describes the functionalities and the APIs of this software and illustrates how to use them.

A.1 RUN THE PROGRAM

To start the program, double click *HanTagger\bin\Debug\HanTagger.exe*.

A.2 FUNCTIONALITIES

The software provides the following nine functionalities.

A.2.1 Tagging the entered text

File → Tag input text

It tags any text in the left text box and the result appears in the right box, one token-tag pair per line (Figure A.1).

A.2.2 Tagging a file

File → Import file and tag it

You can choose any kind of text files. For testing purpose, I supplied the untagged 1-million-word Brown corpus. Its path is *HanTagger\data\rawBrown.txt*. If the file is not too big, the tagged text will appear in the right box; otherwise the tagged text will be saved in a file and its name and location will be shown in the right box instead. The time used will be reported as well, including the time on tokenization.

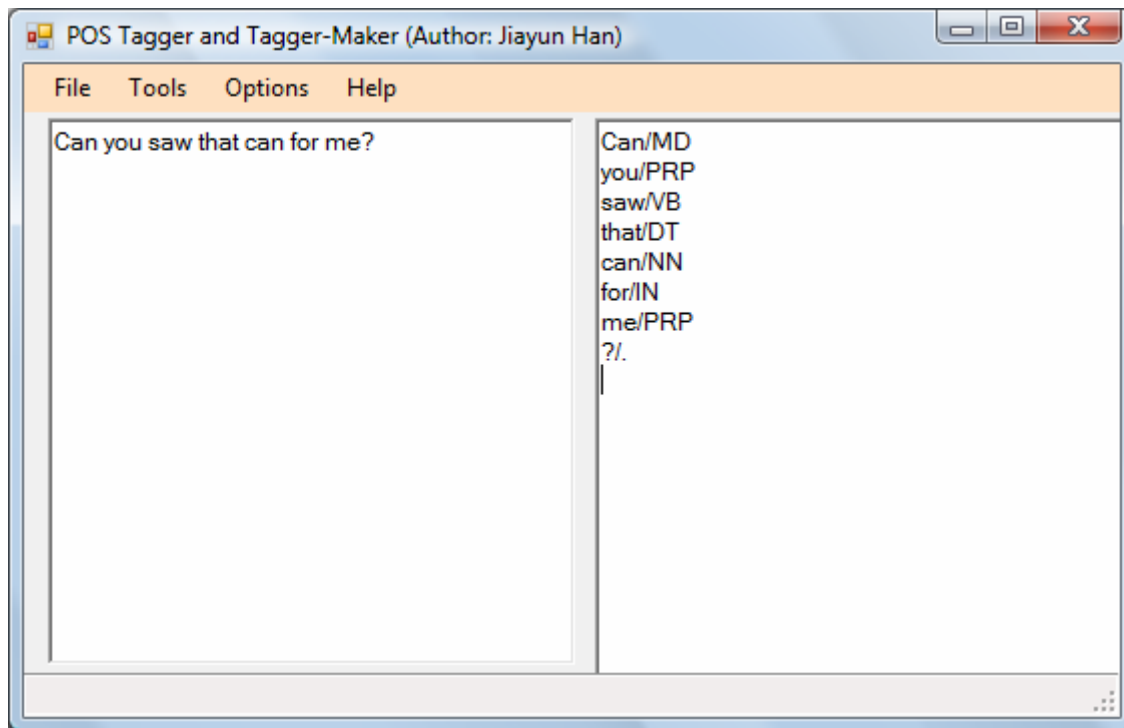


Figure A.1 *The screenshot of tagging the entered text*

A.2.3 Check tagging accuracy

Tools → *Check tagging accuracy*

The tagger will tag the words and compare the assigned tags with the corresponding tags in the pre-tagged file. The program reports three things: (1) the accuracy, (2) the actual tagging speed, and (3) lines of strings in the right box, each in the form of a token-tag pair given by the corpus followed by the token-tag pair produced by the tagger; if they don't match, the line is marked by <<<< *unmatched* (Figure A.2). Note that item (3) will not be reported if the file is too big.

The chosen file should use the same tagset as that used by the tagger. Currently the program supports two tagsets: that used by Treebank and the C5 tagset of CLAWS4. For more tagsets, you need to use this program to build a tagger out of them (See section 2.5 below).

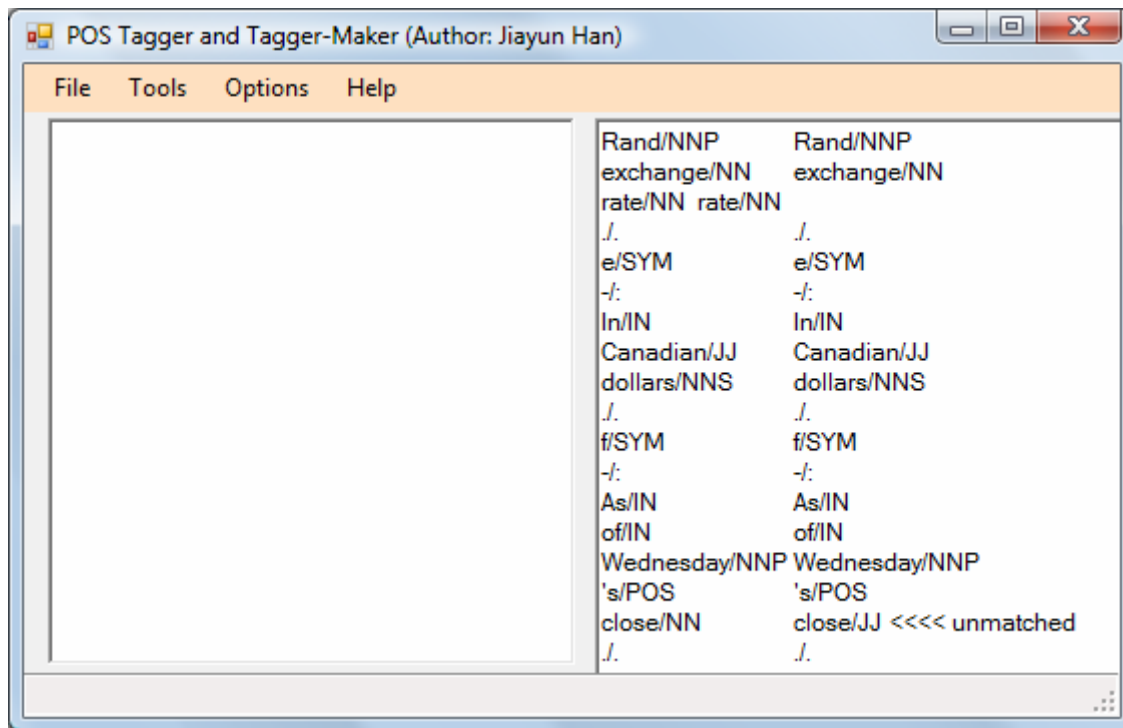


Figure A.2: The screenshot of checking tagging accuracy

For your convenience, I supplied the following four sets of files for accuracy checking.

(1) All Treebank testing files, located in: *HanTagger\data\PennTestingFiles*. These are the testing files set aside by Michael Covington.

(2) A big Treebank training file, which is made by merging all Treebank training files into a big string of token-tag pairs. Its path is *HanTagger\data\TRAINING_penn\tts_training.POS.TXT*.

Note that this file contains more than 4.5-million token-tag pairs and it takes about 25 - 60 seconds to finish checking the accuracy, depending on your computer's power and the tagging options you are using. As stated in the thesis, applying contextual adjustments slows down the speed by about one third.

(3) A single Treebank testing file, which is made by merging (1) into an 86,000 token-tag pairs. Its path is *HanTagger\data\PennTestingFiles\ tts_testing.txt*.

(4) A single file containing 60% of Brown corpus tagged by CLAWS4. Its path is
`\HanTagger\data\TRAINING_nonPenn\tts_training.txt`

A.2.4 Improving the accuracy of the tagger

Tools → *Improve accuracy*

This allows you to use the pre-tagged corpus to improve the tagger's accuracy. Clicking it will pop up the *Tagging and Learning Options* panel (Figure A.3) where you need to specify (1) the corpus source and (2) whether to use the default training parameter values. If you choose setting your own parameters instead, the *Set Training Parameter Values* panel will show up (Figure A.4).

The corpus you specified must (a) use the same tagset as the tagger to be trained and (b) be big enough to prevent the tagger from overfitting this small corpus.

Training parameters and their meanings

If you tell the program that you want to set the training parameter values by yourself, the *Set Training Parameter Values* panel (Figure 4) will show up. The parameters and their values are explained below.

Error Tolerance: if a tagging error's frequency is smaller than this value, the error will not be corrected. The bigger this value is, the quicker the learning is, of course with less obvious improvements. The smaller it is, the stricter the learning it is. This value decreases by 20 for each learning generation.

Accuracy Goal: learning stops if the accuracy has reached this value.

Generations to stop: learning stops if the number of learning cycles has reached this value.

Strong Tax Rate: the amount to be deducted from a strong tag's emission or transition probability is calculated by multiplying its old probability by this value, which is 0 to 1,

exclusive. If it is too small, learning will be very slow but will ultimately take place. However, if it is too big, learning will never happen. The default value is 0.1, which means a strong tag's probability deduction is one tenth of its own probability.

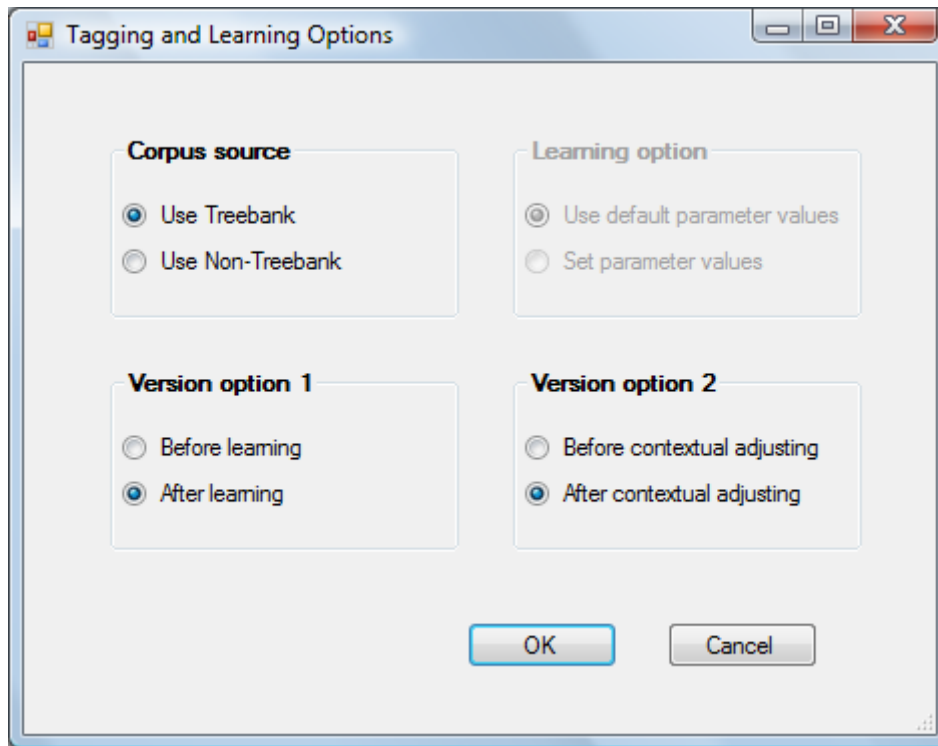


Figure A.3: The screenshot of tagging and Training Options panel

Common Tax Rate: the amount to be deducted from a common tag's emission or transition probability is calculated by multiplying its old probability by *Strong Tax Rate* and by this value, which is 0 to 1, inclusive. If it is 1, the deduction value is the same as the strong tag's deduction value; if it is 0, the deduction value is zero. The default value is 0.2, which means a common tag's deduction is one fifth of the strong tag's deduction.

Training file: the tagger learns from this corpus. It must be big enough (the current threshold is half million words) and use the same tagset as the tagger to be trained.

A.2.5 Making a new tagger

Tools → *Build a new tagger*

This function turns a pre-tagged corpus of considerably big size into a new tagger. You need to tell the program whether the corpora use the same tagging scheme as that used by Treebank and tell the program the training corpus' path.

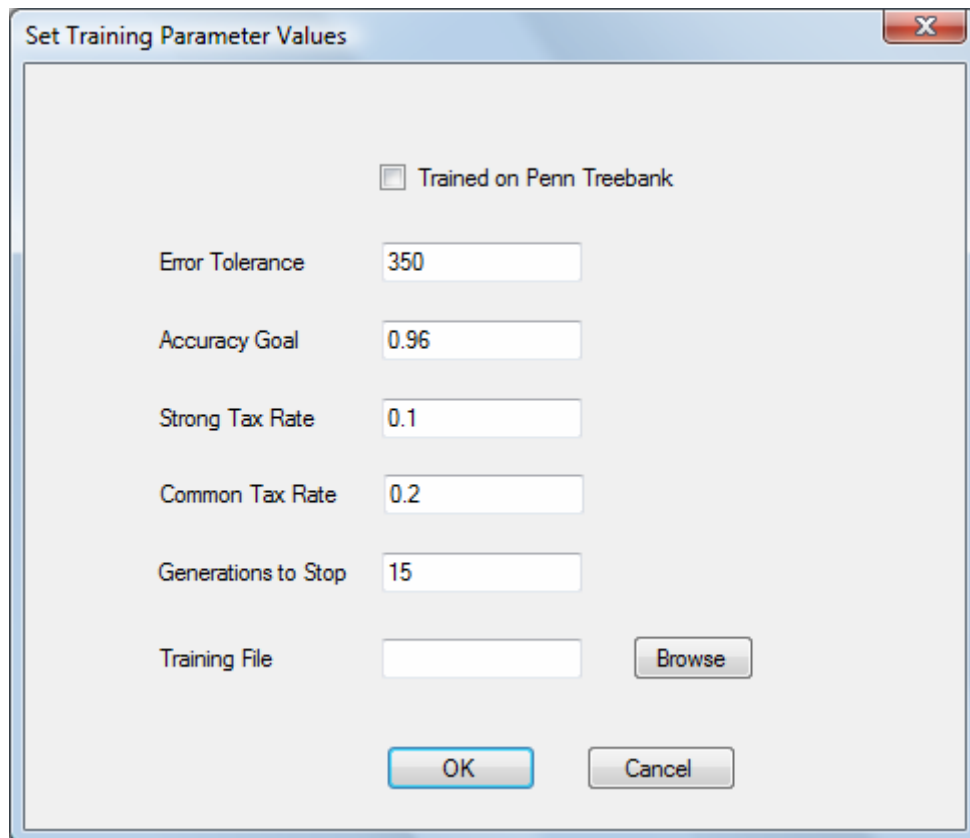


Figure A.4: *The screenshot of setting the training parameters*

After that, it takes about 5 – 60 seconds to build the raw tagger.

Once the raw tagger is built, the program will ask you whether to allow it to extract the contextual rules. These rules are used for contextual adjustments.

Finally, the program will ask you whether to allow it to train the tagger. Training is necessary but it takes up to five hours.

A.2.6 Setting Tagging options

Options → *Set tagging options*

This allows you to specify the tagger type you want to test. By clicking different radio buttons, you have eight tagger versions (See Figure A.3). The default is the most accurate one which was trained on Treebank and which applies the contextual adjustments.

These options are there only for academic purposes, i.e., for checking the tagging differences produced by training and by applying contextual constraints. In the real tagging situations, we will use the best one, of course.

Corpus Source: this determines which tagging scheme to use. Currently I supplied two: the Treebank and the C5 of CLAWS4.

Version option 1: this allows us to see the tagging difference produced by learning.

Version option 2: this allows us to see the tagging improvements obtained from performing the contextual constraints.

A.2.7 Other functions:

The other three functions under *File* are trivial and self-explanatory: *Save input text*, *Save output text*, and *Clear*, the last clearing the texts of both text boxes.

A.3 API's PROVIDED TO PROGRAMMERS

This program provides the following nine APIs which can be incorporated into your program. In each API, the first is the data type to be returned.

A.3.1 Parameters and their values

- o *file* is the path of the file to be tagged
- o *connector* is the string used to separate one token-tag pair from another (default = “\r\n”)

- o *toLower* = true changes the text to lower case while *toLower* = false does nothing. The default value = false. This is used in tokenization.
- o *keepPunc* = true keeps the punctuation marks while *keepPunc* = false removes the punctuations from the text. The default value = true. This is used in tokenization.
- o *keepDigits* = true keeps the digits while *keepDigit* = false removes the digits from the text. The default value = true. This is used in tokenization.

A.3.2 APIs

- o List<string> TagFile_toList(string *file*, bool *toLower*, bool *keepPuncs*, bool *keepDigits*)
- o List<string> TagFile_toList(string *file*), which is the same as TagFile_toList (*file*, *false*, *true*, *true*)
- o string TagFile_toStr(string *file*, bool *toLower*, bool *keepPuncs*, bool *keepDigits*, string *connector*)
- o string TagFile_toStr(string *file*, string *connector*) , which is the same as TagFile_toStr (*file*, *false*, *true*, *true*, *connector*)
- o List<string> TagList_toList(List<string> *tokens*)
- o List<string> TagStr_toList(string *str*, bool *toLower*, bool *keepPuncs*, bool *keepDigits*)
- o string TagStr_toList(string *str*) , which is the same as = TagStr_toList (*str*, *false*, *true*, *true*)
- o string TagStr_toStr(string *str*, bool *toLower*, bool *keepPuncs*, bool *keepDigits*, string *connector*)
- o string TagStr_toStr(string *str*, string *connector*), which is the same as TagStr_toStr *str*, *false*, *true*, *true*, *connector*)

A.4 HOW TO INCORPORATE THIS PROGRAM INTO YOUR OWN

To use the above APIs, follow these steps: (1) unzip the package *APIs and Data.zip*, which produces *HanLibrary.dll* and the *data* folder, (2) add *HanLibrary.dll* to your references, (3) add directive *using HanLibrary.PosTagging* to your main class, (4) create a *HTagger* instance, using the default constructor (a) or the overloaded constructor (b), and (5) call the methods needed by your program that are provided by the *HTagger* instance.

(a) *HTagger tagger = new HTagger();*

(b) *HTagger tagger = new HTagger(folder_name_and_path);*

If you use the default constructor, you need to copy the *data* folder with its files and place it two levels up to your main C# executable. Suppose your main C# executable is called *Tagger.exe*, located in *bin\Debug*, then the directory layout is:

- *data*
- *bin*
- *Debug*
- *Tagger.exe*

If you use the overloaded constructor, *folder_name_and_path* is the name and path of your own folder to host the data files needed by the tagger. You can use any name you like. The path of your folder can be specified absolutely such as “C:\\Users\\Me\\MyData\\” or relatively to the location of your main executable, such as *HTagger tagger = new HTagger(“...\\...\\..\\MyData\\”)*. If you forgot the ending slashes, the program will supply them. You need to copy the data files and place them inside your own data folder.

APPENDIX B: PENN TREEBANK TAGSET

(Source: <http://www.mozart-oz.org/mogul/doc/lager/brill-tagger/penn.html>)

POS Tag	Description	Example
CC	coordinating conjunction	and
CD	cardinal number	1, third
DT	determiner	the
EX	existential there	there
FW	foreign word	d'hoevre
IN	preposition/subordinating conjunction	in, of, like
JJ	adjective	green
JJR	adjective, comparative	greener
JJS	adjective, superlative	greenest
LS	list marker	1)
MD	modal	could, will
NN	noun, singular or mass	table
NNS	noun plural	tables
NNP	proper noun, singular	John
NNPS	proper noun, plural	Vikings
PDT	predeterminer	<i>both</i> the boys
POS	possessive ending	friend's
PRP	personal pronoun	I, he, it
PRP\$	possessive pronoun	my, his
RB	adverb	however, usually, naturally, here, good
RBR	adverb, comparative	better
RBS	adverb, superlative	best
RP	particle	<i>give up</i>
TO	to	<i>to go, to him</i>
UH	interjection	uhhuhhuhh
VB	verb, base form	take
VBD	verb, past tense	took
VBG	verb, gerund/present participle	taking
VBN	verb, past participle	taken
VBP	verb, sing. present, non-3d	take
VBZ	verb, 3rd person sing. present	takes
WDT	wh-determiner	which
WP	wh-pronoun	who, what
WP\$	possessive wh-pronoun	whose
WRB	wh-abverb	where, when

APPENDIX C: THE C5 TAGSET OF CLAWS4

(Source: <http://ucrel.lancs.ac.uk/claws5tags.html>)

POS Tag	Description	Example
AJ0	adjective (unmarked)	good, old
AJC	comparative adjective	better, older
AJS	superlative adjective	best, oldest
AT0	article	the, a, an
AV0	adverb (unmarked)	often, well, longer, furthest
AVP	adverb particle	up, off, out
AVQ	wh-adverb	when, how, why
CJC	coordinating conjunction	and, or
CJS	subordinating conjunction	although, when
CJT	the conjunction	that
CRD	cardinal numeral	3, fifty-five, 6609 (excl one)
DPS	possessive determiner form	your, their
DT0	general determiner	these, some
DTQ	wh-determiner	whose, which
EX0	existential	there
ITJ	interjection or other isolate	oh, yes, mhm
NN0	noun (neutral for number)	aircraft, data
NN1	singular noun	pencil, goose
NN2	plural noun	pencils, geese
NP0	proper noun	london, michael, mars
NULL	the null tag (for items not to be tagged)	
ORD	ordinal	sixth, 77th, last
PNI	indefinite pronoun	none, everything
PNP	personal pronoun	you, them, ours
PNQ	wh-pronoun	who, whoever
PNX	reflexive pronoun	itself, ourselves
POS	the possessive (or genitive morpheme) 'S or '	
PRF	the preposition OF	
PRP	preposition (except for OF)	for, above, to
PUL	punctuation - left bracket	(, [
PUN	punctuation - general mark	! , ; - ? ...
PUQ	punctuation - quotation mark	` ' "
PUR	punctuation - right bracket	),]
TO0	infinitive marker TO	
UNC	"unclassified" items which are not words of the English lexicon	

APPENDIX C: THE C5 TAGSET OF CLAWS4 (CONTINUED)

(Source: <http://ucrel.lancs.ac.uk/claws5tags.html>)

POS Tag	Description	Example
VBB	the "base forms" of the verb "BE" (except the infinitive)	am, are
VBD	past form of the verb "BE"	was, were
VBG	-ing form of the verb "BE"	being
VBI	infinitive of the verb "BE"	
VBN	past participle of the verb "BE"	been
VBZ	-s form of the verb "BE"	is, 's
VDB	base form of the verb "DO" (except the infinitive), i.e.	
VDD	past form of the verb "DO"	did
VDG	-ing form of the verb "DO"	doing
VDI	infinitive of the verb "DO"	do
VDN	past participle of the verb "DO"	done
VDZ	-s form of the verb "DO"	does
VHB	base form of the verb "HAVE" (except the infinitive), i.e. HAVE	have
VHD	past tense form of the verb "HAVE"	had, 'd
VHG	-ing form of the verb "HAVE"	having
VHI	infinitive of the verb "HAVE"	have
VHN	past participle of the verb "HAVE"	had
VHZ	-s form of the verb "HAVE"	has, 's
VM0	modal auxiliary verb	can, could, will, 'll
VVB	base form of lexical verb (except the infinitive)	take, live
VVD	past tense form of lexical verb	took, lived
VVG	-ing form of lexical verb	taking, living
VVI	infinitive of lexical verb	take, live
VVN	past participle form of lexical verb	taken, lived
VVZ	-s form of lexical verb	takes, lives
XX0	the negative NOT or N'T	
ZZ0	alphabetical symbol	a, b, c, d