CHAPTER 3

WORD LEVEL ANALYSIS

CHAPTER OVERVIEW

This chapter focuses on processing carried out at word level, including methods for characterizing word sequences, identifying morphological variants, detecting and correcting misspelled words, and identifying correct part-of-speech of a word. The part-of-speech tagging methods covered in this chapter are: rule-based (linguistic), Stochastic (data-driven), and hybrid.

3.1 INTRODUCTION

As discussed in Chapter 1, natural language processing (NLP) involves different levels and complexities of processing. One way to analyse natural language text is by breaking it down into constituent units (words, phrases, sentences, and paragraphs) and then analyse these units. In Chapter 2, we discussed various language models that are used for analysing the syntax of natural language sentences. Before analysing syntax, we need to understand words, as words are the fundamental unit (syntactic as well as semantic) of any natural language text. This chapter focuses on NLP carried out at word level, including characterizing word sequences, identifying morphological variants, detecting and correcting misspelled words, and identifying correct part-of-speech of a word.

Regular expressions are a beautiful means for describing words. In many text applications, we wish to work with string patterns. Suppose you have just come across the word 'supernova'. It catches your interest and you jump to a search engine to find out more on 'supernovas'. But you do not know whether to type in 'supernova', 'Supernova', or 'supernovas'. Obviously, you need a system, which will retrieve relevant articles using any one of these word forms. Regular expressions are used for describing text strings in situations like this and in other information

retrieval applications. In this chapter, we introduce regular expressions and discuss standard notations for describing text patterns.

After defining regular expressions, we discuss their implementation using finite-state automaton (FSA). Readers who have gone through a course in formal language theory will be familiar with FSA. The FSAs and their variants, such as finite state transducers, have found useful applications in speech recognition and synthesis, spell checking, and information extraction. As we will be using FSA throughout this book, we formally define FSAs in this chapter.

Errors in typing and spelling are quite common in text processing applications. Detecting and correcting these errors are the next topics of discussion in this chapter. Numerous web pages also contain misspelled words and often, query terms entered into the search engines are misspelled. An interactive spelling facility that informs users of such errors and presents appropriate corrections to them, is useful in these applications.

There are different classes of words. A word has many forms and the same word may have many different meanings depending on the context. Identifying the class to which a word belongs, its basic form, and its meaning are crucial to analysing text. This is the last topic covered in this chapter.

3.2 REGULAR EXPRESSIONS

Regular expressions, or regexes for short, are a pattern-matching standard for string parsing and replacement. They are a powerful way to find and replace strings that take a defined format. For example, regular expressions can be used to parse dates, urls and email addresses, log files, configuration files, command line switches, or programming scripts. They are useful tools for the design of language compilers and have been used in NLP for tokenization, describing lexicons, morphological analysis, etc. We have all used simplified forms of regular expressions, such as the file search patterns used by MS DOS, e.g., dir*.txt.

The use of regular expressions in computer science was made popular by a Unix-based editor, 'ed'. Perl was the first language that provided integrated support for regular expressions. It used a slash around each regular expression; we will follow the same notation in this book. However, slashes are not a part of regular expressions.

Regular expressions were originally studied as a part of theory of computation. They were first introduced by Kleene (1956). A regular expression is an algebraic formula whose value is a pattern consisting of a

set of strings, called the language of the expression. The simplest kind of regular expression contains a single symbol. For example, the expression

/a/

denotes the set containing the string 'a'. A regular expression may specify a sequence of characters also. For example, the expression

/supernova/

denotes the set that contains the string 'supernova' and nothing else. In a search application, the first instance of each match to regular expression is underlined in Table 3.1.

Table 3.1 Some simple regular expressions

Regular expression	Example patterns
/book/	The world is a book, and those who do not travel read only one page.
/book/	Reporters, who do not read the stylebook, should not criticize their editors.
/face/	Not everything that is faced can be changed. But nothing can be changed until it is faced.
/a/	Reason, Observation, and Experience—the Holy Trinity of Science.

3.2.1 Character Classes

Characters are grouped by putting them between square brackets. This way, any character in the class will match one character in the input. For example, the pattern /[abcd]/ will match (any of) a, b, c, and d. The use of brackets specifies a disjunction of characters. The regular expression /[0123456789]/ specifies any single digit. The character classes are important building blocks in expressions. They sometimes lead to cumbersome notation. For example, it is inconvenient to write the regular expression

/[abcdefghijklmnopqrstuvwxyz]/

to specify 'any lowercase letter'. In these cases, a dash is used to specify a range. The regular expression /[5–9]/ specifies any one of the characters 5, 6, 7, 8, or 9. The pattern /[m–p]/ specifies any one of the letter m, n, o, or p.

Regular expressions can also specify what a single character cannot be, by the use of a caret at the beginning. For example, the pattern /[^x]/ matches any single character except x. This interpretation is true only when a caret appears as the first symbol. If it occurs at any other place, it refers to the caret symbol itself. Table 3.2 shows a few examples explaining these concepts.

Table 3.2 Use of square brackets

RE	Match	Example patterns matched
[abc]	Match any of a, b, and c	'Refresher course will start tomorrow'
[A–Z]	Match any character between A and Z (ASCII order)	'the course will end on Jan. 10, 2006'
[^A–Z]	Match any character other than an uppercase letter	'TREC Conference'
[^abc]	Match anything other than a, b, and c	'TREC Conference'
[+*?.]	Match any of +, *, ?, or the dot.	'3 + 2 = 5'
[a^]	Match a or ^	'^ has three different uses.'

Regular expressions are *case sensitive*. The pattern /s/ matches a lower case 's' but not an uppercase 'S'. This means that the pattern /sana/ will not match the string /Sana/. This problem can be solved by using the disjunction of character s and S. The pattern /[sS]/ will match the strings containing either 's' or 'S'. The pattern /[sS]ana/ matches with the string 'sana' or 'Sana'.

While the use of square brackets solves the capitalization problem, we still need a solution for how to specify both 'supernova' and 'supernovas'. The pattern /[sS]upernova[sS]/ matches with any of the strings 'supernovas', 'supernovas', 'Supernovas', and 'SupernovaS', but not with the string 'supernova'. This is achieved with the use of a question mark /?/. A question mark makes the preceding character optional, i.e., zero or one occurrence of the previous character. The regular expression

/supernovas?/

specifies both 'supernova' and 'supernovas'. Often, we need to specify repeated occurrences of a character. The * operator, called the *Kleene* * (pronounced 'cleany star'), allow us to do this. The * operator specifies zero or more occurrences of a preceding character or regular expression. The regular expression /b*/ will match any string containing zero or more occurrences of 'b', i.e., it will match 'b', 'bb', or 'bbbb'. It will also match 'aaa', since that string contains zero occurrences of 'b'. To match a string containing one or more 'b's, the regular expression is /bb*/. The regular expression /bb*/ means a 'b' followed by zero or more 'b's. This will match with any of the strings 'b', 'bb', 'bbb', 'bbbb'. Similarly, the regular expression /[ab]*/ specifies 'zero or more "a"s or "b"s. This will match strings like 'aa', 'bb', or 'abab'. The kleene+ provides a shorter notation to specify one or more occurrences of a character. The regular

expression /a+/ means one or more occurrences of 'a'. Using *Kleene+*, we can specify a sequence of digits by the regular expression /[0–9]+/. Complex regular expressions can be built up from simpler ones by means of regular expression operators.

The caret (^) is also used as an anchor to specify a match at the beginning of a line. The dollar sign, $, is an anchor that is used to specify a match at the end of the line. ^ and $ are important to regexes. If you wish to search for a line containing only the phrase 'The nature.' and nothing else, you need to specify a regular expression for this search. The anchors ^ and $ are of great help in this case. The regular expression /^The nature\.$/ will search exactly for this line.

A number of special characters are also used to build regular expressions. One such character is the dot (.), called wildcard character, which matches any single character. The wildcard expression /./ matches any single character. The regular expression /.at/ matches with any of the string cat, bat, rat, gat, kat, mat, etc. It will also match the meaningless words such as nat, 4at, etc. Table 3.3 shows some of the special characters and their likely use.

Table 3.3 Some special characters

RE	Description
.	The dot matches any single character.
\n	Matches a new line character (or CR+LF combination).
\t	Matches a tab (ASCII 9).
\d	Matches a digit [0–9].
\D	Matches a non-digit.
\w	Matches an alphanumeric character.
\W	Matches a non-alphanumberic character.
\s	Matches a whitespace character.
\S	Matches a non-whitespace character.
\	Use \ to escape special characters. For example, \. matches a dot, * matches a * and \\ matches a backslash.

We can also use the wildcard symbol for counting characters. For instance /.....berry/ matches ten-letter strings that end in berry. This finds patterns like strawberry, sugarberry, and blackberry but fails to match with blueberry or hackberry.

Suppose you are searching a text for the presence of 'Tanveer' or 'Siddiqui'. You cannot use square brackets for this. You need a disjunction operator, shown by the pipe symbol(|). The pattern blackberry|blackberries matches either 'blackberry' or 'blackberries'. You might prefer to do this

matching by merely writing blackberry|ies. Unfortunately, this does not work. The pattern blackberry|ies matches with either 'blackberry' or 'ies'. This is because sequences take precedence over disjunction. In order to apply the disjunction operator to a specific pattern, we need to enclose it within parentheses. The parenthesis operator makes it possible to treat the enclosed pattern as a single character for the purposes of neighboring operators. Now, we will consider an example from real application.

Example 3.1 Suppose we need to check if a string is an email address or not. An email address consist of a non-empty sequence of characters followed by the 'at' symbol, @, followed by another non-empty sequence of characters ending with pattern like .xx, .xxx, .xxxx, etc. The regular expression for an email address is

$$\text{^[A-Za-z0-9\_\.-]}^{+}\text{+@[A-Za-z0-9\_\.-]}^{+}\text{+[A-Za-z0-9\_][A-Za-z0-9\_]\$}$$

Table 3.4 shows the various parts of this 'rgex'.

Table 3.4 Parts of regular expression of Example 3.1

Pattern	Description
^[A-Za-z0-9_\.-]+	Match a positive number of acceptable characters at the start of the string.
@	Match the @ sign.
[A-Za-z0-9_\.-]+	Match any domain name, including a dot.
[A-Za-z0-9_][A-Za-z0-9_]$	Match two acceptable characters but not a dot. This ensures that the email address ends with .xx, .xxx, .xxxx, etc.

This example works for most cases. However, the specification is not based on any standard and may not be accurate enough to match all correct email addresses. It may accept non-working email addresses and reject working ones. Fine-tuning is required for accurate characterization.

A regular expression characterizes a particular kind of formal language, called a regular language. The language of regular expressions is similar to formulas of Boolean logic. Like logic formulas, regular expressions represent sets. Regular language is set of strings described by the regular expression. Regular languages may be encoded as finite state networks.

A regular expression may contain symbol pairs. For example, the regular expression /a:b/ represents a pair of string. The regular expression /a:b/ actually denotes a regular relation. A regular relation may be viewed as a mapping between two regular languages. The a:b relation is simply the cross product of the languages denoted by the expressions /a/ and /b/. To differentiate the two languages that are involved in a regular relation, we call the first one, the upper, and the second one, the lower

language, of the relation. Similarly, in the pair /a:b/, the first symbol, a, can be called the upper symbol and the second symbol, b, the lower symbol. The two components of a symbol pair are separated in our notation by a colon (:) without any whitespace before or after. To make the notation less cumbersome, we ignore the distinction between the language A and the identity relation that maps every string of A to itself. Therefore, we also write /a:a/ simply as /a/.

Regular expressions have clean, declarative semantics (Voutilainen 1996). Mathematically, they are equivalent to finite automata, both having the same expressive power. This makes it possible to encode regular languages using finite-state automata, leading to easier manipulation of context free and other complex languages. Similarly, regular relations can be represented using finite-state transducers. With this representation, it is possible to derive new regular languages and relations by applying regular operators, instead of re-writing the grammars.

3.3 FINITE-STATE AUTOMATA

In our childhood, each of us must have played some game that fits the following description:

Pieces are set up on a playing board; dice are thrown or a wheel is spun, and a number is generated at random. Based on the number appearing on the dice, the pieces on the board are rearranged specified by the rules of the game. Then, it is your opponent's turn; she also does the same thing, resulting in rearrangement of the pieces based on the number generated. There is no skill or choice involved. The entire game is based on the values of the random numbers.

Consider all possible positions of the pieces on the board and call them *states*. The state in which the game begins is termed the *initial state*, and the state corresponding to the winning positions is termed the *final state*. We begin with the initial state of the starting positions of the pieces on the board. The game then changes from one state to another based on the value of the random number. For each possible number, there is one and only one resulting state given the input of the number and the current state. This continues until one player wins and the game is over. This is called a final state.

Now consider a very simple machine with an input device, a processor, some memory, and an output device. The machine starts in the initial state. It checks the input and goes to next state, which is completely determined by the prior state and the input. If all goes well, the machine

reaches final state and terminates. If the machine gets an input for which the next state is not specified, and it gets stuck at a non-final state, we say the machine has failed or rejected the input.

A general model of this type of machine is called a finite automaton; 'finite' because the number of states and the alphabet of input symbols is finite; 'automaton' because the machine moves automatically, i.e., the change of state is completely governed by the input. This type of machine is more commonly called deterministic.

A finite automaton has the following properties:

1. A finite set of states, one of which is designated the *initial* or *start state*, and one or more of which are designated as the *final states*.
2. A finite alphabet set, Σ, consisting of input symbols.
3. A finite set of *transitions* that specify for *each* state and *each* symbol of the input alphabet, the state to which it next goes.

A finite automaton can be deterministic or non-deterministic. In a non-deterministic automaton, more than one transition out of a state is possible for the same input symbol.

Example 3.2 Suppose $\Sigma = \{a, b\}$, the set of states $= \{q_0, q_1, q_2, q_3, q_4\}$ with q_0 being the start state and q_4 the final state, we have the following rules of transition:

1. From state q_0 and with input a, go to state q_1.
2. From state q_1 and with input b, go to state q_2.
3. From state q_1 and with input c, go to state q_3.
4. From state q_2 and with input b, go to state q_4.
5. From state q_3 and with input b, go to state q_4.

This finite-state automaton is shown as a directed graph, called transition diagram, in Figure 3.1. The nodes in this diagram correspond to the states, and the arcs to transitions. The arcs are labelled with inputs. The final state is represented by a double circle. As seen in the figure, there is exactly one transition leading out of each state. Hence, this automaton is deterministic.

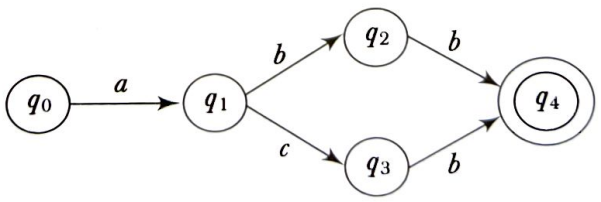

Figure 3.1 A deterministic finite-state automaton (DFA)

Finite-state automata have been used in a wide variety of areas, including linguistics, electrical engineering, computer science, mathematics, and logic. These are an important tool in computational linguistics and have been used as a mathematical device to implement regular expressions. Any regular expression can be represented by a finite automaton and the language of any finite automaton can be described by a regular expression. Both have the same expressive power. The following formal definitions of the two types of finite state automaton, namely, deterministic and non-deterministic finite automaton, are taken from Hopcroft and Ullman (1979).

A *deterministic finite-state automaton* (DFA) is defined as a 5-tuple $(Q, \Sigma, \delta, S, F)$, where Q is a set of *states*, Σ is an alphabet, S is the *start* state, $F \subseteq Q$ is a set of *final* states, and δ is a *transition function*. The transition function δ defines mapping from $Q \times \Sigma$ to Q. That is, for each state q and symbol a, there is at most one transition possible as shown in Figure 3.1.

Unlike DFA, the transition function of a *non-deterministic finite-state automaton* (NFA) maps $Q \times (\Sigma \cup \{\varepsilon\})$ to a subset of the power set of Q. That is, for each state, there can be more than one transition on a given symbol, each leading to a different state.

This is shown in Figure 3.2, where there are two possible transitions from state q_0 on input symbol a.

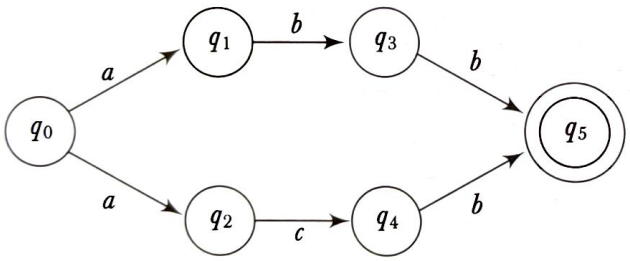

Figure 3.2 Non-deterministic finite-state automaton (NFA)

A *path* is a sequence of transitions beginning with the start state. A path leading to one of the final states is a *successful path*. The FSAs encode *regular* languages. The language that an FSA encodes is the set of strings that can be formed by concatenating the symbols along each successful path. Clearly, for automata with cycles, these sets are not finite.

We now examine what happens to various input strings that are presented to finite state automata. Consider the deterministic automaton described in Example 3.2 and the input, *ac*. We start with state q_0 and go to state q_1. The next input symbol is *c*, so we go to state q_3. No more input is left and we have not reached the final state, i.e., we have an unsuccessful end. Hence, the string *ac* is not recognized by the automaton.

This example illustrates how an FSA can be used to accept or recognize a string. The set of all strings that leave us in a final state is called the language accepted or defined by the FA. This means *ac* is not a word in the language defined by the automaton of Figure 3.1.

Now, consider the input acb. Again, we start with state q_0 and go to state q_1. The next input symbol is *c*, so we go to state q_3. The next input symbol is *b*, which leads to state q_4. No more input is left and we have reached to final state, i.e., this time we have a successful termination. Hence, the string *acb* is a word of the language defined by the automaton.

The language defined by this automaton can be described by the regular expression */abb|acb/*.

The example considered here is quite simple. Typically, the list of transition rules can be quite long. Listing all transition rules may be inconvenient, so often we represent an automaton as a *state-transition table*. The rows in this table represent states and the columns correspond to input. The entries in the table represent the transition corresponding to a given state-input pair. A ϕ entry indicates missing transition. This table contains all the information needed by FSA. The state transition table for the automaton considered in Example 3.2 is shown in Table 3.5.

Table 3.5 The state-transition table for the DFA shown in Figure 3.1

		Input		
State		***a***	***b***	***c***
start:	q_0	q_1	ϕ	ϕ
	q_1	ϕ	q_2	q_3
	q_2	ϕ	q_4	ϕ
	q_3	ϕ	q_4	ϕ
final:	q_4	ϕ	ϕ	ϕ

Now, consider a language consisting of all strings containing only *a*s and *b*s and ending with *baa*. We can specify this language by the regular expression */(a|b)*baa$/*. The NFA implementing this regular expression is shown in Figure 3.3.

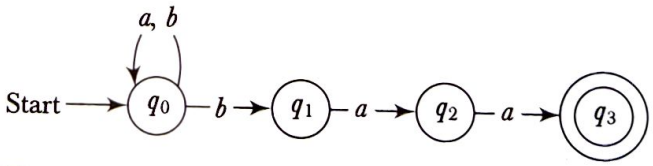

Figure 3.3 NFA for the regular expression /(a|b)*baa$/

Table 3.6 The state-transition table for the NFA shown in Figure 3.3.

		Input	
State		***a***	***b***
start:	q_0	$\{q_0\}$	$\{q_0, q_1\}$
	q_1	$\{q_2\}$	ϕ
	q_2	$\{q_3\}$	ϕ
final:	q_3	ϕ	ϕ

Two automata that define the same language are said to be *equivalent.* An NFA can be converted to an equivalent DFA and vice versa. The equivalent DFA for the NFA shown in Figure 3.3 is shown in Figure 3.4.

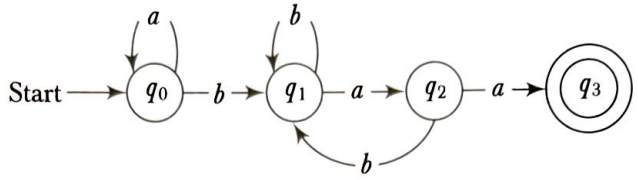

Figure 3.4 Equivalent DFA for NFA in Figure 3.3

3.4 MORPHOLOGICAL PARSING

Morphology is a sub-discipline of linguistics. It studies word structure and the formation of words from smaller units (morphemes). The goal of morphological parsing is to discover the morphemes that build a given word. Morphemes are the smallest meaning-bearing units in a language. For example, the word 'bread' consists of a single morpheme and 'eggs' consist of two: the morpheme egg and the morpheme -s. A morphological parser should be able to tell us that the word 'eggs' is the plural form of the noun stem 'egg'.

There are two broad classes of morphemes: stems and affixes. The stem is the main morpheme, i.e., the morpheme that contains the central meaning. Affixes modify the meaning given by the stem. Affixes are divided into prefix, suffix, infix, and circumfix. Prefixes are morphemes which appear before a stem, and suffixes are morphemes applied to the end of the stem. Circumfixes are morphemes that may be applied to either end of the stem while infixes are morphemes that appear inside a stem. Prefixes and suffixes are quite common in Urdu, Hindi, and English. For example, the Urdu word, بےوقت (*bewaqt*), meaning untimely, is composed of the stem, *waqt*, and the prefix, *be-*, گھوڑوں (*ghodhon*) is composed of the stem, گھوڑا (*ghodha*) and the suffix, وں (*on*). Also, the word, ضرورت‌مند (*zarooratmand*), is composed of the stem, ضرورت

(*zaroorat*), and the suffix, مند (*mand*). Similarly, the English word, unhappy, is composed of the stem, happy, and the prefix, un-. The English word, birds, is composed of the stem, bird, and the suffix, -s. Likewise, the Hindi word, शीतलता is composed of a stem शीतल and the suffix – ता.

Telgu word గుర్రములు (*gurramulu*–plural form of *gurramu*, meaning horse) is composed of the stem గుర్రము (*gurramu*) and suffix -లు (*lu*). Some commonly used suffixes in plural forms of Telugu nouns are: లు (*lu*), ల్లు (*llu*), ల్లు (*dlu*). Here is a list of singular and plural forms of a few Telugu words:

Singular	Meaning	Plural
పిల్లి (*pilli*)	Cat	పిల్లులు (*pillulu*)
పడుచు (*paduchu*)	Women	పడుచులు (*paduchulu*)
ఆడది (*aadadi*)	Women	ఆడవాండ్లు (*aadawandlu** or *aadawalu*)
మగవాడు (*magavadu*)	Man	మగవాండ్లు (*magavandlu* or *magavallu*)
పురుషుడు (*purushudu*)	Man	పురుషులు (*purushulu*)
చెవి (*chevi*)	Ear	చెవులు (*chevulu*)
ఇల్లు (*illu*)	House	ఇల్లులు (*illulu* or *illlu*)

*Another plural form of *aadadi* is *adawaru*, which is used to show respect.

There are three main ways of word formation: inflection, derivation, and compounding. In inflection, a root word is combined with a grammatical morpheme to yield a word of the same class as the original stem. Derivation combines a word stem with a grammatical morpheme to yield a word belonging to a different class, e.g., formation of the noun 'computation' from the verb 'compute'. The formation of a noun from a verb or adjective is called nominalization. Compounding is the process of merging two or more words to form a new word. For example, personal computer, desktop, overlook. Morphological analysis and generation deal with inflection, derivation and compounding process in word formation.

New words are continually forming a natural language. Many of these are morphologically related to known words. Understanding morphology is therefore important to understand the syntactic and semantic properties of new words. Morphological analysis and generation are essential to many NLP applications ranging from spelling corrections to machine translations. In parsing, e.g., it helps to know the agreement features of words. In information retrieval, morphological analysis helps identify the presence of a query word in a document in spite of different morphological variants.

Parsing, in general, means taking an input and producing some sort of structures for it. In NLP, this structure might be morphological, syntactic, semantic, or pragmatic. Morphological parsing takes as input the inflected

surface form of each word in a text. As output, it produces the parsed form consisting of a canonical form (or *lemma*) of the word and a set of tags showing its syntactical category and morphological characteristics, e.g., possible part of speech and/or inflectional properties (gender, number, person, tense, etc.). Morphological generation is the inverse of this process. Both analysis and generation rely on two sources of information: a dictionary of the valid lemmas of the language and a set of inflection paradigms.

A morphological parser uses following information sources:

1. **Lexicon**
   A lexicon lists stems and affixes together with basic information about them.
2. **Morphotactics**
   There exists certain ordering among the morphemes that constitute a word. They cannot be arranged arbitrarily. For example, rest-less-ness is a valid word in English but not rest-ness-less. Morphotactics deals with the ordering of morphemes. It describes the way morphemes are arranged or touch each other.
3. **Orthographic rules**
   These are spelling rules that specify the changes that occur when two given morphemes combine. For example the $y \rightarrow ier$ spelling rule changes '*easy*' to '*easier*' and not to '*easyer*'.

Morphological analysis can be avoided if an exhaustive lexicon is available that lists features for all the word-forms of all the roots. Given a word, we simply consult the lexicon to get its feature values. For example, suppose an exhaustive lexicon for Hindi contains the following entries for the Hindi root-word *ghodhaa*.

Table 3.7 A sample lexicon entry

Word form	Category	Root	Gender	Number	Person
Ghodhaa	noun	GhoDaa	masculine	singular	3rd
Ghodhii	-do-	-do-	feminine	-do-	-do-
Ghodhon	-do-	-do-	masculine	plural	-do-
Ghodhe	-do-	-do-	-do-	-do-	-do-

Given a word, say *ghodhon*, we can look up the lexicon to get its feature values.

However, this approach has several limitations. First, it puts a heavy demand on memory. We have to list every form of the word, which results in a large number of, often redundant, entries in the lexicon.

Second, an exhaustive lexicon fails to show the relationship between different roots having similar word-forms. That means the approach fails to capture linguistic generalization, which is essential to develop a system capable of understanding unknown words.

Third, for morphologically complex languages, like Turkish, the number of possible word-forms may be theoretically infinite. It is not practical to list all possible word-forms in these languages.

These limitations explain why morphological parsing is necessary. The complexity of the morphological analysis varies widely among the world's languages, and is quite high even in relatively simple cases, such as English.

The simplest morphological systems are stemmers that collapse morphological variations of a given word (word-forms) to one lemma or stem. They do not require a lexicon. Stemmers have been especially used in information retrieval. Two widely used stemming algorithms have been developed by Lovins (1968) and Porter (1980). Stemmers do not use a lexicon; instead, they make use of rewrite rules of the form:

ier → *y* (e.g., earlier → early)

ing → ε (e.g., playing → play)

Stemming algorithms work in two steps:

(i) Suffix removal: This step removes predefined endings from words.

(ii) Recoding: This step adds predefined endings to the output of the first step.

These two steps can be performed sequentially as in Lovins's stemmer or simultaneously as in Porter's stemmer. For example, Porter's stemmer makes use of the following transformation rule:

ational → *ate*

to transform word such as 'rotational' into 'rotate'.

It is difficult to use stemming with morphologically rich languages. Even in English, stemmers are not perfect. Krovitz (1993) pointed out errors of omissions and commissions in the Porter algorithm, such as transformation of the word 'organization' into 'organ' and 'noise' into 'noisy'. Another problem with Porter's algorithm is that it reduces only suffixes; prefixes and compounds are not reduced.

A more efficient *two-level morphological model*, first proposed by Koskenniemi (1983), can be used for highly inflected languages. In this model, a word is represented as a correspondence between its lexical level form and its surface level form. The surface level represents the actual spelling of the word while the lexical level represents the concatenation of its constituent morphemes. Morphological parsing is viewed as a mapping from the surface level into morpheme and feature sequences on the lexical level.

For example, the surface form 'playing' is represented in the lexical form as play + V + PP as shown in Figure 3.5. The lexical form consists of the stem 'play' followed by the morphological information +V +PP, which tells us that 'playing' is the present participle form of the verb.

Surface level	p	l	a	y	i	n	g

Lexical level	p	l	a	y	+V	PP

Figure 3.5 Surface and lexical forms of a word

Similarly, the surface form 'books' is represented in the lexical form as 'book + N + PL', where the first component is the stem and the second component (N + PL) is the morphological information, which tells us that the surface level form is a plural noun.

This model is usually implemented with a kind of finite-state automata, called *finite-state transducer* (FST). A transducer maps a set of symbols to another. A finite state transducer does this through a finite state automaton. An FST can be thought of as a two-state automaton, which recognizes or generates a pair of strings. FST passes over the input string by consuming the input symbols on the tape it traverses and consists it to the output string in the form of symbols. Formally, an FST has been defined by Hopcroft and Ullman (1979) as follows:

A finite-state transducer is a 6-tuple $(\Sigma_1, \Sigma_2, Q, \delta, S, F)$, where θ is set of states, S is the initial state, and $F \subseteq Q$ is a set of final states, Σ_1 is *input* alphabet, Σ_2 is *output* alphabet, and δ is a function mapping $Q \times (\Sigma_1 \cup \{\varepsilon\}) \times (\Sigma_2 \cup \{\varepsilon\})$ to a subset of the power set of Q.

Transducers can be seen as automata with transitions labelled with symbols from $\Sigma_1 \times \Sigma_2$, where Σ_1 and Σ_2 are the alphabets of input and output respectively. Thus, an FST is similar to an NFA except in that transitions are made on strings rather than on symbols and, in addition, they have outputs.

Figure 3.6 shows a simple transducer that accepts two input strings, hot and cat, and maps them onto cot and bat respectively. It is a common practice to represent a pair like *a:a* by a single letter.

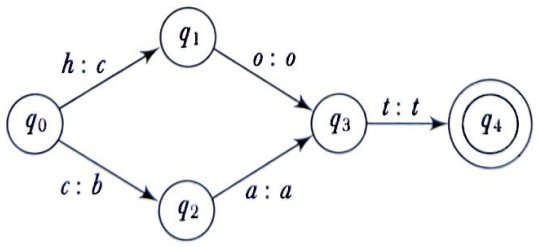

Figure 3.6 Finite-state transducer

Just as FSAs encode regular languages, FSTs encode regular relations. Regular relation is the relation between regular languages. The regular language encoded on the upper side of an FST is called upper language, and the one on the lower side is termed lower language. If T is a transducer, and s is a string, then we use $T(s)$ to represent the set of strings encoded by T such that the pair (s, t) is in the relation.

The FSTs are closed under union concatenation, composition, and Kleene closure. However, in general, they are not closed under intersection and complementation.

With this introduction, we can now implement the two-level morphology using FST. To get from the surface form of a word to its morphological analysis, we proceed in two steps as illustrated in Figure 3.7.

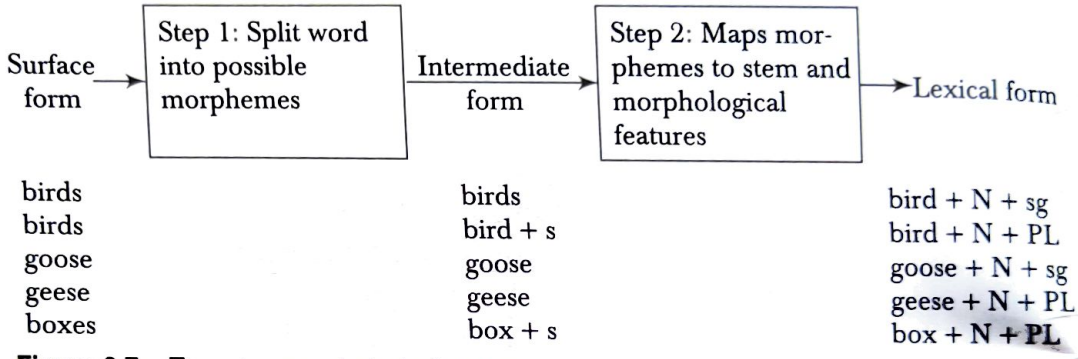

Figure 3.7 Two-step morphological parser

First, we split the words up into its possible components. For example, we make *bird* + *s* out of *birds*, where + indicates morpheme boundaries. In this step, we also consider spelling rules. Thus, there are two possible ways of splitting up *boxes*, namely *boxe* + *s* and *box* + *s*. The first one assumes that *boxe* is the stem and *s* the suffix, while the second assumes that *box* the stem is and that *e* has been introduced due to the spelling rule. The output of this step is a concatenation of morphemes, i.e., stems and affixes. There can be more than one representation for a given word. A transducer that does the mapping (translation) required by this step for the surface form 'lesser' might look like Figure 3.8. This FST represents the information that the comparative form of the adjective *less* is *lesser*, ε here is the empty string. The automaton is inherently bi-directional: the same transducer can be used for analysis (surface input, 'upward' application) or for generation (lexical input, 'downward' application).

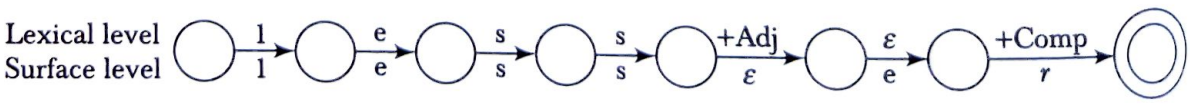

Figure 3.8 A simple FST

In the second step, we use a lexicon to look up categories of the stems and meaning of the affixes. So, *bird* + *s* will be mapped to *bird*+ *N*+ *PL*, and *box* + *s* to *box*+ *N* + *PL*. We also find out now that *boxe* is not a legal stem. This tells us that splitting *boxes* into *boxe* + *s* is an incorrect way of splitting *boxes*, and should therefore be discarded. This may not be the case always. We have words like *spouses* or *parses* where splitting the word into *spouse* + *s* or *parse* + *s* is correct. Orthographic rules are used to handle these spelling variations. For instance, one of the spelling rules says- add e after -s, -z, -x, -ch, -sh before the s (e.g., dish→dishes, box→boxes).

Each of these steps can be implemented with the help of a transducer. Thus, we need to build two transducers: one that maps the surface form to the intermediate form and another that maps the intermediate form to the lexical form.

We now develop an FST-based morphological parser for singular and plural nouns in English. The plural form of regular nouns usually end with -s or -es. However, a word ending in 's' need not necessarily be the plural form of a word. There are a number of singular words ending in s, e.g., miss and ass. One of the required translations is the deletion of the 'e' when introducing a morpheme boundary. This deletion is usually required for words ending in xes, ses, zes (e.g., suffixes and boxes). The transducer in Figure 3.9 does this. Figure 3.10 shows the possible sequences of states that the transducer undergoes, given the surface forms *birds* and *boxes* as input.

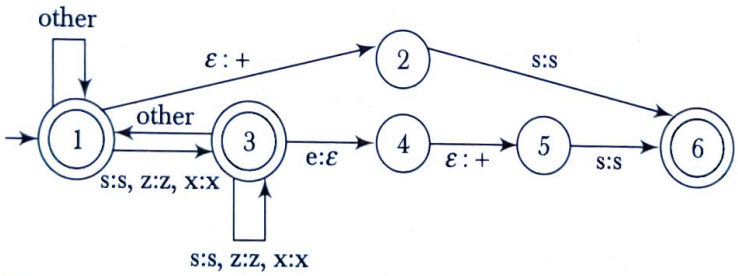

Figure 3.9 A simplified FST, mapping English nouns to the intermediate form

The next step is to develop a transducer that does the mapping from the intermediate level to the lexical level. The input to transducer has one of the following forms:

- Regular noun stem, e.g., *bird, cat*
- Regular noun stem + s, e.g., *bird* + *s*
- Singular irregular noun stem, e.g., *goose*
- Plural irregular noun stem, e.g., *geese*

In the first case, the transducer has to map all symbols of the stem to themselves and then output *N* and *sg* (Figure 3.7). In the second case, it

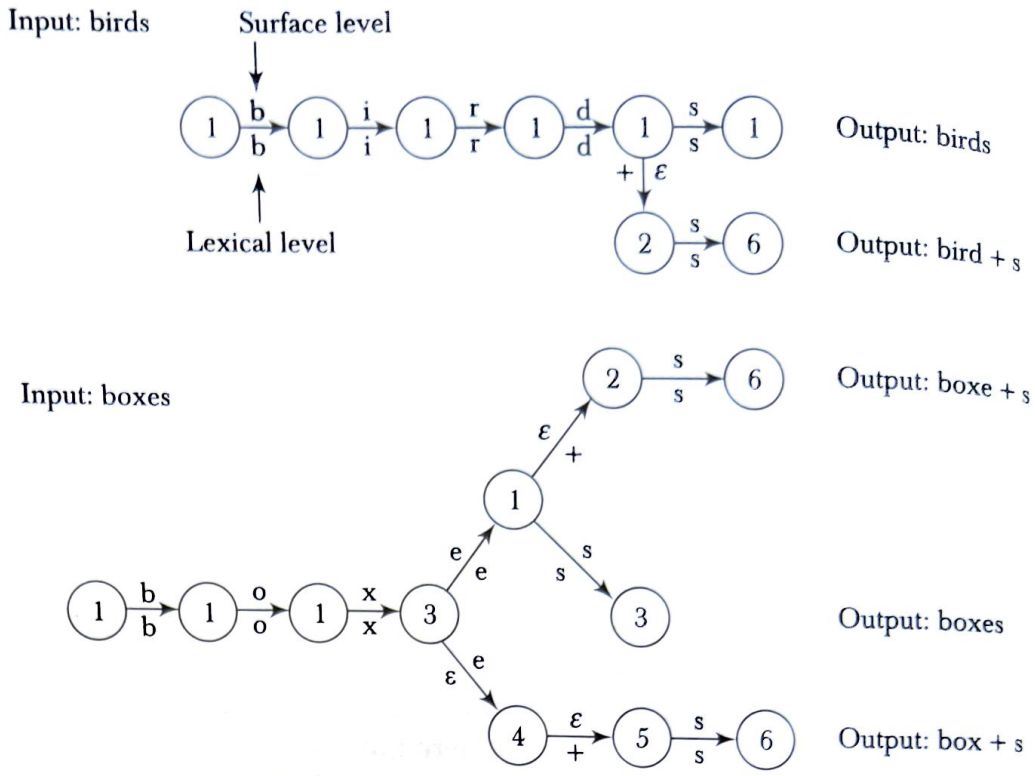

Figure 3.10 Possible sequences of states

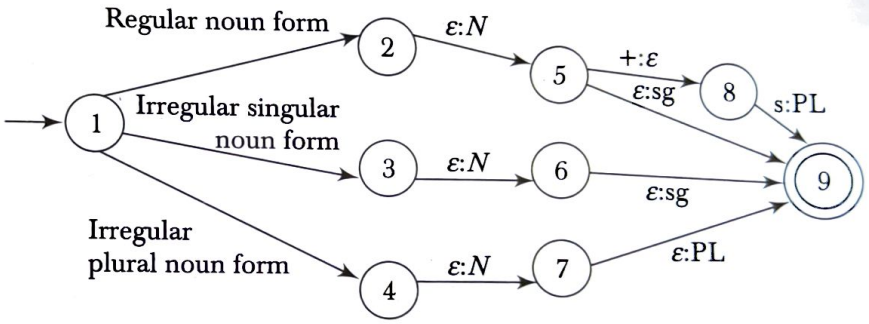

Figure 3.11 Transducer for Step 2

has to map all symbols of the stem to themselves, but then output *N* and replaces *PL* with *s*. In the third case, it has to do the same as in the first case. Finally, in the fourth case, the transducer has to map the irregular plural noun stem to the corresponding singular stem (e.g., *geese* to *goose*) and then it should add *N* and *PL*. The general structure of this transducer looks like Figure 3.11.

The mapping from State 1 to State 2, 3, or 4 is carried out with the help of a transducer encoding a lexicon. The transducer implementing the lexicon maps the individual regular and irregular noun stems to their correct noun stem, replacing labels like regular noun form, etc. This lexicon maps the surface form *geese*, which is an irregular noun, to its correct stem *goose* in the following way:

g:g e:o e:o s:s e:e

Mapping for the regular surface form of *bird* is b:b i:i r:r d:d. Representing pairs like *a:a* with a single letter, these two representations are reduced to g e:o e:o s e and b i r d respectively.

Composing this transducer with the previous one, we get a single two-level transducer with one input tape and one output tape. This maps plural nouns into the stem plus the morphological marker + pl and singular nouns into the stem plus the morpheme + sg. Thus a surface word form *birds* will be mapped to bird + N + pl as follows.

b:b i:i r:r d:d + ε:N + s:pl

Each letter maps to itself, while ε maps to morphological feature +N, and s maps to morphological feature pl. Figure 3.12 shows the resulting composed transducer.

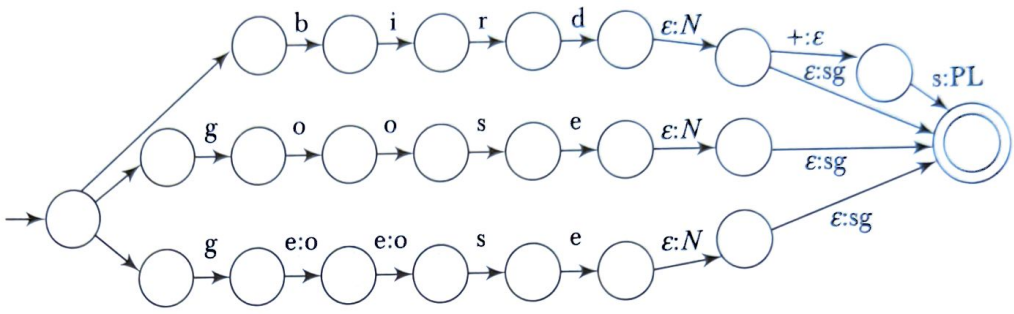

Figure 3.12 A transducer mapping nouns to their stem and morphological features

The power of the transducer lies in the fact that the same transducer can be used for analysis and generation. That is, we can run it in the downward direction (input: surface form and output: lexical form) or in the upward direction.

3.5 SPELLING ERROR DETECTION AND CORRECTION

In computer-based information systems, errors of typing and spelling constitute a very common source of variation between strings. These errors have been widely investigated. All investigations agree that single character omission, insertion, substitution, and reversal are the most common typing mistakes. In an early investigation, Damearu (1964) reported that over 80% of the typing errors were single-error misspellings:

(1) substitution of a single letter,
(2) omission of a single letter,
(3) insertion of a single letter, and
(4) transposition of two adjacent letters.

Shafer and Hardwick (1968) found that the most common type of single character error was substitution, followed by omission of a letter, and then insertion of a letter. Single character omission occurs when a single character is missed (deleted), e.g., when 'concept' is accidentally typed as 'concpt'. Insertion error refers to the presence of an extra character in a word, e.g. when 'error' is misspell as 'errorn'. Substitution error occurs when a wrong letter is typed in place of the right one, as in 'errpr', where 'p' appears in place of 'o'. Reversal refers to a situation in which the sequence of characters is reversed, e.g., 'aer' instead of 'are'. This is also termed transposition.

Optical character recognition (OCR) and other automatic reading devices introduce errors of substitution, deletion, and insertion but not of reversal. OCR errors are usually grouped into five classes: substitution, multi-substitution (or framing), space deletion or insertion, and failures. Unlike substitution errors made in typing, OCR substitution errors are caused due to visual similarity such as c→e, 1→l, r→n. The same is true for multi-substitution, e.g., m→rn. Failure occurs when the OCR algorithm fails to select a letter with sufficient accuracy. The frequency and type of errors are characteristics of the particular device. These errors can be corrected using 'context' or by using linguistic structures.

Many approaches to speech recognition deal with strings of phonemes (or symbols representing sounds), and attempt to match a spoken utterance with a dictionary of known utterances.

Unlike typing errors, spelling errors are mainly phonetic, where the misspell word is pronounced in the same way as the correct word. Phonetic errors are harder to set right because they distort the word by more than a single insertion, deletion, or substitution. Phonetic variations are common in transliteration. For example,

Spelling errors belong to one of two distinct categories: non-word errors and real word errors. When an error results in a word that does not appear in a given lexicon or is not a valid orthographic word form, it is termed a non-word error. Most of the early research on spelling errors focused on the detection of such non-words. The two main techniques used were *n*-gram analysis and dictionary lookup. Non-word error detection is now considered a solved problem.

A real-word error results in actual words of the language. It occurs due to typographical mistakes or spelling errors, e.g., substituting the spelling of a homophone or near-homophone, such as piece for peace or meat for meet. Real-word errors may cause local syntactic errors, global syntactic

errors, semantic errors, or errors at discourse or pragmatic levels. It becomes impossible to decide that a word is wrong without some contextual information.

Spelling correction consists of *detecting* and *correcting* errors. Error detection is the process of finding misspelled words and error correction is the process of suggesting correct words to a misspelled one. These sub-problems are addressed in two ways:

1. Isolated-error detection and correction
2. Context-dependent error detection and correction

In isolated-word error detection and correction, each word is checked separately, independent of its context. Detecting whether or not a word is correct seems simple–why not to look up the word in a lexicon? Unfortunately, it is not as simple as it appears. There are a number of problems associated with this simple strategy.

- The strategy requires the existence of a lexicon containing all correct words. Such a lexicon would take a long time to compile and occupy a lot of space.
- Some languages are highly productive. It is impossible to list all the correct words of such languages.
- This strategy fails when spelling error produces a word that belongs to the lexicon, e.g., when 'theses' is written in place of 'these'. Such an error is called a *real-word error.*
- The larger the lexicon, the more likely it is that an error goes undetected, because the chance of a word being found is greater in a large lexicon.

Context dependent error detection and correction methods, utilize the context of a word to detect and correct errors. This requires grammatical analysis and is thus more complex and language dependent. Even in context dependent methods, the list of candidate words must first be obtained using an isolated-word method before making a selection depending on the context.

The spelling correction algorithm has been broadly categorized by Kukich (1992) as follows.

Minimum edit distance The minimum edit distance between two strings is the minimum number of operations (insertions, deletions, or substitutions) required to transform one string into another. Spelling correction algorithms based on minimum edit distance are the most studied algorithms.

Similarity key techniques The basic idea in a similarity key technique is to change a given string into a key such that similar strings will change into the same key. The SOUNDEX system (Odell and Russell 1918) is an example of a system that uses this technique in phonetic spelling correction applications.

***n*-gram based techniques** The *n*-grams can be used for both non-word and real-word error detection because in the English alphabet, certain bi-grams and tri-grams of letters never occur or rarely do so; for example the tri-gram *qst* and the bi-gram *qd*. This information can be used to handle non-word error. Strings that contain these unusual *n*-grams can be identified as possible spelling errors. *n*-gram techniques usually require a large corpus or dictionary as training data, so that an *n*-gram table of possible combinations of letters can be compiled. In case of real-word error detection, we calculate the likelihood of one character following another and use this information to find possible correct word candidates.

Neural nets These have the ability to do associative recall based on incomplete and noisy data. They can be trained to adapt to specific spelling error patterns. The drawback of neural nets is that they are computationally expensive.

Rule-based techniques In a rule-based technique, a set of rules (heuristics) derived from knowledge of a common spelling error pattern is used to transform misspelled words into valid words. For example, if it is known that many errors occur from the letters *ue* being typed as *eu*, then we may write a rule that represents this.

3.5.1 Minimum Edit Distance

The minimum edit distance is the number of insertions, deletions, and substitutions required to change one string into another (Wagner and Fischer 1974). When we talk about distance between two strings, we are talking of the minimum edit distance. For example, the minimum edit distance between 'tutor' and 'tumor' is 2: We substitute 'm' for 't' and insert 'u' before 'r'. No smaller edit sequence can be found for this conversion. Therefore, the minimum edit distance is 2. Edit distance between two strings can be represented as a binary function, ed, which maps two strings to their edit distance. ed is symmetric. For any two strings, s and t, ed(s, t) is always equal to ed(t, s).

Edit distance can be viewed as a string alignment problem. By aligning two strings, we can measure the degree to which they match. There may be more than one possible alignment between two strings. The best

possible alignment corresponds to the minimum edit distance between the strings. The alignment shown here, between *tutor* and *tumour*, has a distance of 2.

t u **t** o – r

t u **m** o u r

A dash in the upper string indicates insertion. A substitution occurs when the two alignment symbols do not match (shown in bold). We can associate a weight or cost with each operation. The *Levensthein* distance between two sequences is obtained by assigning a unit cost to each operation. Another possible alignment for this sequences is:

t u **t** – o – r

t u – **m** o u r

which has a cost of 3. We already have a better alignment than this one.

The problem of finding minimum edit distance seems quite simple but in fact is not so. A choice that seems good initially might lead to problems later. Dynamic programming algorithms can be quite useful for finding minimum edit distance between two sequences. Dynamic programming refers to a class of algorithms that apply a table-driven approach to solve problems by combining solutions to sub-problems. The dynamic programming algorithm for minimum edit distance is implemented by creating an edit distance matrix. This matrix has one row for each symbol in the source string and one column for each matrix in the target string. The (i, j)th cell in this matrix represents the distance between the first i character of the source and the first j character of the target string. Each cell can be computed as a simple function of its surrounding cells. Thus, by starting at the beginning of the matrix, it is possible to fill each entry iteratively. The value in each cell is computed in terms of three possible paths.

$$dist[i, j] = \begin{cases} dist[i-1, j] + insert\_cost, \\ dist[i-1, j-1] + subst\_\cos t[source_i, t\arg et_j] \\ dist[i, j-1] + delete\_cost \end{cases}$$

The substitution will be 0 if the ith character in the source matches with jth character in the target. The minimum edit distance algorithm is shown in Figure 3.13. How the algorithm computes the minimum edit distance between *tutor* and *tumour* is shown in Figure 3.14.

```
Input: Two strings, X and Y
Output: The minimum edit distance between X and Y
m ← length(X)
n ← length(Y)
for i = 0 to m do
    dist[i,0] ← i
for j = 0 to n do
    dist[0,j] ← j
for i = 0 to m do
    for j = 0 to n do
        dist[i,j] = min{ dist[i-1,j] + insert_cost,
                         dist[i-1,j-1] + subst_cost(X_i, Y_j),
                         dist[i,j-1] + delet_cost }
```

Figure 3.13 Minimum edit distance algorithm

	#	t	u	m	o	u	r
#	0	1	2	3	4	5	6
t	1	0	1	2	3	4	5
u	2	1	0	1	2	3	4
t	3	2	1	1	2	3	4
o	4	3	2	2	1	2	3
r	5	4	3	3	2	2	2

Figure 3.14 Computing minimum edit distance

Minimum edit distance algorithms are also useful for determining accuracy in speech recognition systems. Kemal Oflazer (1996) proposed an efficient algorithm based on spelling correction with finite-state automata.

3.6 WORDS AND WORD CLASSES

Words are classified into categories called part-of-speech. These are sometimes called word classes or lexical categories. These lexical categories are usually defined by their syntactic and morphological behaviours. The most common lexical categories are nouns and verbs. Other lexical categories include adjectives, adverbs, prepositions, and conjunctions. Table 3.8 shows some of the word classes in English. Lexical categories and their properties vary from language to language. Word classes are further categorized as open and closed word classes. Open word classes constantly

acquire new members while closed word classes do not (or only infrequently do so). Nouns, verbs (except auxiliary verbs), adjectives, adverbs, and interjections are open word classes. Prepositions, auxiliary verbs, delimiters, conjunction, and particles are closed word classes.

Table 3.8 Part-of-speech example

NN	noun	student, chair, proof, mechanism
VB	verb	study, increase, produce
ADJ	adj	large, high, tall, few,
JJ	adverb	carefully, slowly, uniformly
IN	preposition	in, on, to, of
PRP	pronoun	I, me, they
DET	determiner	the, a, an, this, those

3.7 PART-OF-SPEECH TAGGING

Part-of-speech tagging is the process of assigning a part-of-speech (such as a noun, verb, pronoun, preposition, adverb, and adjective), to each word in a sentence. The input to a tagging algorithm is the sequence of words of a natural language sentence and specified tag sets (a finite list of part-of-speech tags). The output is a single best part-of-speech tag for each word. Many words may belong to more than one lexical category. For example, the English word 'book' can be a noun as in '*I am reading a good book*' or a verb as in '*The police booked the snatcher*'. The same is true for other languages. For example, the Hindi word '*sona*' may mean 'gold' (noun) or 'sleep' (verb). However, only one of the possible meanings is used at a time. In tagging, we try to determine the correct lexical category of a word in its context. No tagger is efficient enough to identify the correct lexical category of each word in a sentence in every case. The tag assigned by a tagger is the most likely for a particular use of word in a sentence.

The collection of tags used by a particular tagger is called a *tag set*. Most part-of-speech tag sets make use of the same basic categories, i.e., noun, verb, adjective, and prepositions. However, tag sets differ in how they define categories and how finely they divide words into categories. For example, both *eat* and *eats* might be tagged as a verb in one tag set, but assigned distinct tags in another tag set. In addition, most tag sets capture morpho-syntactic information such as singular/plural, number, gender, tense, etc. Consider the following sentences:

Zuha *eats* an apple daily.

Aman *ate* an apple yesterday.

They have *eaten* all the apples in the basket.

I like to *eat* guavas.

The word *eat* has a distinct grammatical form in each of these four sentences. *Eat* is the base form, *ate* its past tense, and the form *eats* requires a third person singular subject. Similarly, *eaten* is the past participle form and cannot occur in another grammatical context. It is required after have or has. Thus, the following sentences are ungrammatical:

I like to *eats* guava.

They *eaten* all the apples.

The number of tags used by different taggers varies substantially. Some use 20, while others use over 400 tags. The Penn Treebank tag set contains 45 tags while C7 uses 164. For a language like English, which is not morphologically rich, the C7 tagset is too big. The tagging process would yield too many mistagged words and the result would have to be manually corrected. Despite this, bigger tag sets have been used, e.g., TOSCA-ICE for the International Corpus of English with 270 tags (Garside 1997), or TESS with 200 tags. The larger the tag set, the greater the information captured about a linguistic context. However, the task of tagging becomes complicated and requires manual correction. A bigger tag set can be used for morphologically rich languages without introducing too many tagging errors. A tag set that uses just one tag to denote all the verbs will assign identical tags to all the forms of a verb. Although this coarse-grained distinction may be appropriate for some tasks, a fine-grained tag set captures more information. This is useful for tasks like syntactic pattern detection. The Penn Treebank tag set captures finer distinctions by assigning distinct tags to distinct grammatical forms of a verb, as summarized in Table 3.9. Tags assigned to the four different forms of the word *eat* according to this tag set is shown in Table 3.10.

Table 3.9 Tags from Penn Treebank tag set

VB	Verb, base form Subsumes imperatives, infinitives, and subjunctives
VBD	Verb, past tense Includes the conditional form of the verb *to be*
VBG	Verb, gerund, or present participle
VBN	Verb, past participle
VBP	Verb, non-3rd person singular present
VBZ	Verb, 3rd person singular present

Table 3.10 Possible tags for the word to *eat*

eat	VB
ate	VBD
eaten	VBN
eats	VBP

Here is an example of a tagged sentence:

Speech/NN sounds/NNS were/VBD sampled/VBN by/IN a/DT microphone/NN.

The tag set used is Penn Treebank.

Another tagging possible for this sentence is as follows:

Speech/NN sounds/VBZ were/VBD sampled/VBN by/IN a/DT microphone/NN.

It is easy to see that the second tagged sequence is not correct. It leads to semantic incoherence. We resolve the ambiguity using the context of the word. The context is also utilized by automatic taggers.

Part-of-speech tagging is an early stage of text processing in many NLP applications including speech synthesis, machine translation, information retrieval, and information extraction. In information retrieval, part-of-speech tagging can be used for indexing (for identifying useful tokens like nouns and phrases) and for disambiguating word senses. Tagging is not as complex as parsing. In tagging, a complete parse tree is not built; part-of-speech is assigned to words using contextual information.

Part-of-speech tagging methods fall under the three general categories.

- Rule-based (linguistic)
- Stochastic (data-driven)
- Hybrid

Rule-based taggers use hand-coded rules to assign tags to words. These rules use a lexicon to obtain a list of candidate tags and then use rules to discard incorrect tags.

Stochastic taggers have data-driven approaches in which frequency-based information is automatically derived from corpus and used to tag words. Stochastic taggers disambiguate words based on the probability that a word occurs with a particular tag. The simplest scheme is to assign the most frequent tag to each word. An early example of stochastic tagger was CLAWS (constituent likelihood automatic word-tagging system). CLAWS is the Stochastic equivalent of TAGGIT. Hidden Markov model (HMM) is tshe standard Stochastic tagger.

Hybrid taggers combine features of both these approaches. Like rule-based systems, they use rules to specify tags. Like stochastic systems, they use machine-learning to induce rules from a tagged training corpus automatically. The transformation-based tagger or Brill tagger is an example of the hybrid approach.

3.7.1 Rule-based Tagger

Most rule-based taggers have a two-stage architecture. The first stage is simply a dictionary look-up procedure, which returns a set of potential tags (parts-of-speech) and appropriate syntactic features for each word. The second stage uses a set of hand-coded rules to discard contextually illegitimate tags to get a single part-of-speech for each word. For example, consider the noun-verb ambiguity in the following sentence:

The show must go on.

The potential tags for the word *show* in this sentence is {VB, NN}. We resolve this ambiguity by using the following rule.

IF *preceding word is determiner* THEN *eliminate VB tag.*

This rule simply disallows verbs after a determiner. Using this rule the word *show* in the given sentence can only be noun.

In addition to contextual information, many taggers use morphological information to help in the disambiguation process. An example of a rule that make use of morphological information is:

IF *word ends in –ing and preceding word is a verb* THEN *label it a verb (VB).*

Capitalization information can be utilized in the tagging of unknown nouns. Rule-based taggers usually require supervised training. Instead, rules can be induced automatically. To induce rules untagged text is run through a tagger. The output is then manually corrected. The corrected text is then submitted to the tagger, which learns correction rules by comparing the two sets of data. This process may be repeated several times.

The earlier systems for automatic tagging were all rule-based. An example is TAGGIT (Greene and Rubin 1971), which was used for the initial tagging of the Brown corpus (Francis and Kucera 1982). This was also rule-based system. It used 3,300 disambiguation rules and was able to tag 77% of the words in the Brown corpus with their correct part-of-speech. The rest was done manually over several years. Yet another rule-based tagger is ENGTWOL (Voutilainen 1995).

Speed is an advantage of the rule-based tagger, and unlike stochastic taggers, they are deterministic. One of the arguments against them is the skill and effort required in writing disambiguation rules. However, stochastic taggers also require manual work if good performance is to be achieved. In the rule-based system, time is spent in writing a rule-set. For stochastic taggers, time is spent developing restrictions on transitions and emissions to improve tagger performance. Another disadvantage of the rule-based tagger is that it is usable for only one language. Using it for another one requires a rewrite of most of the program; unlike a probabilistic tagger which would be usable with only slight changes and new training.

3.7.2 Stochastic Tagger

The standard stochastic tagger algorithm is the HMM tagger. A Markov model applies the simplifying assumption that the probability of a chain of symbols can be approximated in terms of its parts or *n*-grams. The simplest *n*-gram model is the unigram model, which assigns the most likely tag (part-of-speech) to each token.

The unigram model needs to be trained using a tagged training corpus before it can be used to tag data. The most likely statistics are gathered over the corpus and used for tagging. The context used by the unigram tagger is the text of the word itself. For example, it will assign the tag JJ for each occurrence of *fast*, since *fast* is used as an adjective more frequently than it is used as a noun, verb, or adverb. This results in incorrect tagging in each of the following sentences:

She had a *fast*. (3.1)

Muslims *fast* during Ramadan. (3.2)

Those who were injured in the accident need to be helped *fast*. (3.3)

In the first sentence, *fast* is used as a noun. In the second, it is a verb, and in the third, an adverb.

We would expect more accurate predictions if we took more context into account when making a tagging decision. A bi-gram tagger uses the current word and the tag of the previous word in the tagging process. As the tag sequence "DT NN" is more likely than the tag sequence "DT JJ", a bi-gram model will assign a correct tag to the word *fast* in sentence (3.1). Similarly, it is more likely that an adverb (rather than a noun or an adjective) follows a verb. Hence, in sentence (3.3), the tag assigned to *fast* will be RB.

In general, an *n*-gram model considers the current word and the tag of the previous *n*–1 words in assigning a tag to a word. The context considered by a tri-gram model is shown in Figure 3.15. The area shaded in grey represents the context.

Tokens	w_{n-2}	w_{n-1}	w_n	w_{n+1}
tags	t_{n-2}	t_{n-1}	t_n	t_{n+1}

Figure 3.15 Context used by a tri-gram tagger

So far, we have considered how a tag is assigned to a word given the previous tag(s). However, the objective of a tagger is to assign a tag sequence to a given sentence. We now discuss how the HMM tagger assigns the most likely tag sequence to a given sentence. We call this the HMM because it uses two layers of states: a visible layer corresponding to the input words, and a hidden layer learnt by the system corresponding to the tags. While tagging the input data, we only observe the words—the tags (states) are hidden. States of the model are visible in training, not during the tagging task.

As discussed earlier, the HMM makes use of lexical and bi-gram probabilities estimated over a tagged training corpus in order to compute the most likely tag sequence for each sentence. One way to store the statistical information is to build a probability matrix. The probability matrix contains both the probability that an individual word belongs to a word class as well as the *n*-gram analysis, e.g., for a bi-gram model, the probability that a word of class X follows a word of class Y. This matrix is then used to drive the HMM tagger while tagging an unknown text.

We now return to the original problem. Given a sequence of words (sentence), the objective is to find the most probable tag sequence for the sentence.

Let W be the sequence of words.

$$W = w_1, w_2, \ldots, w_n$$

The task is to find the tag sequence

$$T = t_1, t_2, \ldots, t_n$$

which maximizes $P(T|W)$, i.e.,

$$T' = \text{argmax}_T\ P(T|W)$$

Applying Bayes Rule, $P(T|W)$ can be estimated using the expression:

$$P(T|W) = P(W|T) * P(T)/P(W)$$

As the probability of the word sequence, $P(W)$, remains the same for each tag sequence, we can drop it. The expression for the most likely tag sequence becomes:

$$T' = \text{argmax}_T \, P(W|T) * P(T)$$

Using the Markov assumption, the probability of a tag sequence can be estimated as the product of the probability of its constituent n-grams, i.e.,

$$P(T) = P(t_1) * P(t_2|t_1) * P(t_3|t_1 t_2) \ldots * P(t_n|t_1 \ldots t_{n-1})$$

$P(W/T)$ is the probability of seeing a word sequence, given a tag sequence. For example, it is asking the probability of seeing 'The egg is rotten' given 'DT NNP VB JJ'. We make the following two assumptions:

- The words are independent of each other.
- The probability of a word is dependent only on its tag.

Using these assumptions, we obtain

$$P(W/T) = P(w_1/t_1) * P(w_2/t_2) \ldots . P(w_i/t_i) * \ldots P(W_n/t_n)$$

i.e., $$P(W/T) \approx \prod_{i=1}^{n} P(w_i|t_i)$$

So, $$P(W/T) * P(T) = \prod_{i=1}^{n} P(w_i|t_i) \times P(t_1) \times P(t_2|t_1) * P(t_3|t_1 t_2) \times \ldots \times P(t_n|t_1 \ldots t_{n-1})$$

Approximating the tag history using only the two previous tags, the transition probability, $P(T)$, becomes

$$P(T) = P(t_1) \times P(t_2|t_1) \times P(t_3|t_1 t_2) \times \ldots \times P(t_n|t_{n-2}\ t_{n-1})$$

Hence, $P(T/W)$ can be estimated as

$$\begin{aligned} P(W/T) * P(T) &= \prod_{i=1}^{n} P(w_i|t_i) \\ &\quad \times P(t_1) \times P(t_2|t_1) * P(t_3|t_2 t_1) \times \ldots \times P(t_n|t_{n-2}\ t_{n-1}) \\ &= \prod_{i=1}^{n} P(w_i|t_i) \times P(t_1) \times P(t_2/t_1) \times \prod_{i=3}^{n} P(t_i|t_{i-2}\, t_{i-1}) \end{aligned}$$

We estimate these probabilities from relative frequencies via Maximum Likelihood Estimation.

$$P(t_i|t_{i-2}\ t_{i-1}) = \frac{c(t_{i-2}, t_{i-1}, t_i)}{c(t_{i-2}, t_{i-1})}$$

$$P(w_i|t_i) = \frac{c(w_i, t_i)}{c(t_i)}$$

where $c(t_{i-2}, t_{i-1}, t_i)$ is the number of occurrences of t_i followed by $t_{i-2}\ t_{i-1}$.

Stochastic models have the advantage of being accurate and language independent. Most stochastic taggers have an accuracy of 96–97%. The accuracy seems to be quite high but it should be noted that this is measured as a percentage of words. An accuracy of 96% means that for a sentence containing 20 words, the error rate per sentence will be $1-0.96^{20} = 56\%$. This corresponds to approximately one word per sentence.

One of the drawbacks of stochastic taggers is that they require a manually tagged corpus for training. Kupiec (1992), Cutting *et al.* (1992), and others have demonstrated that the HMM tagger can be trained from unannotated text. This makes it possible to use the model for languages in which a manually tagged corpus is not available. However, a tagger trained on a hand-coded corpus performs better than one trained on an unannotated text. In order to achieve good performance a tagged corpus is required.

We now consider an example demonstrating how the probability of a particular part-of-speech sequence for a given sentence can be computed.

Example 3.3 Consider the sentence

The bird can fly.

and the tag sequence

DT NNP MD VB

Using bi-gram approximation, the probability

$$P\begin{pmatrix} \text{DT} & \text{NNP} & \text{MD} & \text{VB} \\ | & | & | & | \\ \text{The} & \text{bird} & \text{can} & \text{fly} \end{pmatrix}$$

can be computed as

$$= P(\text{DT}) \times P(\text{NNP}|\text{DT}) * P(\text{MD}|\text{NNP}) \times P(\text{VB}|\text{MD}) \times P(\text{the/DT}) \times P(\text{bird}|\text{NNP}) \times P(\text{can}|\text{MD}) \times P(\text{fly}|\text{VB})$$

3.7.3 Hybrid Taggers

Hybrid approaches to tagging combine the features of both the rule-based and stochastic approaches. They use rules to assign tags to words. Like the stochastic taggers, this is a machine learning technique and rules are automatically induced from the data. Transformation-based learning (TBL) of tags, also known as Brill tagging, is an example of hybrid approach. TBL is a machine learning method introduced by E. Brill (in 1995). Transformation-based error-driven learning has been applied to a number of natural language problems, including part-of-speech tagging, speech generation, and syntactic parsing (Brill 1993, 1994, Huang et al. 1994).

Figure 3.16 illustrates the TBL process. Like most HMM taggers, TBL is also a supervised learning technique. The steps involved in the TBL tagging algorithm are shown in Table 3.11. The input to Brill's TBL tagging algorithm is a tagged corpus and a lexicon (with most frequent information as indicated in the training corpus). The initial state annotator uses the lexicon to assign the most likely tag to each word as the start state. An ordered set of transformation rules are applied sequentially. The rule that results in the most improved tagging is selected. A manually tagged corpus is used as reference for truth. The process is iterated until some stopping criterion is reached, such as when no significant information is achieved over the previous iteration. At each iteration, the transformation that results in the highest score is selected. The output of the algorithm is a ranked list of learned transformation that transform the initial tagging close to the correct tagging. New text can then be annotated by first assigning the most frequent tag and then applying the ranked list of learned transformations in order.

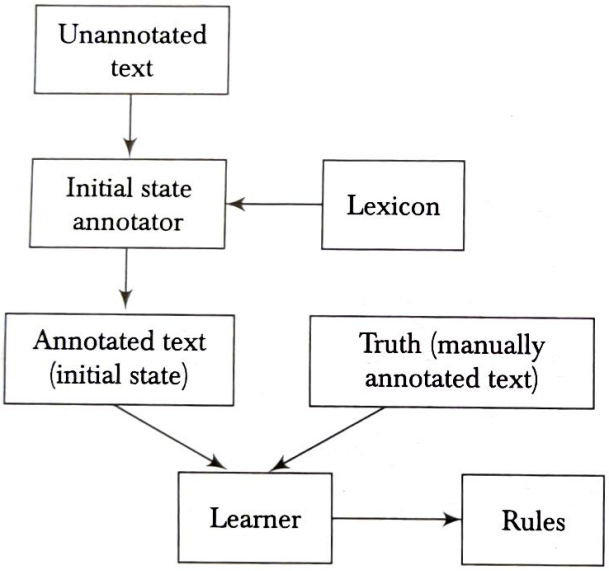

Figure 3.16 TBL learner

Table 3.11 TBL tagging algorithm

INPUT:	Tagged corpus and lexicon (with most frequent information)
Step 1:	Label every word with most likely tag (from dictionary)
Step 2:	Check every possible transformation and select one which most improves tagging
Step 3:	Re-tag corpus applying the rules
Repeat 2–3	Until some stopping criterion is reached
RESULT:	Ranked sequence of transformation rules

Each transformation is a pair of a re-write rule of the form $t_1 \rightarrow t_2$ and a contextual condition. In order to limit the set of transformations, a small set of templates is constructed. Any allowable transformation is an instantiation of these templates. Some of the transformation templates and transformations learned by TBL tagging are listed in Table 3.12.

Table 3.12 Examples of transformation templates and rules learned by the tagger

Change tag a to tag b when:				
1. The preceding (following) word is tagged z.				
2. The preceding (following) word is w.				
3. The word two before (after) is w.				
4. One of the preceding two words is w.				
5. One of the two preceding (following) words is tagged z.				
6. The current word is w and the preceding (following) word is x.				
7. One of the previous three words is tagged z.				
#	**Change tags from**	**to**	**Contextual condition**	**Example**
1.	NN	VB	The previous tag is TO.	To/TO fish/NN
2.	JJ	RB	The previous tag is VBZ.	runs/VBZ fast/JJ

We now explain how the rules are applied in TBL tagger with the help of an example.

Example 3.4 Assume that in a corpus, *fish* is most likely to be a noun.

$$P(\text{NN/fish}) = 0.91$$

$$P(\text{VB/fish}) = 0.09$$

Now consider the following two sentences and their initial tags.

I/PRP like/VB to/TO eat/VB fish/NNP.

I/PRP like/VB to/TO fish/NNP.

As the most likely tag for *fish* is NNP, the tagger assigns this tag to the word in both sentences. In the second case, it is a mistake.

After initial tagging when the transformation rules are applied, the tagger learns a rule that applies exactly to this mis-tagging of *fish*:

Change NNP to VB if the previous tag is TO.

As the contextual condition is satisfied, this rule will change fish/NN to fish/VB:

like/VB to/TO fish/NN → like/VB to/TO fish/VB

The algorithm can be made more efficient by indexing the words in a training corpus using potential transformation. Recent works have involved the use of finite state transducers to compile pattern-action rules, combining

them to yield a single transducer representing the simultaneous application of all rules. Roche and Schabes (1997) have applied this approach to Brill's tagger. The resulting tagger is larger than Brill's original tagger and significantly faster.

Most of the work in part-of-speech tagging is done for English and some European languages. In other languages, part-of-speech tagging, and NLP research in general, is constrained by the lack of annotated corpuses. This is true for Indian language as well. A few part-of-speech tagging systems reported in recent years use morphological analysers along with a tagged corpus, e.g. a Bengali tagger based on HMM developd by Sandipan et al. (2004) and a Hindi tagger developed by Smriti et al. (2006). Smriti et al. used a decision tree based learning algorithm. A number of other part-of-speech taggers for Hindi, Bengali, and Telugu were developed as a result of the NLPAI-2006 machine learning contest on part-of-speech and chunking for Indian languages.

Tagging Urdu is more difficult. A number of factors contribute to this complexity. Among these is the right to left directionality of the written script and the presence of grammatical forms borrowed from Arabic and Persian. Little had been done to develop an extensive tag set for Urdu before Hardie (2003). His work was a part of the EMILLE—Enabling Minority Language Engineering project (see http://www.emille.lancs.ac.uk/about.php)—which focuses on the development of corpus and tools for South Asian languages.

3.7.4 Unknown Words

Unknown words are words that do not appear in dictionary or a training corpus. They create a problem during tagging. There are several potential solutions to this problem. One is to assign the most frequent tag (which occurs with most word types in the training corpus) to the unknown word. Another solution is to assume that the unknown words can be of any part-of-speech and initialize them by assigning them open class tags. Then proceed to disambiguate them using the probabilities of those tags. We can also use morphological information, such as affixes, to guess the possible tag of an unknown word. In this approach, the unknown word is assigned a tag based on the probability of the words belonging to a specific part-of-speech in the training corpus having the same suffix or prefix. A similar approach is used in Brill's tagger.

SUMMARY

This chapter has dealt with word level analysis. The topics covered include methods for characterizing word sequences, identifying morphological variants, detecting and correcting misspelled words, and identifying the correct part-of-speech for a word. The main points are as follows.

- Regular expressions can be used for specifying words. They can be encoded as a finite automaton.
- The goal of morphological parsing is to find out morphemes using which a given word is built. Morphemes are the smallest meaning-bearing units in a language.
- Morphological analysis and generation are essential to many NLP applications ranging from spelling error corrections to machine translations.
- The simplest morphological systems are stemmers. They do not use a lexicon. Instead, they use re-write rules. However, stemmers are not perfect.
- A two-level morphological model is more efficient. Both of its steps can be implemented using a finite state transducer.
- Word errors belong to one of two distinct categories, namely, *non-word errors* and *real word errors*. The latter category requires the context of the word to detect and correct errors.
- Words are classified into categories called part-of-speech or word classes. Word classes can be open or closed.
- Part-of-speech tagging is the process of assigning a part-of-speech like noun, verb, pronoun, preposition, adverb, adjective, etc., to each word in a sentence.
- Part-of-speech tagging methods fall under the following three categories:
  1. Rule-based (linguistic)
  2. Stochastic (data-driven)
  3. Hybrid

  Rule-based taggers use hand-coded rules to assign tags to words. *Stochastic taggers* require a pre-tagged corpus for training. *Hybrid taggers* combine features of both these approaches. Like rule-based systems, they use rules to specify tags. Like stochastic methods, they use machine learning to automatically induce rules from a tagged training corpus.
- Unknown words can be assigned the most frequent tags. We can also use morphological information to guess the correct tag.

REFERENCES

Brill, E., 1993, 'Transformation-based error-driven parsing,' *Proceedings of the Third International Workshop on Parsing Technologies,* Tilburg, The Netherlands.

Brill, E., 1994, 'Some advances in rule-based part of speech tagging,' *Proceedings of the Twelfth National Conference on Artificial Intelligence (AAAI-94),* Seattle.

Brill, E., 1995, 'Error-driven learning and natural language processing: a case study in part-of-speech tagging,' *Computational Linguistics.*

Cutting, D., J. Kupiec, J. Pederson, and P. Sibun, 1992, 'A practical part-of-speech tagger,' *Proceedings of the Third Conference on Applied Natural Language Processing, ACL.*

Damerau, F. J., 1964, 'A technique for computer detection and correction of spelling errors,' *Communications of the ACM,* 7(3) pp. 171–76.

Francis, W. Nelson and Henry Kucera, 1982, *Frequency Analysis of English Usage: Lexicon and Grammar,* Houghton Mifflin, Boston.

Garside, R., G. Leech, and G. Sampson, 1987, *The Computational Analysis of English: A Corpus-Based Approach,* Longman, London.

Green, B. and G. Rubin, 1971, 'Automated grammatical tagging of English,' Department of Linguistics, Brown University.

Hardie, A., 2003, 'Developing a tag-set for automated part-of-speech tagging in Urdu,' *Proceedings of the Corpus Linguistics 2003 Conference,* D. Archer, P. Rayson, A. Wilson, and T. McEnery, (Eds.), UCREL Technical Papers, 16, Department of Linguistics, Lancaster University.

Hopcroft, J.E. and J.D. Ullman, 1979, *Introduction to Automata Theory, Languages, and Computation,* Addison-Wesley, Reading, Massachusetts.

Huang et al., 1994, 'Generation of pronunciation from orthographies using transformation-based error-driven learning,' *Proceedings of Int. Conference on Speech and Language Processing (ICSLP),* Yokohama, Japan.

Kleene, S.C., 1956, 'Representation of events in nerve nets and finite automata,' *Automata Studies,* C. Shannon and J. McCarthy (Eds.), Princeton University Press, Princeton, NJ, pp. 3–41.

Koskenniemi, K., 1983, 'Two-level morphology: A general computational model of word-form recognition and production, Technical Report *Publication No. 11,* Department of General Linguistics,' University of Helsinki.

Krovetz, R., 1993, 'Viewing morphology as an inference process,' In *SIGIR-93,* pp. 191–202.

Kukich, K., 1992, 'Techniques for automatically correcting words in text,' *ACM Computing Surveys*, 24, pp. 377–439.

Kupiec, J., 1992, 'Robust part-of-speech tagging using a Hidden Markov Model,' *Computer Speech and Language*, vol. 6, pp. 225–42.

Lovins, J. B., 1968, 'Development of a stemming algorithm,' *Mechanical Translation and Computational Linguistics*, 11(1–2), pp. 22–31.

Odell, M. K. and R. C. Russell, US Patents 1261167/1435663 (1918/1922).

Oflazer, Kemal, 1996, 'Error-tolerant finite state recognition with applications to morphological analysis and spelling correction,' *Computational Linguistics*, 22(1).

Porter, M. F., 1980, 'An algorithm for suffix stripping program,' 14(3), pp. 130–37.

Roche, E. and Y. Schabes, 1995, 'Deterministic part of speech tagging with finite state transducers,' *Computational Linguistics*.

Sandipan, D., Kumar Nagraj, and Uma Sawant, 2004, 'A hybrid model for part-of-speech tagging and its application to Bengali,' *Proceedings of International Conference on Computational Intelligence*.

Shaffer, L. and J. Hardwick, 1968, 'Typing performance as a function of text,' *Quarterly Journal of Experimental Psychology*, 20, pp. 360–69.

Singh, Smriti, Kuhoo Gupta, Manish Shrivastava, and Pushpak Bhattacharyya, 2006, 'Morphological richness offsets resource demand–experiences in constructing a POS tagger for Hindi,' *Proceedings of the COLING/ACL 2006 Main Conference Poster Sessions*, Association for Computational Linguistics, Sydney, pp. 779–86.

Voutilainen, A., 1996, 'Morphological disambiguation,' *Constraint Grammar: A language-independent system for parsing uncertainty text*, F. Karlsson, A. Voutilainen, J. Heikkila, and A. Anttila (Eds.), Berlin, pp. 165–284.

Wagner, R.A. and M.J. Fischer, 1974, 'The string-to-string correction problem,' *Journal of the Association for Computing Machinery*, 21, 168–73.

EXERCISES

1. Define a finite automaton that accepts the following language: (aa)*(bb)*.
2. A typical URL is of the form:

http	://	www.abc.com	/nlppaper/public	/xxx.html
1	2	3	4	5

In this table, 1 is a protocol, 2 is name of a server, 3 is the directory, and 4 is the name of a document. Suppose you have to write a program that takes a URL and returns the protocol used, the DNS name of the server, the directory and the document name. Develop a regular expression that will help you in writing this program.

3. Distinguish between non-word and real-word error.
4. Compute the minimum edit distance between *paecflu* and *peaceful.*
5. Comment on the validity of the following statements:
   (a) Rule-based taggers are non-deterministic.
   (b) Stochastic taggers are language independent.
   (c) Brill's tagger is a rule-based tagger.
6. How can unknown words be handled in the tagging process?

LAB EXERCISES

1. Write a program to find minimum edit distance between two input strings.
2. Use any tagger available in your lab to tag a text file. Now write a program to find the most likely tag in the tagged text.
3. Write a program to find the probability of a tag given previous two tags, i.e., P(t3/t2 t1).