Speech and Language Processing

An Introduction to Natural Language Processing, Computational Linguistics, and Speech Recognition

Third Edition draft

Daniel Jurafsky
Stanford University

James H. Martin
University of Colorado at Boulder

Draft of February 3, 2024. Comments and typos welcome!

Summary of Contents

Contents

Part I

FUNDAMENTAL ALGORITHMS FOR NLP

In the first part of the book we introduce the fundamental suite of algorithmic tools that make up the modern neural language model that is the heart of end-to-end NLP systems. We begin with tokenization and preprocessing, as well as useful algo rithms like computing edit distance, and then proceed to the tasks of classification, logistic regression, neural networks, proceeding through feedforward networks, re current networks, and then transformers. We'll also see the role of embeddings as a model of word meaning.

CHAPTER

1 Introduction

La derniere chose qu'on trouve en faisant un ouvrage est de savoir celle qu'il faut ` mettre la premiere. `

[The last thing you figure out in writing a book is what to put first.]

Pascal

CHAPTER

2 Regular Expressions, Text Normalization, Edit Distance

User: I am unhappy.
ELIZA: DO YOU THINK COMING HERE WILL HELP YOU NOT TO BE UNHAPPY
User: I need some help, that much seems certain.
ELIZA: WHAT WOULD IT MEAN TO YOU IF YOU GOT SOME HELP

User: Perhaps I could learn to get along with my mother.
ELIZA: TELL ME MORE ABOUT YOUR FAMILY
User: My mother takes care of me.
ELIZA: WHO ELSE IN YOU FAMILY TAKES CARE OF YOU
User: My father.
ELIZA: YOUR FATHER
User: You are like my father in some ways.

Weizenbaum (1966)

ELIZA The dialogue above is from ELIZA, an early natural language processing system that could carry on a limited conversation with a user by imitating the responses of a Rogerian psychotherapist (Weizenbaum, 1966). ELIZA is a surprisingly simple program that uses pattern matching to recognize phrases like "I need X" and translate them into suitable outputs like "What would it mean to you if you got X?". This simple technique succeeds in this domain because ELIZA doesn't actually need to *know* anything to mimic a Rogerian psychotherapist. As Weizenbaum notes, this is one of the few dialogue genres where listeners can act as if they know nothing of the world. ELIZA's mimicry of human conversation was remarkably successful: many people who interacted with ELIZA came to believe that it really *understood* them and their problems, many continued to believe in ELIZA's abilities even after the program's operation was explained to them (Weizenbaum, 1976), and even today chatbots such chatbots are a fun diversion.

Of course modern conversational agents are much more than a diversion; they can answer questions, book flights, or find restaurants, functions for which they rely on a much more sophisticated understanding of the user's intent, as we will see in Chapter 15. Nonetheless, the simple pattern-based methods that powered ELIZA and other chatbots play a crucial role in natural language processing.

We'll begin with the most important tool for describing text patterns: the regular expression. Regular expressions can be used to specify strings we might want to extract from a document, from transforming "I need X" in ELIZA above, to defining strings like *$199* or *$24.99* for extracting tables of prices from a document.

We'll then turn to a set of tasks collectively called text normalization, in which text normalization regular expressions play an important part. Normalizing text means converting it to a more convenient, standard form. For example, most of what we are going to do with language relies on first separating out or tokenizing words from running tokenization text, the task of tokenization. English words are often separated from each other by whitespace, but whitespace is not always sufficient. *New York* and *rock 'n' roll* are sometimes treated as large words despite the fact that they contain spaces, while sometimes we'll need to separate *I'm* into the two words *I* and *am*. For processing tweets or texts we'll need to tokenize emoticons like :) or hashtags like #nlproc.

Some languages, like Japanese, don't have spaces between words, so word tokeniza tion becomes more difficult.

lemmatization Another part of text normalization is lemmatization, the task of determining that two words have the same root, despite their surface differences. For example, the words *sang*, *sung*, and *sings* are forms of the verb *sing*. The word *sing* is the common *lemma* of these words, and a lemmatizer maps from all of these to *sing*. Lemmatization is essential for processing morphologically complex languages like

stemming Arabic. Stemming refers to a simpler version of lemmatization in which we mainly just strip suffixes from the end of the word. Text normalization also includes sen tence segmentation: breaking up a text into individual sentences, using cues like sentence segmentation periods or exclamation points.

Finally, we'll need to compare words and other strings. We'll introduce a

metric called edit distance that measures how similar two strings are based on the number of edits (insertions, deletions, substitutions) it takes to change one string into the other. Edit distance is an algorithm with applications throughout language process ing, from spelling correction to speech recognition to coreference resolution.

2.1 Regular Expressions

One of the unsung successes in standardization in computer science has been the regular expression (often shortened to regex), a language for specifying text search regular expression strings. This practical language is used in every computer language, word processor, and text processing tools like the Unix tools grep or Emacs. Formally, a regular ex pression is an algebraic notation for characterizing a set of strings. Regular expres sions are particularly useful for searching in texts, when we have a pattern to search for and a corpus of texts to search through. A regular expression search function will search through the corpus, returning all texts that match the pattern. The corpus can be a single document or a collection. For example, the Unix command-line tool grep takes a regular expression and returns every line of the input document that matches the expression.

A search can be designed to return every match on a line, if there are more than one, or just the first match. In the following examples we generally underline the exact part of the pattern that matches the regular expression and show only the first match. We'll show regular expressions delimited by slashes but note that slashes are *not* part of the regular expressions.

Regular expressions come in many variants. We'll be describing extended regu lar expressions; different regular expression parsers may only recognize subsets of these, or treat some expressions slightly differently. Using an online regular expres sion tester is a handy way to test out your expressions and explore these variations.

2.1.1 Basic Regular Expression Patterns

The simplest kind of regular expression is a sequence of simple characters; putting concatenation characters in sequence is called concatenation. To search for *woodchuck*, we type /woodchuck/. The expression /Buttercup/ matches any string containing the substring *Buttercup*; grep with that expression would return the line *I'm called lit tle Buttercup*. The search string can consist of a single character (like /!/) or a sequence of characters (like /urgl/) (see Fig. 2.1).

Regular expressions are case sensitive; lower case /s/ is distinct from upper case /S/ (/s/ matches a lower case *s* but not an upper case *S*). This means that the pattern /woodchucks/ will not match the string *Woodchucks*. We can solve this problem with the use of the square braces [and]. The string of characters inside the braces specifies a disjunction of characters to match. For example, Fig. 2.2 shows that the pattern /[wW]/ matches patterns containing either *w* or *W*.

Regex	Example Patterns Matched
/woodchucks/	"interesting links to woodchucks and lemurs"
/a/	"Mary Ann stopped by Mona's"
/!/	"You've left the burglar behind again!" said Nori

Figure 2.1 Some simple regex searches.

Regex Match Example Patterns
/[wW]oodchuck/ Woodchuck or woodchuck “Woodchuck”
/[abc]/ ‘a’, ‘b’, *or* ‘c’ “In uomini, in soldati” /[1234567890]/ any digit “plenty of 7 to 5”

Figure 2.2 The use of the brackets [] to specify a disjunction of characters.

The regular expression /[1234567890]/ specifies any single digit. While such classes of characters as digits or letters are important building blocks in expressions, they can get awkward (e.g., it’s inconvenient to specify

/[ABCDEFGHIJKLMNOPQRSTUVWXYZ]/

to mean “any capital letter”). In cases where there is a well-defined sequence asso ciated with a set of characters, the brackets can be used with the dash (-) to specify range any one character in a range. The pattern /[2-5]/ specifies any one of the charac ters *2*, *3*, *4*, or *5*. The pattern /[b-g]/ specifies one of the characters *b*, *c*, *d*, *e*, *f*, or *g*. Some other examples are shown in Fig. 2.3.

Regex Match Example Patterns Matched
/[A-Z]/ an upper case letter “we should call it ‘Drenched Blossoms’ ”
/[a-z]/ a lower case letter “my beans were impatient to be hoed!”
/[0-9]/ a single digit “Chapter 1: Down the Rabbit Hole”

Figure 2.3 The use of the brackets [] plus the dash - to specify a range.

The square braces can also be used to specify what a single character *cannot* be, by use of the caret ˆ. If the caret ˆ is the first symbol after the open square brace [, the resulting pattern is negated. For example, the pattern /[ˆa]/ matches any single character (including special characters) except *a*. This is only true when the caret is the first symbol after the open square brace. If it occurs anywhere else, it usually stands for a caret; Fig. 2.4 shows some examples.

Regex Match (single characters) Example Patterns Matched /[ˆA-Z]/ not an upper case letter “Oyfn pripetchik” /[ˆSs]/ neither ‘S’ nor ‘s’ “I have no exquisite reason for’t” /[ˆ.]/ not a period “our resident Djinn” /[eˆ]/ either ‘e’ or ‘ˆ’ “look up ˆ now” /aˆb/ the pattern ‘aˆb’ “look up aˆ b now”

Figure 2.4 The caret ˆ for negation or just to mean ˆ. See below re: the backslash for escaping the period.

How can we talk about optional elements, like an optional *s* in *woodchuck* and *woodchucks*? We can’t use the square brackets, because while they allow us to say

“s or S”, they don’t allow us to say “s or nothing”. For this we use the question mark /?/, which means “the preceding character or nothing”, as shown in Fig. 2.5.

Regex Match Example Patterns Matched /woodchucks?/ woodchuck or woodchucks “woodchuck”
/colou?r/ color or colour “color”

Figure 2.5 The question mark ? marks optionality of the previous expression.

We can think of the question mark as meaning “zero or one instances of the previous character”. That is, it’s a way of specifying how many of something that we want, something that is very important in regular expressions. For example, consider the language of certain sheep, which consists of strings that look like the following:

baa!

baaa!
baaaa!
baaaaa!
. . .

This language consists of strings with a *b*, followed by at least two *a*'s, followed by an exclamation point. The set of operators that allows us to say things like "some Kleene * number of *a*s" are based on the asterisk or *, commonly called the Kleene * (gen erally pronounced "cleany star"). The Kleene star means "zero or more occurrences of the immediately previous character or regular expression". So /a*/ means "any string of zero or more *a*s". This will match *a* or *aaaaaa*, but it will also match the empty string at the start of *Off Minor* since the string *Off Minor* starts with zero *a*'s. So the regular expression for matching one or more *a* is /aa*/, meaning one *a* fol lowed by zero or more *a*s. More complex patterns can also be repeated. So /[ab]*/ means "zero or more *a*'s or *b*'s" (not "zero or more right square braces"). This will match strings like *aaaa* or *ababab* or *bbbb*.

For specifying multiple digits (useful for finding prices) we can extend /[0-9]/, the regular expression for a single digit. An integer (a string of digits) is thus /[0-9][0-9]*/. (Why isn't it just /[0-9]*/?)

Sometimes it's annoying to have to write the regular expression for digits twice, so there is a shorter way to specify "at least one" of some character. This is the Kleene + Kleene +, which means "one or more occurrences of the immediately preceding character or regular expression". Thus, the expression /[0-9]+/ is the normal way to specify "a sequence of digits". There are thus two ways to specify the sheep language: /baaa*!/ or /baa+!/.

One very important special character is the period (/./), a wildcard expression that matches any single character (*except* a carriage return), as shown in Fig. 2.6.

Regex	Match	Example Matches
/beg.n/	any character between *beg* and *n*	begin, beg'n, begun

Figure 2.6 The use of the period . to specify any character.

The wildcard is often used together with the Kleene star to mean "any string of characters". For example, suppose we want to find any line in which a particular word, for example, *aardvark*, appears twice. We can specify this with the regular expression /aardvark.*aardvark/.

anchors Anchors are special characters that anchor regular expressions to particular places

in a string. The most common anchors are the caret ^ and the dollar sign $. The caret ^ matches the start of a line. The pattern /^The/ matches the word *The* only at the start of a line. Thus, the caret ^ has three uses: to match the start of a line, to in dicate a negation inside of square brackets, and just to mean a caret. (What are the contexts that allow grep or Python to know which function a given caret is supposed to have?) The dollar sign $ matches the end of a line. So the pattern $ is a useful pattern for matching a space at the end of a line, and /^The dog\.$/ matches a line that contains only the phrase *The dog*. (We have to use the backslash here since we want the . to mean "period" and not the wildcard.)

Regex	Match
^	start of line
$	end of line
\b	word boundary
\B	non-word boundary

Figure 2.7 Anchors in regular expressions.

There are also two other anchors: \b matches a word boundary, and \B matches a non-boundary. Thus, /\bthe\b/ matches the word *the* but not the word *other*. More technically, a "word" for the purposes of a regular expression is defined as any sequence of digits, underscores, or letters; this is based on the definition of "words" in programming languages. For example, /\b99\b/ will match the string *99* in *There are 99 bottles of beer on the wall* (because 99 follows a space) but not *99* in *There are 299 bottles of beer on the wall* (since 99 follows a number). But it will match *99* in *$99* (since *99* follows a dollar sign ($), which is not a digit, underscore, or letter).

2.1.2 Disjunction, Grouping, and Precedence

Suppose we need to search for texts about pets; perhaps we are particularly interested in cats and dogs. In such a case, we might want to search for either the string *cat* or the string *dog*. Since we can't use the square brackets to search for "cat or dog" (why

disjunction can't we say /[catdog]/?), we need a new operator, the disjunction operator, also called the pipe symbol |. The pattern /cat|dog/ matches either the string cat or the string dog.

Sometimes we need to use this disjunction operator in the midst of a larger se quence. For example, suppose I want to search for information about pet fish for my cousin David. How can I specify both *guppy* and *guppies*? We cannot simply say /guppy|ies/, because that would match only the strings *guppy* and *ies*. This

precedence is because sequences like guppy take precedence over the disjunction operator |. To make the disjunction operator apply only to a specific pattern, we need to use the parenthesis operators (and). Enclosing a pattern in parentheses makes it act like a single character for the purposes of neighboring operators like the pipe | and the Kleene*. So the pattern /gupp(y|ies)/ would specify that we meant the disjunc tion only to apply to the suffixes y and ies.

The parenthesis operator (is also useful when we are using counters like the Kleene*. Unlike the | operator, the Kleene* operator applies by default only to a single character, not to a whole sequence. Suppose we want to match repeated instances of a string. Perhaps we have a line that has column labels of the form *Column 1 Column 2 Column 3*. The expression /Column [0-9]+ */ will not match any number of columns; instead, it will match a single column followed by

any number of spaces! The star here applies only to the space that precedes it, not to the whole sequence. With the parentheses, we could write the expression /(Column [0-9]+ *)*/ to match the word *Column*, followed by a number and optional spaces, the whole pattern repeated zero or more times.

This idea that one operator may take precedence over another, requiring us to sometimes use parentheses to specify what we mean, is formalized by the operator precedence hierarchy for regular expressions. The following table gives the order

operator

precedence of RE operator precedence, from highest precedence to lowest precedence.

Parenthesis ()
Counters * + ? {}
Sequences and anchors the ^my end$
Disjunction |

Thus, because counters have a higher precedence than sequences,

/the*/ matches *theeeee* but not *thethe*. Because sequences have a higher prece dence than disjunction, /the|any/ matches *the* or *any* but not *thany* or *theny*.

Patterns can be ambiguous in another way. Consider the expression /[a-z]*/ when matching against the text *once upon a time*. Since /[a-z]*/ matches zero or more letters, this expression could match nothing, or just the first letter *o*, *on*, *onc*, or *once*. In these cases regular expressions always match the *largest* string they can;

greedy we say that patterns are greedy, expanding to cover as much of a string as they can. non-greedy There are, however, ways to enforce non-greedy matching, using another mean- *? ing of the ? qualifier. The operator *? is a Kleene star that matches as little text as +? possible. The operator +? is a Kleene plus that matches as little text as possible.

2.1.3 A Simple Example

Suppose we wanted to write a RE to find cases of the English article *the*. A simple (but incorrect) pattern might be:

/the/

One problem is that this pattern will miss the word when it begins a sentence and hence is capitalized (i.e., *The*). This might lead us to the following pattern:

/[tT]he/

But we will still incorrectly return texts with the embedded in other words (e.g., *other* or *theology*). So we need to specify that we want instances with a word bound ary on both sides:

/\b[tT]he\b/

Suppose we wanted to do this without the use of /\b/. We might want this since /\b/ won't treat underscores and numbers as word boundaries; but we might want to find *the* in some context where it might also have underlines or numbers nearby (*the* or *the25*). We need to specify that we want instances in which there are no alphabetic letters on either side of the *the*:

/[^a-zA-Z][tT]he[^a-zA-Z]/

But there is still one more problem with this pattern: it won't find the word *the* when it begins a line. This is because the regular expression [^a-zA-Z], which

we used to avoid embedded instances of *the*, implies that there must be some single (although non-alphabetic) character before the *the*. We can avoid this by specify ing that before the *the* we require *either* the beginning-of-line or a non-alphabetic character, and the same at the end of the line:

/(^|[^a-zA-Z])[tT]he([^a-zA-Z]|$)/

The process we just went through was based on fixing two kinds of errors: false false positives positives, strings that we incorrectly matched like *other* or *there*, and false nega false negatives tives, strings that we incorrectly missed, like *The*. Addressing these two kinds of errors comes up again and again in implementing speech and language processing systems. Reducing the overall error rate for an application thus involves two antag onistic efforts:

- Increasing precision (minimizing false positives)
- Increasing recall (minimizing false negatives)

We'll come back to precision and recall with more precise definitions in Chapter 4.

2.1.4 More Operators

Figure 2.8 shows some aliases for common ranges, which can be used mainly to save typing. Besides the Kleene * and Kleene + we can also use explicit numbers as counters, by enclosing them in curly brackets. The regular expression /{3}/ means "exactly 3 occurrences of the previous character or expression". So /a\.{24}z/ will match *a* followed by 24 dots followed by *z* (but not *a* followed by 23 or 25 dots followed by a *z*).

Regex	Expansion Match	First Matches
\d	[0-9]	any digit Party of 5
\D	[^0-9]	any non-digit Blue moon
\w	[a-zA-Z0-9_]	any alphanumeric/underscore Daiyu
\W	[^\w]	a non-alphanumeric !!!!
\s	[\r\t\n\f]	whitespace (space, tab) in Concord
\S	[^\s]	Non-whitespace in Concord

Figure 2.8 Aliases for common sets of characters.

A range of numbers can also be specified. So /{n,m}/ specifies from *n* to *m* occurrences of the previous char or expression, and /{n,}/ means at least *n* occur rences of the previous expression. REs for counting are summarized in Fig. 2.9.

Regex	Match
*	zero or more occurrences of the previous char or expression
+	one or more occurrences of the previous char or expression
?	zero or one occurrence of the previous char or expression
{n}	exactly *n* occurrences of the previous char or expression
{n,m}	from *n* to *m* occurrences of the previous char or expression
{n,}	at least *n* occurrences of the previous char or expression
{,m}	up to *m* occurrences of the previous char or expression

Figure 2.9 Regular expression operators for counting.

Finally, certain special characters are referred to by special notation based on the newline backslash (\) (see Fig. 2.10). The most common of these are the newline character

\n and the tab character \t. To refer to characters that are special themselves (like ., *, [, and \), precede them with a backslash, (i.e., /\./, /*/, /\[/, and /\\/).

Regex	Match	First Patterns Matched
*	an asterisk "*"	"K*A*P*L*A*N"
\.	a period "."	"Dr. Livingston, I presume"
\?	a question mark	"Why don't they come and lend a hand?"
\n	a newline	
\t	a tab	

Figure 2.10 Some characters that need to be backslashed.

2.1.5 A More Complex Example

Let's try out a more significant example of the power of REs. Suppose we want to build an application to help a user buy a computer on the Web. The

user might want “any machine with at least 6 GHz and 500 GB of disk space for less than $1000”. To do this kind of retrieval, we first need to be able to look for expressions like *6 GHz* or *500 GB* or *Mac* or *$999.99*. In the rest of this section we’ll work out some simple regular expressions for this task.

First, let’s complete our regular expression for prices. Here’s a regular expres sion for a dollar sign followed by a string of digits:

/$[0-9]+/

Note that the $ character has a different function here than the end-of-line function we discussed earlier. Most regular expression parsers are smart enough to realize that $ here doesn’t mean end-of-line. (As a thought experiment, think about how regex parsers might figure out the function of $ from the context.)

Now we just need to deal with fractions of dollars. We’ll add a decimal point and two digits afterwards:

/$[0-9]+\.[0-9][0-9]/

This pattern only allows *$199.99* but not *$199*. We need to make the cents optional and to make sure we’re at a word boundary:

/(^|\W)$[0-9]+(\.[0-9][0-9])?\b/

One last catch! This pattern allows prices like *$199999.99* which would be far too expensive! We need to limit the dollars:

/(^|\W)$[0-9]{0,3}(\.[0-9][0-9])?\b/

Further fixes (like avoiding matching a dollar sign with no price after it) are left as an exercise for the reader.

How about disk space? We’ll need to allow for optional fractions again (*5.5 GB*); note the use of ? for making the final s optional, and the use of / **/ to mean “zero or more spaces” since there might always be extra spaces lying around:

/\b[0-9]+(\.[0-9]+)? *(GB|[Gg]igabytes?)\b/

Modifying this regular expression so that it only matches more than 500 GB is left as an exercise for the reader.

2.1.6 Substitution, Capture Groups, and ELIZA

substitution An important use of regular expressions is in substitutions. For example, the substi tution operator s/regexp1/pattern/ used in Python and in Unix commands like vim or sed allows a string characterized by a regular expression to be replaced by another string:

s/colour/color/

It is often useful to be able to refer to a particular subpart of the string matching the first pattern. For example, suppose we wanted to put angle brackets around all integers in a text, for example, changing *the 35 boxes* to *the <35> boxes*. We’d like a way to refer to the integer we’ve found so that we can easily add the brackets. To do this, we put parentheses (and) around the first pattern and use the number operator \1 in the second pattern to refer back. Here’s how it looks:

s/([0-9]+)/<\1>/

The parenthesis and number operators can also specify that a certain string or expression must occur twice in the text. For example, suppose we are looking for the pattern "the Xer they were, the Xer they will be", where we want to constrain the two X's to be the same string. We do this by surrounding the first X with the parenthesis operator, and replacing the second X with the number operator \1, as follows:

/the (.*)er they were, the \1er they will be/

Here the \1 will be replaced by whatever string matched the first item in paren theses. So this will match *the bigger they were, the bigger they will be* but not *the bigger they were, the faster they will be*.

capture group This use of parentheses to store a pattern in memory is called a capture group. Every time a capture group is used (i.e., parentheses surround a pattern), the re register sulting match is stored in a numbered register. If you match two different sets of parentheses, \2 means whatever matched the *second* capture group. Thus

/the (.*)er they (.*), the \1er we \2/

will match *the faster they ran, the faster we ran* but not *the faster they ran, the faster we ate*. Similarly, the third capture group is stored in \3, the fourth is \4, and so on. Parentheses thus have a double function in regular expressions; they are used to group terms for specifying the order in which operators should apply, and they are used to capture something in a register. Occasionally we might want to use parentheses for grouping, but don't want to capture the resulting pattern in a register. In that case we use a non-capturing group, which is specified by putting the special non-capturing group commands ?: after the open parenthesis, in the form (?: pattern).

/(?:some|a few) (people|cats) like some \1/

will match *some cats like some cats* but not *some cats like some some*.

Substitutions and capture groups are very useful in implementing simple chat bots like ELIZA (Weizenbaum, 1966). Recall that ELIZA simulates a Rogerian psychologist by carrying on conversations like the following:

$User_1$: Men are all alike.
$ELIZA_1$: IN WHAT WAY
$User_2$: They're always bugging us about something or other.
$ELIZA_2$: CAN YOU THINK OF A SPECIFIC EXAMPLE
$User_3$: Well, my boyfriend made me come here.
$ELIZA_3$: YOUR BOYFRIEND MADE YOU COME HERE
$User_4$: He says I'm depressed much of the time.
$ELIZA_4$: I AM SORRY TO HEAR YOU ARE DEPRESSED

ELIZA works by having a series or cascade of regular expression substitutions each of which matches and changes some part of the input lines. After the input is uppercased, substitutions change all instances of *MY* to *YOUR*, and *I'M* to *YOU ARE*, and so on. That way when ELIZA repeats back part of the user utterance, it will seem to be referring correctly to the user. The next set of substitutions matches and replaces other patterns in the input. Here are some examples:

```
s/.* YOU ARE (depressed|sad) .*/I AM SORRY TO HEAR YOU ARE \1/
s/.* YOU ARE (depressed|sad) .*/WHY DO YOU THINK YOU ARE \1/
s/.* all .*/IN WHAT WAY/
```

s/.* always .*/CAN YOU THINK OF A SPECIFIC EXAMPLE/

Since multiple substitutions can apply to a given input, substitutions are assigned a rank and applied in order. Creating patterns is the topic of Exercise 2.3, and we return to the details of the ELIZA architecture in Chapter 15.

2.1.7 Lookahead Assertions

Finally, there will be times when we need to predict the future: look ahead in the text to see if some pattern matches, but not yet advance the pointer we always keep to where we are in the text, so that we can then deal with the pattern if it occurs, but if it doesn't we can check for something else instead.

lookahead These lookahead assertions make use of the (? syntax that we saw in the previ ous section for non-capture groups. The operator (?= pattern) is true if pattern zero-width occurs, but is zero-width, i.e. the match pointer doesn't advance. The operator (?! pattern) only returns true if a pattern does not match, but again is zero-width and doesn't advance the pointer. Negative lookahead is commonly used when we are parsing some complex pattern but want to rule out a special case. For example suppose we want to match, at the beginning of a line, any single word that doesn't start with "Volcano". We can use negative lookahead to do this:

/ˆ(?!Volcano)[A-Za-z]+/

2.2 Words

Before we talk about processing words, we need to decide what counts as a word. corpus Let's start by looking at one particular corpus (plural corpora), a computer-readable corpora collection of text or speech. For example the Brown corpus is a million-word col lection of samples from 500 written English texts from different genres (newspa per, fiction, non-fiction, academic, etc.), assembled at Brown University in 1963–64 (Kucera and Francis ˇ , 1967). How many words are in the following Brown sentence?

He stepped out into the hall, was delighted to encounter a water brother.

This sentence has 13 words if we don't count punctuation marks as words, 15 if we count punctuation. Whether we treat period ("."), comma (","), and so on as words depends on the task. Punctuation is critical for finding boundaries of things (commas, periods, colons) and for identifying some aspects of meaning (question marks, exclamation marks, quotation marks). For some tasks, like part-of-speech tagging or parsing or speech synthesis, we sometimes treat punctuation marks as if they were separate words.

The Switchboard corpus of American English telephone conversations between strangers was collected in the early 1990s; it contains 2430 conversations averaging 6 minutes each, totaling 240 hours of speech and about 3 million words (Godfrey et al., 1992). Such corpora of spoken language introduce other complications with regard to defining words. Let's look at one utterance from Switchboard; an utter utterance ance is the spoken correlate of a sentence:

I do uh main- mainly business data processing

disfluency This utterance has two kinds of disfluencies. The broken-off word *main-* is fragment called a fragment. Words like *uh* and *um* are called fillers or filled pauses. Should filled pause we consider these to be words? Again, it depends on the application.

If we are building a speech transcription system, we might want to eventually strip out the disfluencies.

But we also sometimes keep disfluencies around. Disfluencies like *uh* or *um* are actually helpful in speech recognition in predicting the upcoming word, because they may signal that the speaker is restarting the clause or idea, and so for speech recognition they are treated as regular words. Because people use different disflu encies they can also be a cue to speaker identification. In fact Clark and Fox Tree (2002) showed that *uh* and *um* have different meanings. What do you think they are?

Perhaps most important, in thinking about what is a word, we need to distinguish word type two ways of talking about words that will be useful throughout the book. Word types are the number of distinct words in a corpus; if the set of words in the vocabulary word instance is V, the number of types is the vocabulary size $|V|$. Word instances are the total number N of running words.[1]

If we ignore punctuation, the following Brown sentence has 16 instances and 14 types:

They picnicked by the pool, then lay back on the grass and looked at the stars.

We still have decisions to make! For example, should we consider a capitalized string (like *They*) and one that is uncapitalized (like *they*) to be the same word type? The answer is that it depends on the task! *They* and *they* might be lumped together as the same type in some tasks, like speech recognition, where we might just care about getting the words in order and don't care about the formatting, while for other tasks, such as deciding whether a particular word is a noun or verb (part-of-speech tagging) or whether a word is a name of a person or location (named-entity tag ging), capitalization is a useful feature and is retained. Sometimes we keep around two versions of a particular NLP model, one with capitalization and one without capitalization.

How many words are there in English? When we speak about the number of words in the language, we are generally referring to word types. Fig. 2.11 shows the rough numbers of types and instances computed from some English corpora.

[1] In earlier tradition, and occasionally still, you might see word instances referred to as word *tokens*, but we now try to reserve the word *token* instead to mean the output of word tokenization algorithms.

Corpus	Instances = N	Types = $\|V\|$
Shakespeare	884 thousand	31 thousand
Brown corpus	1 million	38 thousand
Switchboard telephone conversations	2.4 million	20 thousand
COCA	440 million	2 million
Google n-grams	1 trillion	13 million

Figure 2.11 Rough numbers of wordform types and instances for some English language corpora. The largest, the Google n-grams corpus, contains 13 million types, but this count only includes types appearing 40 or more times, so the true number would be much larger.

The larger the corpora we look at, the more word types we find, and in fact this relationship between the number of types $|V|$ and number of instances N is called Herdan's Law Herdan's Law (Herdan, 1960) or Heaps' Law (Heaps, 1978) after its discoverers Heaps' Law (in linguistics and information retrieval respectively). It is shown in Eq. 2.1, where k and β are positive constants, and $0 < \beta < 1$.

$$|V| = kN^{\beta} \tag{2.1}$$

The value of β depends on the corpus size and the genre, but at least for the large corpora in Fig. 2.11, β ranges from .67 to .75. Roughly then we can say that the vocabulary size for a text goes up significantly faster than the square root of its length in words.

It's sometimes useful to make a further distinction. Consider inflected forms like *cats* versus *cat*. We say these two words are different wordforms but have the lemma same lemma. A lemma is a set of lexical forms having the same stem, the same wordform major part-of-speech, and the same word sense. The wordform is the full inflected or derived form of the word. The two wordforms *cat* and *cats* thus have the same lemma, which we can represent as *cat*.

For morphologically complex languages like Arabic, we often need to deal with lemmatization. For most tasks in English, however, wordforms are sufficient, and when we talk about words in this book we almost always mean wordsforms (al though we will discuss basic algorithms for lemmatization and the related task of stemming below in Section 2.6). One of the situations even in English where we talk about lemmas is when we measure the number of words in a dictionary. Dictio nary entries or boldface forms are a very rough approximation to (an upper bound on) the number of lemmas (since some lemmas have multiple boldface forms). The 1989 edition of the Oxford English Dictionary had 615,000 entries.

Finally, we should note that in practice, for many NLP applications (for example for neural language modeling) we don't actually use words as our internal unit of representation at all! We instead tokenize the input strings into tokens, which can be words but can also be only parts of words. We'll return to this tokenization question when we introduce the BPE algorithm in Section 2.5.2.

2.3 Corpora

Words don't appear out of nowhere. Any particular piece of text that we study is produced by one or more specific speakers or writers, in a specific dialect of a specific language, at a specific time, in a specific place, for a specific function.

Perhaps the most important dimension of variation is the language. NLP algo rithms are most useful when they apply across many languages. The world has 7097

languages at the time of this writing, according to the online Ethnologue catalog (Simons and Fennig, 2018). It is important to test algorithms on more than one lan guage, and particularly on languages with different properties; by contrast there is an unfortunate current tendency for NLP algorithms to be developed or tested just on English (Bender, 2019). Even when algorithms are developed beyond English, they tend to be developed for the official languages of large industrialized nations (Chinese, Spanish, Japanese, German etc.), but we don't want to limit tools to just these few languages. Furthermore, most languages also have multiple varieties, of ten spoken in different regions or by different social groups. Thus, for example,

AAE if we're processing text that uses features of African American English (AAE) or African American Vernacular English (AAVE)—the variations of English used by millions of people in African American communities (King 2020)—we must use NLP tools that function with features of those varieties. Twitter posts might use fea tures often used by speakers of African American English, such as constructions like

MAE *iont* (*I don't* in Mainstream American English (MAE)), or *talmbout* corresponding to MAE *talking about*, both examples that influence word segmentation (Blodgett et al. 2016, Jones 2015).

It's also quite common for speakers or writers to use multiple languages in a code switching single communicative act, a phenomenon called code switching. Code

switching is enormously common across the world; here are examples showing Spanish and (transliterated) Hindi code switching with English (Solorio et al. 2014, Jurgens et al. 2017):

(2.2) Por primera vez veo a @username actually being hateful! it was beautiful:) *[For the first time I get to see @username actually being hateful! it was beautiful:)]*

(2.3) dost tha or ra- hega ... dont wory ... but dherya rakhe
["he was and will remain a friend ... don't worry ... but have faith"]

Another dimension of variation is the genre. The text that our algorithms must process might come from newswire, fiction or non-fiction books, scientific articles, Wikipedia, or religious texts. It might come from spoken genres like telephone conversations, business meetings, police body-worn cameras, medical interviews, or transcripts of television shows or movies. It might come from work situations like doctors' notes, legal text, or parliamentary or congressional proceedings.

Text also reflects the demographic characteristics of the writer (or speaker): their age, gender, race, socioeconomic class can all influence the linguistic properties of the text we are processing.

And finally, time matters too. Language changes over time, and for some lan guages we have good corpora of texts from different historical periods.

Because language is so situated, when developing computational models for lan guage processing from a corpus, it's important to consider who produced the lan guage, in what context, for what purpose. How can a user of a dataset know all these

datasheet details? The best way is for the corpus creator to build a datasheet (Gebru et al., 2020) or data statement (Bender et al., 2021) for each corpus. A datasheet specifies properties of a dataset like:

Motivation: Why was the corpus collected, by whom, and who funded it?

Situation: When and in what situation was the text written/spoken? For example, was there a task? Was the language originally spoken conversation, edited text, social media communication, monologue vs. dialogue?

Language variety: What language (including dialect/region) was the corpus in?

Speaker demographics: What was, e.g., the age or gender of the text's authors? Collection process: How big is the data? If it is a subsample how was it sampled? Was the data collected with consent? How was the data pre-processed, and what metadata is available?

Annotation process: What are the annotations, what are the demographics of the annotators, how were they trained, how was the data annotated?

Distribution: Are there copyright or other intellectual property restrictions?

2.4 Simple Unix Tools for Word Tokenization

Before almost any natural language processing of a text, the text has to be normal ized, a task called text normalization. At least three tasks are commonly applied as [text normalization] part of any normalization process:

1. Tokenizing (segmenting) words
2. Normalizing word formats
3. Segmenting sentences

In the next sections we walk through each of these tasks, but we'll first start with an easy, if somewhat naive version of word tokenization and normalization (and fre quency computation) that can be accomplished for English solely in a single UNIX command-line, inspired by Church (1994). We'll make use of some Unix com mands: tr, used to systematically change particular characters in the input; sort, which sorts input lines in alphabetical order; and uniq, which collapses and counts adjacent identical lines.

For example let's begin with the 'complete words' of Shakespeare in one file, sh.txt. We can use tr to tokenize the words by changing every sequence of non alphabetic characters to a newline ('A-Za-z' means alphabetic and the -c option complements to non-alphabet, so together they mean to change every non-alphabetic character into a newline. The -s ('squeeze') option is used to replace the result of multiple consecutive changes into a single output, so a series of non-alphabetic characters in a row would all be 'squeezed' into a single newline):

```
tr -sc 'A-Za-z' '\n' < sh.txt
```

The output of this command will be:

```
THE
SONNETS
by
William
Shakespeare
From
fairest
creatures
We
...
```

Now that there is one word per line, we can sort the lines, and pass them to uniq -c which will collapse and count them:

```
tr -sc 'A-Za-z' '\n' < sh.txt | sort | uniq -c
```


with the following output:

```
1945 A
72 AARON
19 ABBESS
25 Aaron
6 Abate
1 Abates
5 Abbess
6 Abbey
3 Abbot
...
```

Alternatively, we can collapse all the upper case to lower case:

```
tr -sc 'A-Za-z' '\n' < sh.txt | tr A-Z a-z | sort | uniq -c
```

whose output is

14725 a
97 aaron
1 abaissiez
10 abandon
2 abandoned
2 abase
1 abash
14 abate
3 abated
3 abatement
...

Now we can sort again to find the frequent words. The -n option to sort means to sort numerically rather than alphabetically, and the -r option means to sort in reverse order (highest-to-lowest):

```
tr -sc 'A-Za-z' '\n' < sh.txt | tr A-Z a-z | sort | uniq -c | sort -n -r
```

The results show that the most frequent words in Shakespeare, as in any other corpus, are the short function words like articles, pronouns, prepositions:

27378 the
26084 and
22538 i
19771 to
17481 of
14725 a
13826 you
12489 my
11318 that
11112 in
...

Unix tools of this sort can be very handy in building quick word count statistics for any corpus in English. While in some versions of Unix these command-line tools also correctly handle Unicode characters and so can be used for many languages, in general for handling most languages outside English we use more sophisticated tokenization algorithms.

2.5 Word Tokenization

tokenization

The simple UNIX tools above were fine for getting rough word statistics but more sophisticated algorithms are generally necessary for tokenization, the task of seg menting running text into words. There are roughly two classes of tokenization algorithms. In top-down tokenization, we define a standard and implement rules to implement that kind of tokenization. In bottom-up tokenization, we use simple statistics of letter sequences to break up words into subword tokens.

2.5.1 Top-down (rule-based) tokenization

While the Unix command sequence just removed all the numbers and punctuation, for most NLP applications we'll need to keep these in our tokenization. We often want to break off punctuation as a separate token; commas are a useful piece of infor mation for parsers, periods help indicate sentence boundaries. But we'll often want to keep the punctuation that occurs word internally, in examples like *m.p.h.*, *Ph.D.*, *AT&T*, and *cap'n*. Special characters and numbers will need to be kept in prices ($45.55) and dates (01/02/06); we don't want to segment that price into separate tokens

of “45” and “55”. And there are URLs (https://www.stanford.edu), Twitter hashtags (#nlproc), or email addresses (someone@cs.colorado.edu).

Number expressions introduce other complications as well; while commas nor mally appear at word boundaries, commas are used inside numbers in English, every three digits: 555,500.50. Languages, and hence tokenization requirements, differ on this; many continental European languages like Spanish, French, and German, by contrast, use a comma to mark the decimal point, and spaces (or sometimes periods) where English puts commas, for example, 555 500,50.

clitic A tokenizer can also be used to expand clitic contractions that are marked by apostrophes, for example, converting what’re to the two tokens what are, and we’re to we are. A clitic is a part of a word that can’t stand on its own, and can only occur when it is attached to another word. Some such contractions occur in other alphabetic languages, including articles and pronouns in French (j’ai, l’homme).

Depending on the application, tokenization algorithms may also tokenize mul tiword expressions like New York or rock ’n’ roll as a single token, which re quires a multiword expression dictionary of some sort. Tokenization is thus inti mately tied up with named entity recognition, the task of detecting names, dates, and organizations (Chapter 8).

Penn Treebank tokenization

One commonly used tokenization standard is known as the Penn Treebank to kenization standard, used for the parsed corpora (treebanks) released by the Linguistic Data Consortium (LDC), the source of many useful datasets. This standard separates out clitics (*doesn’t* becomes *does* plus *n’t*), keeps hyphenated words to gether, and separates out all punctuation (to save space we’re showing visible spaces ‘ ’ between tokens, although newlines is a more common output):

Input: "The San Francisco-based restaurant," they said, "doesn’t charge $10".

Output: " The San Francisco-based restaurant , " they said , " does n’t charge $ 10 " .

In practice, since tokenization needs to be run before any other language process ing, it needs to be very fast. The standard method for tokenization is therefore to use deterministic algorithms based on regular expressions compiled into very efficient

finite state automata. For example, Fig. 2.12 shows an example of a basic regular expression that can be used to tokenize English with the nltk.regexp tokenize function of the Python-based Natural Language Toolkit (NLTK) (Bird et al. 2009; https://www.nltk.org).

```
>>> text = ’That U.S.A. poster-print costs $12.40...’
>>> pattern = r’’’(?x) # set flag to allow verbose regexps ... (?:[A-Z]\.)+ #
abbreviations, e.g. U.S.A. ... | \w+(?:-\w+)* # words with optional internal
hyphens ... | \$?\d+(?:\.\d+)?%? # currency, percentages, e.g. $12.40, 82% ... |
\.\.\. # ellipsis
... | []][.,;"’?():_‘-] # these are separate tokens; includes ], [ ... ’’’
>>> nltk.regexp_tokenize(text, pattern)
[’That’, ’U.S.A.’, ’poster-print’, ’costs’, ’$12.40’, ’...’]
```

Figure 2.12 A Python trace of regular expression tokenization in the NLTK Python-based natural language processing toolkit (Bird et al., 2009), commented for readability; the (?x) verbose flag tells Python to strip comments and whitespace. Figure from Chapter 3 of Bird et al. (2009).

Carefully designed deterministic algorithms can deal with the ambiguities that arise, such as the fact that the apostrophe needs to be tokenized differently when used as a genitive marker (as in *the book's cover*), a quotative as in *'The other class', she said*, or in clitics like *they're*.

Word tokenization is more complex in languages like written Chinese, Japanese, and Thai, which do not use spaces to mark potential word-boundaries. In Chinese, hanzi for example, words are composed of characters (called hanzi in Chinese). Each character generally represents a single unit of meaning (called a morpheme) and is pronounceable as a single syllable. Words are about 2.4 characters long on average. But deciding what counts as a word in Chinese is complex. For example, consider the following sentence:

(2.4) 姚明进入总决赛 yao m ´ ´ıng j`ın ru z ` ong ju ˇ e s ´ ai`
"Yao Ming reaches the finals"

As Chen et al. (2017b) point out, this could be treated as 3 words ('Chinese Tree bank' segmentation):

(2.5) 姚明	进入	总决赛	
YaoMing	reaches	finals	

or as 5 words ('Peking University' segmentation):

(2.6) 姚	明	进入	总	决赛	
Yao	Ming	reaches	overall	finals	

Finally, it is possible in Chinese simply to ignore words altogether and use characters as the basic elements, treating the sentence as a series of 7 characters:

(2.7) 姚	明	enter	总	decisio	game	
Yao	Ming	入	overall	n		
	进	enter	决	赛		

In fact, for most Chinese NLP tasks it turns out to work better to take characters rather than words as input, since characters are at a reasonable semantic level for most applications, and since most word standards, by contrast, result in a huge vo cabulary with large numbers of very rare words (Li et al., 2019b).

However, for Japanese and Thai the character is too small a unit, and so algo rithms for word segmentation are required. These can also be useful for Chinese word segmentation in the rare situations where word rather than character boundaries are required. The standard segmentation algorithms for these languages use neural sequence mod els trained via supervised machine learning on hand-segmented training sets; we'll introduce sequence models in Chapter 8 and Chapter 9.

2.5.2 Byte-Pair Encoding: A Bottom-up Tokenization Algorithm

There is a third option to tokenizing text, one that is most commonly used by large language models. Instead of defining tokens as words (whether delimited by spaces or more complex algorithms), or as characters (as in Chinese), we can use our data to automatically tell us what the tokens should be. This is especially useful in dealing with unknown words, an important problem in language processing. As we will see in the next chapter, NLP algorithms often learn some facts about language from one corpus (a training corpus) and then use these facts to make decisions about a separate test corpus and its language. Thus if our training corpus contains, say the words *low*, *new*, *newer*, but not *lower*, then if the word

lower appears in our test corpus, our system will not know what to do with it.

To deal with this unknown word problem, modern tokenizers automatically in subwords duce sets of tokens that include tokens smaller than words, called subwords. Sub words can be arbitrary substrings, or they can be meaning-bearing units like the morphemes *-est* or *-er*. (A morpheme is the smallest meaning-bearing unit of a lan guage; for example the word *unlikeliest* has the morphemes *un-*, *likely*, and *-est*.) In modern tokenization schemes, most tokens are words, but some tokens are fre quently occurring morphemes or other subwords like *-er*. Every unseen word like *lower* can thus be represented by some sequence of known subword units, such as *low* and *er*, or even as a sequence of individual letters if necessary.

Most tokenization schemes have two parts: a token learner, and a token seg menter. The token learner takes a raw training corpus (sometimes roughly pre separated into words, for example by whitespace) and induces a vocabulary, a set of tokens. The token segmenter takes a raw test sentence and segments it into the tokens in the vocabulary. Two algorithms are widely used: byte-pair encoding (Sennrich et al., 2016), and unigram language modeling (Kudo, 2018), There is also a SentencePiece library that includes implementations of both of these (Kudo and Richardson, 2018a), and people often use the name SentencePiece to simply mean unigram language modeling tokenization.

In this section we introduce the simplest of the three, the byte-pair encoding or BPE BPE algorithm (Sennrich et al., 2016); see Fig. 2.13. The BPE token learner begins with a vocabulary that is just the set of all individual characters. It then examines the training corpus, chooses the two symbols that are most frequently adjacent (say 'A', 'B'), adds a new merged symbol 'AB' to the vocabulary, and replaces every adjacent 'A' 'B' in the corpus with the new 'AB'. It continues to count and merge, creating new longer and longer character strings, until *k* merges have been done creating *k* novel tokens; *k* is thus a parameter of the algorithm. The resulting vocabulary consists of the original set of characters plus *k* new symbols.

The algorithm is usually run inside words (not merging across word boundaries), so the input corpus is first white-space-separated to give a set of strings, each corre sponding to the characters of a word, plus a special end-of-word symbol , and its counts. Let's see its operation on the following tiny input corpus of 18 word tokens with counts for each word (the word *low* appears 5 times, the word *newer* 6 times,

and so on), which would have a starting vocabulary of 11 letters:

corpus vocabulary

5 l o w , d, e, i, l, n, o, r, s, t, w

2 l o w e s t
6 n e w e r
3 w i d e r
2 n e w

The BPE algorithm first counts all pairs of adjacent symbols: the most frequent is the pair e r because it occurs in *newer* (frequency of 6) and *wider* (frequency of 3) for a total of 9 occurrences.[2] We then merge these symbols, treating er as one symbol, and count again:

corpus vocabulary
5 l o w , d, e, i, l, n, o, r, s, t, w, er 2 l o w e s t
6 n e w er
3 w i d er
2 n e w

Now the most frequent pair is er , which we merge; our system has learned that there should be a token for word-final er, represented as er_:

corpus vocabulary
5 l o w , d, e, i, l, n, o, r, s, t, w, er, er
2 l o w e s t
6 n e w er
3 w i d er
2 n e w

Next n e (total count of 8) get merged to ne:

corpus vocabulary
5 l o w , d, e, i, l, n, o, r, s, t, w, er, er_, ne
2 l o w e s t
6 ne w er
3 w i d er
2 ne w

If we continue, the next merges are:

merge current vocabulary
(ne, w) , d, e, i, l, n, o, r, s, t, w, er, er_, ne, new
(l, o) , d, e, i, l, n, o, r, s, t, w, er, er_, ne, new, lo
(lo, w) , d, e, i, l, n, o, r, s, t, w, er, er_, ne, new, lo, low
(new, er_) , d, e, i, l, n, o, r, s, t, w, er, er_, ne, new, lo, low, newer
(low,) , d, e, i, l, n, o, r, s, t, w, er, er_, ne, new, lo, low, newer_, low

Once we've learned our vocabulary, the token segmenter is used to tokenize a test sentence. The token segmenter just runs on the test data the merges we have learned from the training data, greedily, in the order we learned them. (Thus the frequencies in the test data don't play a role, just the frequencies in the training data). So first we segment each test sentence word into characters. Then we apply the first rule: replace every instance of e r in the test corpus with er, and then the second rule: replace every instance of er in the test corpus with er_, and so on.

[2] Note that there can be ties; we could have instead chosen to merge *r* first, since that also has a frequency of 9.

function BYTE-PAIR ENCODING(strings *C*, number of merges *k*) returns vocab *V*

V←all unique characters in *C* # initial set of tokens is characters
for *i* = 1 to *k* do # merge tokens *k* times
  t_L, t_R ←Most frequent pair of adjacent tokens in *C*
  t_{NEW}←t_L + t_R # make new token by concatenating *V*←*V* + t_{NEW} # update the vocabulary
  Replace each occurrence of t_L, t_R in *C* with t_{NEW} # and update the corpus
return *V*

Figure 2.13 The token learner part of the BPE algorithm for taking a corpus broken up into individual characters or bytes, and learning a vocabulary by iteratively merging tokens. Figure adapted from Bostrom and Durrett (2020).

By the end, if the test corpus contained the character sequence n e w e r , it would be tokenized as a full word. But the characters of a new (unknown) word like l o w e r would be merged into the two tokens low er_.

Of course in real settings BPE is run with many thousands of merges on a very large input corpus. The result is that most words will be represented as full symbols, and only the very rare words (and unknown words) will have to be represented by their parts.

2.6 Word Normalization, Lemmatization and Stemming

normalization Word normalization is the task of putting words/tokens in a standard format. The case folding simplest case of word normalization is case folding. Mapping everything to lower case means that *Woodchuck* and *woodchuck* are represented identically, which is very helpful for generalization in many tasks, such as information retrieval or speech recognition. For sentiment analysis and other text classification tasks, information extraction, and machine translation, by contrast, case can be quite helpful and case folding is generally not done. This is because maintaining the difference between, for example, US the country and us the pronoun can outweigh the advantage in generalization that case folding would have provided for other words.

Systems that use BPE or other kinds of bottom-up tokenization may do no fur ther word normalization. In other NLP systems, we may want to do further nor malizations, like choosing a single normal form for words with multiple forms like USA and US or uh-huh and uhhuh. This standardization may be valuable, despite the spelling information that is lost in the normalization process. For information retrieval or information extraction about the US, we might want to see information from documents whether they mention the US or the USA.

2.6.1 Lemmatization

For other natural language processing situations we also want two morphologically different forms of a word to behave similarly. For example in web search, someone may type the string *woodchucks* but a useful system might want to also return pages that mention *woodchuck* with no *s*. This is especially common in morphologically complex languages like Polish, where for example the word *Warsaw* has different endings when it is the subject (*Warszawa*), or after a preposition like "in Warsaw" (*w*

lemmatization *Warszawie*), or "to Warsaw" (*do Warszawy*), and so on. Lemmatization is the task of determining that two words have the same root, despite their surface differences. The words *am*, *are*, and *is* have the shared lemma *be*; the words *dinner* and *dinners* both have the lemma *dinner*. Lemmatizing each of these forms to the same lemma will let us find all mentions of words in Polish like *Warsaw*. The lemmatized form of a sentence like *He is reading detective stories* would thus be *He be read detective story*.

How is lemmatization done? The most sophisticated methods for lemmatization involve complete morphological parsing of the word. Morphology is the study of morpheme the way words are built up from smaller meaning-bearing units called morphemes. stem Two broad classes of morphemes can be distinguished: stems—the central mor affix pheme of the word, supplying the main meaning—and affixes—adding "additional" meanings of various kinds. So, for example, the word *fox* consists of one morpheme (the morpheme *fox*) and the word *cats* consists of two: the morpheme *cat* and the morpheme *-s*. A morphological parser takes a word like *cats* and parses it into the two morphemes *cat* and *s*, or parses a Spanish word like *amaren* ('if in the future they would love') into the morpheme *amar* 'to love', and the morphological features *3PL* and *future subjunctive*.

Stemming: The Porter Stemmer

Lemmatization algorithms can be complex. For this reason we sometimes make use of a simpler but cruder method, which mainly consists of chopping off word stemming final affixes. This naive version of morphological analysis is called stemming. For Porter stemmer example, the Porter stemmer, a widely used stemming algorithm (Porter,

1980), when applied to the following paragraph:

> This was not the map we found in Billy Bones's chest, but an accurate copy, complete in all things-names and heights and soundings-with the single exception of the red crosses and the written notes.

produces the following stemmed output:

> Thi wa not the map we found in Billi Bone s chest but an accur copi complet in all thing name and height and sound with the singl except of the red cross and the written note The

algorithm is based on series of rewrite rules run in series: the output of each pass is fed as input to the next pass. Here are some sample rules (more details can be found at https://tartarus.org/martin/PorterStemmer/):

ATIONAL → ATE (e.g., relational → relate)

ING → if the stem contains a vowel (e.g., motoring → motor)

SSES → SS (e.g., grasses → grass)

Simple stemmers can be useful in cases where we need to collapse across differ ent variants of the same lemma. Nonetheless, they do tend to commit errors of both over- and under-generalizing, as shown in the table below (Krovetz, 1993):

Errors of Commission	Errors of Omission
organization	organ
European	Europe
doing	doe
analyzes	analysis
numerical	numerous
noisy	noise
policy	police
sparsity	sparse

2.7 Sentence Segmentation

Sentence segmentation is another important step in text processing. The most use-
sentence segmentation ful cues for segmenting a text into sentences are punctuation, like periods, question marks, and exclamation points. Question marks and exclamation points are rela tively unambiguous markers of sentence boundaries. Periods, on the other hand, are more ambiguous. The period character "." is ambiguous between a sentence bound ary marker and a marker of abbreviations like *Mr.* or *Inc.* The previous sentence that you just read showed an even more complex case of this ambiguity, in which the final period of *Inc.* marked both an abbreviation and the sentence boundary marker. For this reason, sentence tokenization and word tokenization may be addressed jointly. In general, sentence tokenization methods work by first deciding (based on rules or machine learning) whether a period is part of the word or is a sentence-boundary marker. An abbreviation dictionary can help determine whether the period is part of a commonly used abbreviation; the dictionaries can be hand-built or machine learned (Kiss and Strunk, 2006), as can the final sentence splitter. In the Stanford CoreNLP toolkit (Manning et al., 2014), for example sentence splitting is rule-based, a deterministic consequence of tokenization; a sentence ends when a sentence-ending punctuation (., !, or ?) is not already grouped with other characters into a token (such as for an abbreviation or number), optionally followed by additional final quotes or brackets.

2.8 Minimum Edit Distance

Much of natural language processing is concerned with measuring how similar two strings are. For example in spelling correction, the user typed some erroneous string—let's say graffe–and we want to know what the user meant. The user prob ably intended a word that is similar to graffe. Among candidate similar words, the word giraffe, which differs by only one letter from graffe, seems intuitively to be more similar than, say grail or graf, which differ in more letters. Another example comes from coreference, the task of deciding whether two strings such as the following refer to the same entity:

Stanford President Marc Tessier-Lavigne
Stanford University President Marc Tessier-Lavigne

Again, the fact that these two strings are very similar (differing by only one word) seems like useful evidence for deciding that they might be coreferent.

Edit distance gives us a way to quantify both of these intuitions about string sim ilarity. More formally, the minimum edit distance between two strings is defined minimum edit distance as the minimum number of editing operations (operations like insertion, deletion, substitution) needed to transform one string into another.

The gap between *intention* and *execution*, for example, is 5 (delete an i, substi tute e for n, substitute x for t, insert c, substitute u for n). It's much easier to see alignment this by looking at the most important visualization for string distances, an alignment between the two strings, shown in Fig. 2.14. Given two sequences, an alignment is a correspondence between substrings of the two sequences. Thus, we say I aligns with the empty string, N with E, and so on. Beneath the aligned strings is another representation; a series of symbols expressing an operation list for converting the

top string into the bottom string: d for deletion, s for substitution, i for insertion.

Figure 2.14 Representing the minimum edit distance between two strings as an alignment. The final row gives the operation list for converting the top string into the bottom string: d for deletion, s for substitution, i for insertion.

We can also assign a particular cost or weight to each of these operations. The Levenshtein distance between two sequences is the simplest weighting factor in which each of the three operations has a cost of 1 (Levenshtein, 1966)—we assume that the substitution of a letter for itself, for example, t for t, has zero cost. The Lev enshtein distance between *intention* and *execution* is 5. Levenshtein also proposed an alternative version of his metric in which each insertion or deletion has a cost of 1 and substitutions are not allowed. (This is equivalent to allowing substitution, but giving each substitution a cost of 2 since any substitution can be represented by one insertion and one deletion). Using this version, the Levenshtein distance between *intention* and *execution* is 8.

2.8.1 The Minimum Edit Distance Algorithm

How do we find the minimum edit distance? We can think of this as a

search task, in which we are searching for the shortest path—a sequence of edits—from one string to another.

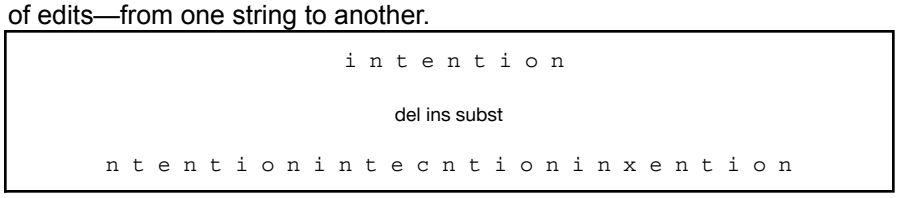


Figure 2.15 Finding the edit distance viewed as a search problem

The space of all possible edits is enormous, so we can't search naively. However, lots of distinct edit paths will end up in the same state (string), so rather than recomputing all those paths, we could just remember the shortest path to a state each time we saw it. We can do this by using dynamic programming. Dynamic programming is the name for a class of algorithms, first introduced by Bellman (1957), that apply a table-driven method to solve problems by combining solutions to subproblems. Some of the most commonly used algorithms in natural language processing make use of dynamic programming, such as the Viterbi algorithm (Chapter 8) and the CKY algorithm for parsing (Chapter 17).

dynamic programming

The intuition of a dynamic programming problem is that a large problem can be solved by properly combining the solutions to various subproblems. Consider the shortest path of transformed words that represents the minimum edit distance between the strings *intention* and *execution* shown in Fig. 2.16.

Imagine some string (perhaps it is *exention*) that is in this optimal path (whatever it is). The intuition of dynamic programming is that if *exention* is in the optimal

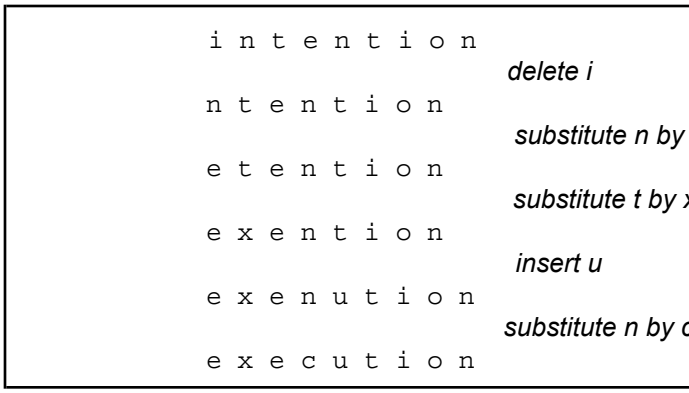


Figure 2.16 Path from *intention* to *execution*.

operation list, then the optimal sequence must also include the optimal path from *intention* to *exention*. Why? If there were a shorter path from *intention* to *exention*, then we could use it instead, resulting in a shorter overall path, and the optimal sequence wouldn't be optimal, thus leading to a contradiction.

The minimum edit distance algorithm was named by Wagner and Fischer (1974) but independently discovered by many people (see the Historical Notes sec tion of Chapter 8).

minimum edit distance

algorithm

Let's first define the minimum edit distance between two strings. Given two strings, the source string X of length n, and target string Y of length m, we'll define $D[i, j]$ as the edit distance between $X[1..i]$ and $Y[1.. j]$, i.e., the first i characters of X and the first j characters of Y. The edit distance between X and Y is thus $D[n,m]$.

We'll use dynamic programming to compute $D[n,m]$ bottom up, combining so lutions to subproblems. In the base case, with a source substring of length i but an empty target string, going from i characters to 0 requires i deletes. With a target substring of length j but an empty source going from 0 characters to j characters requires j inserts. Having computed $D[i, j]$ for small i, j we then compute larger $D[i, j]$ based on previously computed smaller values. The value of $D[i, j]$ is com puted by taking the minimum of the three possible paths through the matrix which arrive there:

$$D[i, j] = \min \begin{cases} D[i-1, j] + \text{del-cost}(source[i]) \\ D[i, j-1] + \text{ins-cost}(target[j]) \\ D[i-1, j-1] + \text{sub-cost}(source[i], target[j]) \end{cases} \tag{2.8}$$

If we assume the version of Levenshtein distance in which the insertions and dele tions each have a cost of 1 (ins-cost(·) = del-cost(·) = 1), and substitutions have a cost of 2 (except substitution of identical letters have zero cost), the computation for $D[i, j]$ becomes:

$$D[i, j] = \min \begin{cases} D[i-1, j] + 1 \\ D[i, j-1] + 1 \\ D[i-1, j-1] + \begin{cases} 0 & \text{if } source[i] = target[j] \\ 2 & \text{if } source[i] \neq target[j] \end{cases} \end{cases} \tag{2.9}$$

The algorithm is summarized in Fig. 2.17; Fig. 2.18 shows the results of applying the algorithm to the distance between *intention* and *execution* with the version of Levenshtein in Eq. 2.9.

Alignment Knowing the minimum edit distance is useful for algorithms like find ing potential spelling error corrections. But the edit distance algorithm is important in another way; with a small change, it can also provide the minimum cost align ment between two strings. Aligning two strings is useful throughout speech and

```
function MIN-EDIT-DISTANCE(source, target) returns min-distance

  n←LENGTH(source)
  m←LENGTH(target)
  Create a distance matrix D[n+1,m+1]

  # Initialization: the zeroth row and column is the distance from the
  empty string D[0,0] = 0
  for each row i from 1 to n do
      D[i,0]←D[i-1,0] + del-cost(source[i])
  for each column j from 1 to m do
      D[0,j]←D[0, j-1] + ins-cost(target[j])

  # Recurrence relation:
  for each row i from 1 to n do
        for each column j from 1 to m do
          D[i, j]←MIN( D[i−1, j] + del-cost(source[i]),
                       D[i−1, j−1] + sub-cost(source[i], target[j]),
                           D[i, j−1] + ins-cost(target[j]))
  # Termination
  return D[n,m]
```

Figure 2.17 The minimum edit distance algorithm, an example of the class of dynamic programming algorithms. The various costs can either be fixed (e.g., $\forall x$,ins-cost(x) = 1) or can be specific to the letter (to model the fact that some letters are more likely to be in serted than others). We assume that there is no cost for substituting a letter for itself (i.e., sub-cost(x, x) = 0).

Src\Tar	#	e	x	e	c	u	t	i	o	n
#	0	1	2	3	4	5	6	7	8	9
i	1	2	3	4	5	6	7	6	7	8
n	2	3	4	5	6	7	8	7	8	7
t	3	4	5	6	7	8	7	8	9	8
e	4	3	4	5	6	7	8	9	10	9
n	5	4	5	6	7	8	9	10	11	10
t	6	5	6	7	8	9	8	9	10	11
i	7	6	7	8	9	10	9	8	9	10
o	8	7	8	9	10	11	10	9	8	9
n	9	8	9	10	11	12	11	10	9	8

Figure 2.18 Computation of minimum edit distance between *intention* and *execution* with the algorithm of Fig. 2.17, using Levenshtein distance with cost of 1 for insertions or dele tions, 2 for substitutions.

language processing. In speech recognition, minimum edit distance alignment is used to compute the word error rate (Chapter 16). Alignment plays a role in ma chine translation, in which sentences in a parallel corpus

(a corpus with a text in two languages) need to be matched to each other.

To extend the edit distance algorithm to produce an alignment, we can start by visualizing an alignment as a path through the edit distance matrix. Figure 2.19 shows this path with boldfaced cells. Each boldfaced cell represents an alignment of a pair of letters in the two strings. If two boldfaced cells occur in the same row, there will be an insertion in going from the source to the target; two boldfaced cells

in the same column indicate a deletion.

Figure 2.19 also shows the intuition of how to compute this alignment path. The computation proceeds in two steps. In the first step, we augment the minimum edit distance algorithm to store backpointers in each cell. The backpointer from a cell points to the previous cell (or cells) that we came from in entering the current cell. We've shown a schematic of these backpointers in Fig. 2.19. Some cells have mul tiple backpointers because the minimum extension could have come from multiple

backtrace previous cells. In the second step, we perform a backtrace. In a backtrace, we start from the last cell (at the final row and column), and follow the pointers back through the dynamic programming matrix. Each complete path between the final cell and the initial cell is a minimum distance alignment. Exercise 2.7 asks you to modify the minimum edit distance algorithm to store the pointers and compute the backtrace to output an alignment.

	#	e	x	e	c	u	t	i	o	n
#	0	← 1	← 2	← 3	← 4	← 5	← 6	← 7	← 8	← 9
i	↑ 1	-←↑ 2	-←↑ 3	-←↑ 4	-←↑ 5	-←↑ 6	-←↑ 7	- 6	← 7	← 8
n	↑ 2	-←↑ 3	-←↑ 4	-←↑ 5	-←↑ 6	-←↑ 7	-←↑ 8	↑ 7	-←↑ 8	- 7
t	↑ 3	-←↑ 4	-←↑ 5	-←↑ 6	-←↑ 7	-←↑ 8	- 7	←↑ 8	-←↑ 9	↑ 8
e	↑ 4	- 3	← 4	-← 5	← 6	← 7	←↑ 8	-←↑ 9	-←↑ 10	↑ 9
n	↑ 5	↑ 4	-←↑ 5	-←↑ 6	-←↑ 7	-←↑ 8	-←↑ 9	-←↑ 10	-←↑ 11	-↑ 10
t	↑ 6	↑ 5	-←↑ 6	-←↑ 7	-←↑ 8	-←↑ 9	- 8	← 9	← 10	←↑ 11
i	↑ 7	↑ 6	-←↑ 7	-←↑ 8	-←↑ 9	-←↑ 10	↑ 9	- 8	← 9	← 10
o	↑ 8	↑ 7	-←↑ 8	-←↑ 9	-←↑ 10	-←↑ 11	↑ 10	↑ 9	- 8	← 9
n	↑ 9	↑ 8	-←↑ 9	-←↑ 10	-←↑ 11	-←↑ 12	↑ 11	↑ 10	↑ 9	- 8

Figure 2.19 When entering a value in each cell, we mark which of the three neighboring cells we came from with up to three arrows. After the table is full we compute an alignment (minimum edit path) by using a backtrace, starting at the 8 in the lower-right corner and following the arrows back. The sequence of bold cells

represents one possible minimum cost alignment between the two strings. Diagram design after Gusfield (1997).

While we worked our example with simple Levenshtein distance, the algorithm in Fig. 2.17 allows arbitrary weights on the operations. For spelling correction, for example, substitutions are more likely to happen between letters that are next to each other on the keyboard. The Viterbi algorithm is a probabilistic extension of minimum edit distance. Instead of computing the "minimum edit distance" between two strings, Viterbi computes the "maximum probability alignment" of one string with another. We'll discuss this more in Chapter 8.

2.9 Summary

This chapter introduced a fundamental tool in language processing, the regular ex pression, and showed how to perform basic text normalization tasks including word segmentation and normalization, sentence segmentation, and stemming. We also introduced the important minimum edit distance algorithm for comparing strings. Here's a summary of the main points we covered about these ideas:

• The regular expression language is a powerful tool for pattern-matching. • Basic operations in regular expressions include concatenation of symbols, disjunction of symbols ([], |, and .), counters (*, +, and {n,m}), anchors

(ˆ, $) and precedence operators ((,)).

• Word tokenization and normalization are generally done by cascades of simple regular expression substitutions or finite automata.

• The Porter algorithm is a simple and efficient way to do stemming, stripping off affixes. It does not have high accuracy but may be useful for some tasks.

• The minimum edit distance between two strings is the minimum number of operations it takes to edit one into the other. Minimum edit distance can be computed by dynamic programming, which also results in an alignment of the two strings.

Bibliographical and Historical Notes

Kleene 1951; 1956 first defined regular expressions and the finite automaton, based on the McCulloch-Pitts neuron. Ken Thompson was one of the first to build regular expressions compilers into editors for text searching (Thompson, 1968). His edi tor *ed* included a command "g/regular expression/p", or Global Regular Expression Print, which later became the Unix grep utility.

Text normalization algorithms have been applied since the beginning of the field. One of the earliest widely used stemmers was Lovins (1968). Stemming was also applied early to the digital humanities, by Packard (1973), who built an affix-stripping morphological parser for Ancient Greek. Currently a wide vari ety of code for tokenization and normalization is available, such as the Stanford Tokenizer (https://nlp.stanford.edu/software/tokenizer.shtml) or spe cialized tokenizers for Twitter (O'Connor et al., 2010), or for sentiment (http:

//sentiment.christopherpotts.net/tokenizing.html). See Palmer (2012) for a survey of text preprocessing. NLTK is an essential tool that offers both useful Python libraries (https://www.nltk.org) and textbook descriptions (Bird et al., 2009) of many algorithms including text normalization and corpus interfaces.

For more on Herdan's law and Heaps' Law, see Herdan (1960, p. 28), Heaps (1978), Egghe (2007) and Baayen (2001); Yasseri et al. (2012) discuss the relation ship with other measures of linguistic complexity. For more on edit distance, see the excellent Gusfield (1997). Our example measuring the edit distance from 'intention' to 'execution' was adapted from Kruskal (1983). There are various publicly avail able packages to compute edit distance, including Unix diff and the NIST sclite program (NIST, 2005).

In his autobiography Bellman (1984) explains how he originally came up with the term *dynamic programming*:

> "...The 1950s were not good years for mathematical research. [the] Secretary of Defense ...had a pathological fear and hatred of the word, research... I decided therefore to use the word, "programming". I wanted to get across the idea that this was dynamic, this was multi stage... I thought, let's ... take a word that has an absolutely precise meaning, namely dynamic... it's impossible to use the word, dynamic, in a pejorative sense. Try thinking of some combination that will pos sibly give it a pejorative meaning. It's impossible. Thus, I thought dynamic programming was a good name. It was something not even a Congressman could object to."

Exercises

2.1 Write regular expressions for the following languages.

1. the set of all alphabetic strings;
2. the set of all lower case alphabetic strings ending in a *b*;
3. the set of all strings from the alphabet *a,b* such that each *a* is immedi ately preceded by and immediately followed by a *b*;

2.2 Write regular expressions for the following languages. By "word", we mean an alphabetic string separated from other words by whitespace, any relevant punctuation, line breaks, and so forth.

1. the set of all strings with two consecutive repeated words (e.g., "Hum bert Humbert" and "the the" but not "the bug" or "the big bug");
2. all strings that start at the beginning of the line with an integer and that end at the end of the line with a word;
3. all strings that have both the word *grotto* and the word *raven* in them (but not, e.g., words like *grottos* that merely *contain* the word *grotto*);
4. write a pattern that places the first word of an English sentence in a register. Deal with punctuation.

2.3 Implement an ELIZA-like program, using substitutions such as those described on page 13. You might want to choose a different domain than a Rogerian psy chologist, although keep in mind that you would need a domain in which your program can legitimately engage in a lot of simple repetition.

2.4 Compute the edit distance (using insertion cost 1, deletion cost 1, substitution cost 1) of "leda" to "deal". Show your work (using the edit distance grid).

2.5 Figure out whether *drive* is closer to *brief* or to *divers* and what the edit

dis tance is to each. You may use any version of *distance* that you like.

2.6 Now implement a minimum edit distance algorithm and use your hand-computed results to check your code.

2.7 Augment the minimum edit distance algorithm to output an alignment; you will need to store pointers and add a stage to compute the backtrace.

CHAPTER

3 N-gram Language Models

> *"You are uniformly charming!" cried he, with a smile of associating and now and then I bowed and they perceived a chaise and four to wish for.*
>
> Random sentence generated from a Jane Austen trigram model

Predicting is difficult—especially about the future, as the old quip goes. But how about predicting something that seems much easier, like the next few words someone is going to say? What word, for example, is likely to follow

Please turn your homework ...

Hopefully, most of you concluded that a very likely word is *in*, or possibly *over*, but probably not *refrigerator* or *the*. In this chapter we formalize this intuition by introducing models that assign a probability to each possible next word.

Models that assign probabilities to upcoming words, or sequences of words language model in general, are called language models or LMs. Why would we want to predict LM upcoming words? It turns out that the large language models that revolutionized modern NLP are trained just by predicting words!! As we'll see in chapters 7-10, large language models learn an enormous amount about language solely from being trained to predict upcoming words from neighboring words.

Language models can also assign a probability to an entire sentence. For exam ple, they can predict that the following sequence has a much higher probability of appearing in a text:

all of a sudden I notice three guys standing on the sidewalk

than does this same set of words in a different order:

on guys all I of notice sidewalk three a sudden standing the

Why does it matter what the probability of a sentence is or how probable the next word is? In many NLP applications we can use the probability as a way to choose a better sentence or word over a less-appropriate one. For example we can correct grammar or spelling errors like *Their are two midterms*, in which *There* was mistyped as *Their*, or *Everything has improve*, in which *improve* should have been *improved*. The phrase *There are* will be much more probable than *Their are*, and *has improved* than *has improve*, allowing a language model to help users select the more grammatical variant. Or for a speech recognizer to realize that you said *I will be back soonish* and not *I will be bassoon dish*, it helps to know that *back soonish* is a much more probable sequence. Language

models can also help in augmentative and alternative communication systems (Trnka et al. 2007, Kane et al. 2017). People
AAC often use such AAC devices if they are physically unable to speak or sign but can instead use eye gaze or other specific movements to select words from a menu. Word prediction can be used to suggest likely words for the menu.

n-gram In this chapter we introduce the simplest kind of language model: the n-gram language model. An n-gram is a sequence of *n* words: a 2-gram (which we'll call bigram) is a two-word sequence of words like "please turn", "turn your", or "your homework", and a 3-gram (a trigram) is a three-word sequence of words like "please turn your", or "turn your homework". But we also (in a bit of terminological ambiguity) use the word 'n-gram' to mean a probabilistic model that can estimate the probability of a word given the n-1 previous words, and thereby also to assign probabilities to entire sequences.

In later chapters we will introduce the much more powerful *neural large lan guage models*, based on the transformer architecture of Chapter 10. But because n grams have a remarkably simple and clear formalization, we begin our study of lan guage modeling with them, introducing major concepts that will play a role through out language modeling, concepts like training and test sets, perplexity, sampling, and interpolation.

3.1 N-Grams

Let's begin with the task of computing $P(w|h)$, the probability of a word w given some history h. Suppose the history h is "*its water is so transparent that*" and we want to know the probability that the next word is *the*:

$$P(\textit{the}|\textit{its water is so transparent that}). \quad (3.1)$$

One way to estimate this probability is from relative frequency counts: take a very large corpus, count the number of times we see *its water is so transparent that*, and count the number of times this is followed by *the*. This would be answering the question "Out of the times we saw the history *h*, how many times was it followed by the word *w*", as follows:

$$P(\textit{the}|\textit{its water is so transparent that}) = \frac{C(\textit{its water is so transparent that the})}{C(\textit{its water is so transparent that})} \quad (3.2)$$

With a large enough corpus, such as the web, we can compute these counts and estimate the probability from Eq. 3.2. You should pause now, go to the web, and compute this estimate for yourself.

While this method of estimating probabilities directly from counts works fine in many cases, it turns out that even the web isn't big enough to give us good estimates in most cases. This is because language is creative; new sentences are created all the time, and we won't always be able to count entire sentences. Even simple extensions of the example sentence may have counts of zero on the web (such as "*Walden Pond's water is so transparent that the*"; well, *used to* have counts of zero).

Similarly, if we wanted to know the joint probability of an entire sequence of words like *its water is so transparent*, we could do it by asking "out of all possible sequences of five words, how many of them are *its water is so transparent*?" We would have to get the count of *its water is so transparent* and divide by the sum of the counts of all possible five word sequences. That seems rather a lot to estimate!

For this reason, we'll need to introduce more clever ways of estimating the prob ability of a word w given a history h, or the probability of an entire word sequence W. Let's start with a little formalizing of notation. To represent the probability of a

particular random variable $X_i$taking on the value "the", or $P(X_i = \text{"the"})$, we will use the simplification $P(the)$. We'll represent a sequence of n words either as $w_1 ...w_n$ or $w_{1:n}$. Thus the expression $w_{1:n-1}$ means the string $w_1,w_2,...,w_{n-1}$, but we'll also be using the equivalent notation $w_{<n}$, which can be read as "all the elements of w from w_1 up to and including w_{n-1}. For the joint probability of each word in a se quence having a particular value $P(X_1 = w_1,X_2 = w_2,X_3 = w_3,...,X_n = w_n)$ we'll use $P(w_1,w_2,...,w_n)$.

Now, how can we compute probabilities of entire sequences like $P(w_1,w_2,...,w_n)$? One thing we can do is decompose this probability using the chain rule of proba bility:

$$P(X_1...X_n) = P(X_1)P(X_2|X_1)P(X_3|X_{1:2})...P(X_n|X_{1:n-1}) = \prod_{k=1}^{n} P(X_k|X_{1:k-1}) \quad (3.3)$$

Applying the chain rule to words, we get

$$P(w_{1:n}) = P(w_1)P(w_2|w_1)P(w_3|w_{1:2})...P(w_n|w_{1:n-1}) = \prod_{k=1}^{n} P(w_k|w_{1:k-1}) \quad (3.4)$$

The chain rule shows the link between computing the joint probability of a sequence and computing the conditional probability of a word given previous words. Equa tion 3.4 suggests that we could estimate the joint probability of an entire sequence of words by multiplying together a number of conditional probabilities. But using the chain rule doesn't really seem to help us! We don't know any way to compute the exact probability of a word given a long sequence of preceding words, $P(w_n|w_{1:n-1})$. As we said above, we can't just estimate by counting the number of times every word occurs following every long string, because language is creative and any particular context might have never occurred before!

The intuition of the n-gram model is that instead of computing the probability of a word given its entire history, we can approximate the history by just the last few words.

bigram The bigram model, for example, approximates the probability of a word given all the previous words $P(w_n|w_{1:n-1})$ by using only the conditional probability of the preceding word $P(w_n|w_{n-1})$. In other words, instead of computing the probability

$$P(\text{the}|\text{Walden Pond's water is so transparent that}) \quad (3.5)$$

we approximate it with the probability

$$P(\text{the}|\text{that}) \quad (3.6)$$

When we use a bigram model to predict the conditional probability of the next word, we are thus making the following approximation:

$$P(w_n|w_{1:n-1}) \approx P(w_n|w_{n-1}) \quad (3.7)$$

The assumption that the probability of a word depends only on the previous word is called a Markov assumption. Markov models are the class of probabilistic models that assume we can predict the probability of some future unit without looking too

far into the past. We can generalize the bigram (which looks one word into the past) to the trigram (which looks two words into the past) and thus to the n-gram (which looks $n-1$ words into the past).

Let's see a general equation for this n-gram approximation to the conditional probability of the next word in a sequence. We'll use N here to mean the n-gram size, so $N = 2$ means bigrams and $N = 3$ means trigrams. Then we approximate the probability of a word given its entire context as follows:

$$P(w_n|w_{1:n-1}) \approx P(w_n|w_{n-N+1:n-1}) \quad (3.8)$$

Given the bigram assumption for the probability of an individual word, we can com pute the probability of a complete word sequence by substituting Eq. 3.7 into Eq. 3.4:

$$P(w_{1:n}) \approx \prod_{k=1}^{n} P(w_k|w_{k-1}) \quad (3.9)$$

How do we estimate these bigram or n-gram probabilities? An intuitive way to estimate probabilities is called maximum likelihood estimation or MLE. We get the MLE estimate for the parameters of an n-gram model by getting counts from a corpus, and normalizing the counts so that they lie between 0 and 1.[1] For example, to compute a particular bigram probability of a word w_n given a previous word w_{n-1}, we'll compute the count of the bigram $C(w_{n-1}w_n)$ and normal ize by the sum of all the bigrams that share the same first word w_{n-1}:

$$P(w_n|w_{n-1}) = \frac{C(w_{n-1}w_n)}{\sum_w C(w_{n-1}w)} \quad (3.10)$$

We can simplify this equation, since the sum of all bigram counts that start with a given word w_{n-1} must be equal to the unigram count for that word w_{n-1} (the reader should take a moment to be convinced of this):

$$P(w_n|w_{n-1}) = \frac{C(w_{n-1}w_n)}{C(w_{n-1})} \quad (3.11)$$

Let's work through an example using a mini-corpus of three sentences. We'll first need to augment each sentence with a special symbol <s> at the beginning of the sentence, to give us the bigram context of the first word. We'll also need a special end-symbol. </s>[2]

<s> I am Sam </s>
<s> Sam I am </s>
<s> I do not like green eggs and ham </s>

Here are the calculations for some of the bigram probabilities from this corpus

$P(\text{I}|\text{<s>}) = \frac{2}{3} = .67$ $P(\text{Sam}|\text{<s>}) = \frac{1}{3} = .33$ $P(\text{am}|\text{I}) = \frac{2}{3} = .67$
$P(\text{</s>}|\text{Sam}) = \frac{1}{2} = 0.5$ $P(\text{Sam}|\text{am}) = \frac{1}{2} = .5$ $P(\text{do}|\text{I}) = \frac{1}{3} = .33$

[1] For probabilistic models, normalizing means dividing by some total count so that the resulting proba bilities fall between 0 and 1.

[2] We need the end-symbol to make the bigram grammar a true probability distribution. Without an end symbol, instead of the sentence probabilities of all sentences summing to one, the sentence probabilities for all sentences *of a given length* would sum to one. This model would define an infinite set of probability distributions, with one distribution per sentence length. See Exercise 3.5.

For the general case of MLE n-gram parameter estimation:

$$P(w_n|w_{n-N+1:n-1}) = \frac{C(w_{n-N+1:n-1}\, w_n)}{C(w_{n-N+1:n-1})} \quad (3.12)$$

Equation 3.12 (like Eq. 3.11) estimates the n-gram probability by dividing the observed frequency of a particular sequence by the observed frequency of a prefix. This ratio is called a relative frequency. We said above that this use of relative frequencies as a way to estimate probabilities is an example of maximum likelihood estimation or MLE. In MLE, the resulting parameter set maximizes the likelihood of the training set T given the model M (i.e., $P(T|M)$). For example, suppose the word *Chinese* occurs 400 times in a corpus of a million words like the Brown corpus. What is the probability that a random word selected from some other text of, say, a million words will be the word *Chinese*? The MLE of its probability is $\frac{400}{1000000}$ or .0004. Now .0004 is not the best possible estimate of the probability of *Chinese* occurring in all situations; it might turn out that in some other corpus or context *Chinese* is a very unlikely word. But it is the probability that makes it *most likely* that Chinese will occur 400 times in a million-word corpus. We present ways to modify the MLE estimates slightly to get better probability estimates in Section 3.6.

relative frequency

Let's move on to some examples from a slightly larger corpus than our 14-word example above. We'll use data from the now-defunct Berkeley Restaurant Project, a dialogue system from the last century that answered questions about a database of restaurants in Berkeley, California (Jurafsky et al., 1994). Here are some text normalized sample user queries (a sample of 9332 sentences is on the website):

can you tell me about any good cantonese restaurants close by
mid priced thai food is what i'm looking for
tell me about chez panisse
can you give me a listing of the kinds of food that are available
i'm looking for a good place to eat breakfast
when is caffe venezia open during the day

Figure 3.1 shows the bigram counts from a piece of a bigram grammar from the Berkeley Restaurant Project. Note that the majority of the values are zero. In fact, we have chosen the sample words to cohere with each other; a matrix selected from a random set of eight words would be even more sparse.

	i	want	to	eat	chinese	food	lunch	spend
i	5	827	0	9	0	0	0	2
want	2	0	608	1	6	6	5	1
to	2	0	4	686	2	0	6	211
eat	0	0	2	0	16	2	42	0
chinese	1	0	0	0	0	82	1	0
food	15	0	15	0	1	4	0	0
lunch	2	0	0	0	0	1	0	0
spend	1	0	1	0	0	0	0	0

Figure 3.1 Bigram counts for eight of the words (out of $V = 1446$) in the Berkeley Restau rant Project corpus of 9332 sentences. Zero counts are in gray.

Figure 3.2 shows the bigram probabilities after normalization (dividing each cell in Fig. 3.1 by the appropriate unigram for its row, taken from the following set of unigram probabilities):

i	want	to	eat	chinese	food	lunch	spend
2533	927	2417	746	158	1093	341	278

	i	want	to	eat	chinese	food	lunch	spend
i	0.002	0.33	0	0.0036	0	0	0	0.00079
want	0.0022	0	0.66	0.0011	0.0065	0.0065	0.0054	0.0011
to	0.00083	0	0.0017	0.28	0.00083	0	0.0025	0.087
eat	0	0	0.0027	0	0.021	0.0027	0.056	0
chinese	0.0063	0	0	0	0	0.52	0.0063	0
food	0.014	0	0.014	0	0.00092	0.0037	0	0
lunch	0.0059	0	0	0	0	0.0029	0	0
spend	0.0036	0	0.0036	0	0	0	0	0

Figure 3.2 Bigram probabilities for eight words in the Berkeley Restaurant Project corpus of 9332 sentences. Zero probabilities are in gray.

Here are a few other useful probabilities:

$$P(\text{i}|\text{<s>}) = 0.25 \quad P(\text{english}|\text{want}) = 0.0011$$
$$P(\text{food}|\text{english}) = 0.5 \quad P(\text{</s>}|\text{food}) = 0.68$$

Now we can compute the probability of sentences like *I want English food* or *I want Chinese food* by simply multiplying the appropriate bigram probabilities to gether, as follows:

$$\begin{aligned} &P(\text{<s> i want english food </s>}) \\ &\quad = P(\text{i}|\text{<s>})P(\text{want}|\text{i})P(\text{english}|\text{want}) \\ &\qquad P(\text{food}|\text{english})P(\text{</s>}|\text{food}) \\ &\quad = .25 \times .33 \times .0011 \times 0.5 \times 0.68 \\ &\quad = .000031 \end{aligned}$$

We leave it as Exercise 3.2 to compute the probability of *i want chinese food*. What kinds of linguistic phenomena are captured in these bigram statistics? Some of the bigram probabilities above encode some facts that we think of as strictly syntactic in nature, like the fact that what comes after *eat* is usually a noun or an adjective, or that what comes after *to* is usually a verb. Others might be a fact about the personal assistant task, like the high probability of sentences beginning with the words *I*. And some might even be cultural rather than linguistic, like the higher probability that people are looking for Chinese versus English food.

Some practical issues: Although for pedagogical purposes we have only described trigram bigram models, in practice we might use trigram models, which condition on the 4-gram previous two words rather than the previous word, or 4-gram or even 5-gram mod 5-gram els, when there is sufficient training data. Note that for these larger n-grams, we'll need to assume extra contexts to the left and right of the sentence end. For example, to compute trigram probabilities at the very beginning of the sentence, we use two pseudo-words for the first trigram (i.e., $P(\text{I}|\text{<s><s>})$.

We always represent and compute language model probabilities in log format as log probabilities. Since probabilities are (by definition) less than or equal to log probabilities 1, the more probabilities we multiply together, the smaller the product becomes. Multiplying enough n-grams together would result in numerical underflow. By using log probabilities instead of raw probabilities, we get numbers that are not as small.

Adding in log space is equivalent to multiplying in linear space, so we combine log probabilities by adding them. The result of doing all computation and storage in log space is that we only need to convert back into probabilities if we need to report them at the end; then we can just take the exp of the logprob:

$$p_1 \times p_2 \times p_3 \times p_4 = \exp(\log p_1 + \log p_2 + \log p_3 + \log p_4) \quad (3.13)$$

In practice throughout this book, we'll use log to mean natural log (ln) when the base is not specified.

3.2 Evaluating Language Models: Training and Test Sets

The best way to evaluate the performance of a language model is to embed it in an application and measure how much the application improves. Such end-to-end evaluation is called extrinsic evaluation. Extrinsic evaluation is the only way to know if a particular improvement in the language model (or any component) is really going to help the task at hand. Thus for evaluating n-gram language models that are a component of some task like speech recognition or machine translation, we can compare the performance of two candidate language models by running the speech recognizer or machine translator twice, once with each language model, and seeing which gives the more accurate transcription.

extrinsic evaluation

Unfortunately, running big NLP systems end-to-end is often very expensive. Instead, it's helpful to have a metric that can be used to quickly evaluate potential improvements in a language model. An intrinsic evaluation metric is one that measures the quality of a model independent of any application. In the next section we'll introduce perplexity, which is the standard intrinsic metric for measuring language model performance, both for simple n-gram language models and for the more so phisticated neural large language models of Chapter 10.

intrinsic evaluation

In order to evaluate any machine learning model, we need to have at least three distinct data sets: the training set, the development set, and the test set.

training set

development set

test set

The training set is the data we use to learn the parameters of our model; for simple n-gram language models it's the corpus from which we get the counts that we normalize into the probabilities of the n-gram language model.

The test set is a different, held-out set of data, not overlapping with the training set, that we use to evaluate the model. We need a separate test set to give us an unbiased estimate of how well the model we trained can generalize when we apply it to some new unknown dataset. A machine learning model that perfectly captured the training data, but performed terribly on any other data, wouldn't be much use when it comes time to apply it to any new data or problem! We thus measure the quality of an n-gram model by its performance on this unseen test set or test corpus.

How should we choose a training and test set? The test set should reflect the language we want to use the model for. If we're going to use our language model for speech recognition of chemistry lectures, the test set should be text of chemistry lectures. If we're going to use it as part of a system for translating hotel booking re quests from Chinese to English, the test set should be text of hotel booking requests. If we want our language model to be general purpose, then the test test should be drawn from a wide variety of texts. In such cases we might collect a lot of texts from different sources, and then divide it up into a training set and a test set. It's important to do the dividing carefully; if we're building a general purpose model,

we don't want the test set to consist of only text from one document, or one author, since that wouldn't be a good measure of general performance.

Thus if we are given a corpus of text and want to compare the performance of two different n-gram models, we divide the data into training and test sets, and train the parameters of both models on the training set. We can then compare how well the two trained models fit the test set.

But what does it mean to "fit the test set"? The standard answer is simple: whichever language model assigns a higher probability to the test set—which means it more accurately predicts the test set—is a better model. Given two proba bilistic models, the better model is the one that has a tighter fit to the test data or that better predicts the details of the test data, and hence will assign a higher probability to the test data.

Since our evaluation metric is based on test set probability, it's important not to let the test sentences into the training set. Suppose we are trying to compute the probability of a particular "test" sentence. If our test sentence is part of the training corpus, we will mistakenly assign it an artificially high probability when it occurs in the test set. We call this situation training on the test set. Training on the test set introduces a bias that makes the probabilities all look too high, and causes huge inaccuracies in perplexity, the probability-based metric we introduce below.

Even if we don't train on the test set, if we test our language model on it many times after making different changes, we might implicitly tune to its characteristics, by noticing which changes seem to make the model better. For this reason, we only want to run our model on the test set once, or a very few number of times, once we are sure our model is ready.

development test

For this reason we normally instead have a third dataset called a development test set or, devset. We do all our testing on this dataset until the very end, and then we test on the test once to see how good our model is.

How do we divide our data into training, development, and test sets? We want our test set to be as large as possible, since a small test set may be accidentally un representative, but we also want as much training data as possible. At the minimum, we would want to pick the smallest test set that gives us enough statistical power to measure a statistically significant difference between two potential models. It's important that the dev set be drawn from the same kind of text as the test set, since its goal is to measure how we would do on the test set.

3.3 Evaluating Language Models: Perplexity

In practice we don't use raw probability as our metric for evaluating language mod els, but a function of probability called perplexity. Perplexity is one of the most important metrics in natural language processing, and we use it to evaluate neural language models as well.

perplexity

The perplexity (sometimes abbreviated as PP or PPL) of a language model on a test set is the inverse probability of the test set (one over the probability of the test set), normalized by the number of words. For this reason it's sometimes called the per-word perplexity. For a test set $W = w_1 w_2 ... w_N$,:

$$\text{perplexity}(W) = P(w_1 w_2 \ldots w_N)^{-\frac{1}{N}} \quad (3.14)$$

$$= \sqrt[N]{\frac{1}{P(w_1 w_2 \ldots w_N)}}$$

Or we can use the chain rule to expand the probability of W:

$$\text{perplexity}(W) = \sqrt[N]{\prod_{i=1}^{N} \frac{1}{P(w_i | w_1 \ldots w_{i-1})}} \quad (3.15)$$

Note that because of the inverse in Eq. 3.15, the higher the probability of the word sequence, the lower the perplexity. Thus the lower the perplexity of a model on the data, the better the model, and minimizing perplexity is equivalent to maximizing the test set probability according to the language model. Why does perplexity use the inverse probability? It turns out the inverse arises from the original definition of perplexity from cross-entropy rate in information theory; for those interested, the explanation is in the advanced section Section 3.9. Meanwhile, we just have to remember that perplexity has an inverse relationship with probability.

The details of computing the perplexity of a test set W depends on which lan guage model we use. Here's the perplexity of W with a unigram language model (just the geometric mean of the unigram probabilities):

$$\text{perplexity}(W) = \sqrt[N]{\prod_{i=1}^{N} \frac{1}{P(w_i)}} \quad (3.16)$$

The perplexity of W computed with a bigram language model is still a geometric mean, but now of the bigram probabilities:

$$\text{perplexity}(W) = \sqrt[N]{\prod_{i=1}^{N} \frac{1}{P(w_i | w_{i-1})}} \quad (3.17)$$

What we generally use for word sequence in Eq. 3.15 or Eq. 3.17 is the entire sequence of words in some test set. Since this sequence will cross many sentence boundaries, if our vocabulary includes a between-sentence token <EOS> or separate begin- and end-sentence markers <s> and </s> then we can include them in the probability computation. If we do, then we also include one token per sentence in the total count of word tokens N.[3]

We mentioned above that perplexity is a function of both the text and the lan guage model: given a text W, different language models will have different perplex ities. Because of this, perplexity can be used to compare different n-gram models. Let's look at an example, in which we trained unigram, bigram, and trigram gram mars on 38 million words (including start-of-sentence tokens) from the *Wall Street Journal*, using a 19,979 word vocabulary. We then computed the perplexity of each

[3] For example if we use both begin and end tokens, we would include the end-of-sentence marker </s> but not the beginning-of-sentence marker <s> in our count of N; This is because

the end-sentence token is followed directly by the begin-sentence token with probability almost 1, so we don't want the probability of that fake transition to influence our perplexity.

of these models on a test set of 1.5 million words, using Eq. 3.16 for unigrams, Eq. 3.17 for bigrams, and the corresponding equation for trigrams. The table below shows the perplexity of a 1.5 million word WSJ test set according to each of these grammars.

	Unigram	Bigram	Trigram
Perplexity	962	170	109

As we see above, the more information the n-gram gives us about the word sequence, the higher the probability the n-gram will assign to the string. A trigram model is less surprised than a unigram model because it has a better idea of what words might come next, and so it assigns them a higher probability. And the higher the probability, the lower the perplexity (since as Eq. 3.15 showed, perplexity is related inversely to the likelihood of the test sequence according to the model). So a lower perplexity can tell us that a language model is a better predictor of the words in the test set.

Note that in computing perplexities, the n-gram model P must be constructed without any knowledge of the test set or any prior knowledge of the vocabulary of the test set. Any kind of knowledge of the test set can cause the perplexity to be artificially low. The perplexity of two language models is only comparable if they use identical vocabularies.

An (intrinsic) improvement in perplexity does not guarantee an (extrinsic) im provement in the performance of a language processing task like speech recognition or machine translation. Nonetheless, because perplexity usually correlates with task improvements, it is commonly used as a convenient evaluation metric. Still, when possible a model's improvement in perplexity should be confirmed by an end-to-end evaluation on a real task.

Advanced: Perplexity as Weighted Average Branching Factor

It turns out that perplexity can also be thought of as the weighted average branch ing factor of a language. The branching factor of a language is the number of possible next words that can follow any word. If we have an artificial deterministic language of integer numbers whose vocabulary consists of the 10 digits (zero, one, two,..., nine), in which any digit can follow any other digit, then the branching factor of that language is 10.

Let's first convince ourselves that if we compute the perplexity of this artificial digit language we indeed get 10. Let's suppose that (in training and in test) each of the 10 digits occurs with exactly equal probability $P = \frac{1}{10}$. Now imagine a test string of digits of length N, and, again, assume that in the training set all the digits occurred with equal probability. By Eq. 3.15, the perplexity will be

$$\begin{aligned} \text{perplexity}(W) &= P(w_1 w_2 \ldots w_N)^{-\frac{1}{N}} \\ &= \left(\frac{1}{10^N}\right)^{-\frac{1}{N}} \\ &= \frac{1}{10^{-1}} \\ &= 10 \end{aligned} \tag{3.18}$$

But suppose that the number zero is really frequent and occurs far more often than other numbers. Let's say that 0 occur 91 times in the training set, and each of the other digits occurred 1 time each. Now we see the following test set: 0 0 0 0 0 3 0 0 0

0. We should expect the perplexity of this test set to be lower since most of the time the next number will be zero, which is very predictable, i.e. has a high probability. Thus, although the branching factor is still 10, the perplexity or *weighted* branching factor is smaller. We leave this exact calculation as exercise 3.12.

3.4 Sampling sentences from a language model

One important way to visualize what kind of knowledge a language model embodies sampling is to sample from it. Sampling from a distribution means to choose random points according to their likelihood. Thus sampling from a language model—which rep resents a distribution over sentences—means to generate some sentences, choosing each sentence according to its likelihood as defined by the model. Thus we are more likely to generate sentences that the model thinks have a high probability and less likely to generate sentences that the model thinks have a low probability.

This technique of visualizing a language model by sampling was first suggested very early on by Shannon (1948) and Miller and Selfridge (1950). It's simplest to visualize how this works for the unigram case. Imagine all the words of the English language covering the probability space between 0 and 1, each word covering an interval proportional to its frequency. Fig. 3.3 shows a visualization, using a unigram LM computed from the text of this book. We choose a random value between 0 and 1, find that point on the probability line, and print the word whose interval includes this chosen value. We continue choosing random numbers and generating words until we randomly generate the sentence-final token </s>.

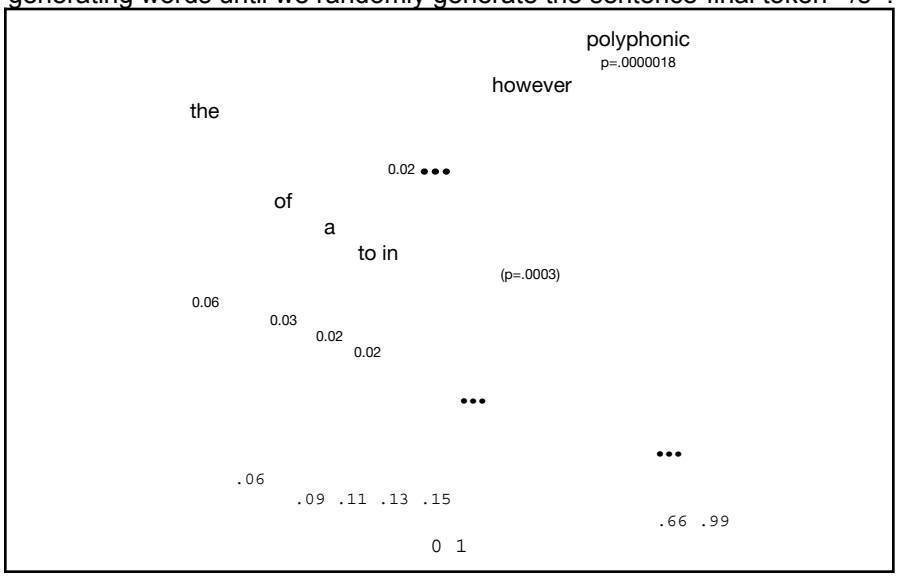


Figure 3.3 A visualization of the sampling distribution for sampling sentences by repeat edly sampling unigrams. The blue bar represents the relative frequency of each word (we've ordered them from most frequent to least frequent, but the choice of order is arbitrary). The number line shows the cumulative probabilities. If we choose a random number between 0 and 1, it will fall in an interval corresponding to some word. The expectation for the random number to fall in the larger intervals of one of the frequent words (*the*, *of*, *a*) is much higher than in the smaller interval of one of the rare words (*polyphonic*).

We can use the same technique to generate bigrams by first generating a ran dom bigram that starts with <s> (according to its bigram probability).

Let's say the second word of that bigram is *w*. We next choose a random bigram starting with *w* (again, drawn according to its bigram probability), and so on.

3.5 Generalization and Zeros

The n-gram model, like many statistical models, is dependent on the training corpus. One implication of this is that the probabilities often encode specific facts about a

given training corpus. Another implication is that n-grams do a better and better job of modeling the training corpus as we increase the value of *N*.

We can use the sampling method from the prior section to visualize both of these facts! To give an intuition for the increasing power of higher-order n-grams, Fig. 3.4 shows random sentences generated from unigram, bigram, trigram, and 4- gram models trained on Shakespeare's works.

1 –To him swallowed confess hear both. Which. Of save on trail for are ay device and rote life have

gram –Hill he late speaks; or! a more to leg less first you enter

2 –Why dost stand forth thy canopy, forsooth; he is this palpable hit the King Henry. Live king. Follow.

gram –What means, sir. I confess she? then all sorts, he is trim, captain. 3 –Fly, and will rid me these news of price. Therefore the sadness of parting, as they say, 'tis done.

gram –This shall forbid it should be branded, if renown made it empty.

4 –King Henry. What! I will go seek the traitor Gloucester. Exeunt some of the watch. A great banquet serv'd in;

gram –It cannot be but so.

Figure 3.4 Eight sentences randomly generated from four n-grams computed from Shakespeare's works. All characters were mapped to lower-case and punctuation marks were treated as words. Output is hand-corrected for capitalization to improve readability.

The longer the context on which we train the model, the more coherent the sen tences. In the unigram sentences, there is no coherent relation between words or any sentence-final punctuation. The bigram sentences have some local word-to-word coherence (especially if we consider that punctuation counts as a word). The tri gram and 4-gram sentences are beginning to look a lot like Shakespeare. Indeed, a careful investigation of

the 4-gram sentences shows that they look a little too much like Shakespeare. The words *It cannot be but so* are directly from *King John*. This is because, not to put the knock on Shakespeare, his oeuvre is not very large as corpora go (N = 884,647,V = 29,066), and our n-gram probability matrices are ridiculously sparse. There are V^2 = 844,000,000 possible bigrams alone, and the number of pos sible 4-grams is V^4 = 7×10^{17}. Thus, once the generator has chosen the first 3-gram (*It cannot be*), there are only seven possible next words for the 4th element (*but*, *I*, *that*, *thus*, *this*, and the period).

To get an idea of the dependence of a grammar on its training set, let's look at an n-gram grammar trained on a completely different corpus: the *Wall Street Journal* (WSJ) newspaper. Shakespeare and the *Wall Street Journal* are both English, so we might expect some overlap between our n-grams for the two genres. Fig. 3.5 shows sentences generated by unigram, bigram, and trigram grammars trained on 40 million words from WSJ.

Compare these examples to the pseudo-Shakespeare in Fig. 3.4. While they both model "English-like sentences", there is clearly no overlap in generated sentences, and little overlap even in small phrases. Statistical models are likely to be pretty use less as predictors if the training sets and the test sets are as different as Shakespeare and WSJ.

How should we deal with this problem when we build n-gram models? One step is to be sure to use a training corpus that has a similar genre to whatever task we are trying to accomplish. To build a language model for translating legal documents,

1 Months the my and issue of year foreign new exchange's september were recession exchange new endorsed a acquire to six executives gram

2 Last December through the way to preserve the Hudson corporation N.

B. E. C. Taylor would seem to complete the major central planners one gram point five percent of U. S. E. has already old M. X. corporation of living on information such as more frequently fishing to keep her

3 They also point to ninety nine point six billion dollars from two hundred four oh six three percent of the rates of interest stores as Mexico and gram Brazil on market conditions

Figure 3.5 Three sentences randomly generated from three n-gram models computed from 40 million words of the *Wall Street Journal*, lower-casing all characters and treating punctua tion as words. Output was then hand-corrected for capitalization to improve readability.

we need a training corpus of legal documents. To build a language model for a question-answering system, we need a training corpus of questions.

It is equally important to get training data in the appropriate dialect or variety, especially when processing social media posts or spoken transcripts. For exam ple some tweets will use features of African American

English (AAE)— the name for the many variations of language used in African American communities (King, 2020). Such features include words like *finna*—an auxiliary verb that marks imme diate future tense —that don't occur in other varieties, or spellings like *den* for *then*, in tweets like this one (Blodgett and O'Connor, 2017):

(3.19) Bored af den my phone finna die!!!

while tweets from English-based languages like Nigerian Pidgin have markedly dif ferent vocabulary and n-gram patterns from American English (Jurgens et al., 2017):

(3.20) @username R u a wizard or wat gan sef: in d mornin - u tweet, afternoon - u tweet, nyt gan u dey tweet. beta get ur IT placement wiv twitter

Matching genres and dialects is still not sufficient. Our models may still be subject to the problem of sparsity. For any n-gram that occurred a sufficient number of times, we might have a good estimate of its probability. But because any corpus is limited, some perfectly acceptable English word sequences are bound to be missing from it. That is, we'll have many cases of putative "zero probability n-grams" that should really have some non-zero probability. Consider the words that follow the bigram *denied the* in the WSJ Treebank3 corpus, together with their counts:

denied the allegations: 5
denied the speculation: 2
denied the rumors: 1
denied the report: 1

But suppose our test set has phrases like:

denied the offer
denied the loan

Our model will incorrectly estimate that the *P*(offer|denied the) is 0!

zeros These zeros—things that don't ever occur in the training set but do occur in the test set—are a problem for two reasons. First, their presence means we are underestimating the probability of all sorts of words that might occur, which will hurt the performance of any application we want to run on this data.

Second, if the probability of any word in the test set is 0, the entire probability of the test set is 0. By definition, perplexity is based on the inverse probability of the

test set. Thus if some words have zero probability, we can't compute perplexity at all, since we can't divide by 0!

What do we do about zeros? There are two solutions, depending on the kind of zero. For words whose n-gram probability is zero because they occur in a novel test set context, like the example of *denied the offer* above, we'll introduce in Section 3.6 algorithms called smoothing or discounting. Smoothing algorithms shave off a bit of probability mass from some more frequent events and give it to these unseen events. But first, let's talk about an even more insidious form of zero: words that the model has never seen before at all (in any context): unknown words!

Unknown Words

What do we do about words we have never seen before? Perhaps the word *Jurafsky* simply did not occur in our training set, but pops up in the test set! We usually disallow this situation by stipulating that we already know all the words that can occur. In such a closed vocabulary system the test set can only contain words from closed

vocabulary this known lexicon, and there will be no unknown words. This is what we do for the neural language models of later chapters. For these models we use subword tokens rather than words. With subword tokenization (like the BPE algorithm of Chapter 2) any unknown word can be modeled as a sequence of smaller subwords, if necessary by a sequence of individual letters, so we never have unknown words.

If our language model is using words instead of tokens, however, we have to OOV deal with unknown words, or out of vocabulary (OOV) words: words we haven't seen before. The percentage of OOV words that appear in the test set is called the OOV rate. One way to create an open vocabulary system is to model potential open vocabulary unknown words in the test set by adding a pseudo-word called <UNK>. Again, most modern language models are closed vocabulary and don't use an <UNK> token. But when necessary, we can train <UNK> probabilities by turning the problem back into a closed vocabulary one by choosing a fixed vocabulary in advance:

1. Choose a vocabulary (word list) that is fixed in advance.
2. Convert in the training set any word that is not in this set (any OOV word) to the unknown word token <UNK> in a text normalization step.
3. Estimate the probabilities for <UNK> from its counts just like any other regular word in the training set.

The exact choice of <UNK> has an effect on perplexity. A language model can achieve low perplexity by choosing a small vocabulary and assigning the unknown word a high probability. Thus perplexities can only be compared across language models with <UNK> if they have the exact same vocabularies (Buck et al., 2014).

3.6 Smoothing

What do we do with words that are in our vocabulary (they are not unknown words) but appear in a test set in an unseen context (for example they appear after a word they never appeared after in training)? To keep a language model from assigning zero probability to these unseen events, we'll have to shave off a bit of probability mass from some more frequent events and give it to the events we've never seen.

smoothing This modification is called smoothing or discounting. In this section and the fol
discounting lowing ones we'll introduce a variety of ways to do smoothing: Laplace (add-one)

smoothing, add-k smoothing, and stupid backoff. At the end of the chapter we also summarize a more complex method, Kneser-Ney smoothing.

3.6.1 Laplace Smoothing

The simplest way to do smoothing is to add one to all the n-gram counts, before we normalize them into probabilities. All the counts that used to be zero will now have a count of 1, the counts of 1 will be 2, and so on. This algorithm is called Laplace smoothing. Laplace smoothing does not perform well enough to be used Laplace smoothing in modern n-gram models, but it usefully introduces many of the concepts that we see in other smoothing algorithms, gives a useful baseline, and is also a practical smoothing algorithm for other tasks like text classification (Chapter 4).

Let's start with the application of Laplace smoothing to unigram probabilities. Recall that the unsmoothed maximum likelihood estimate of the unigram probability of the word w_i is its count c_i normalized by the total number of word tokens N:

$$P(w_i) = \frac{c_i}{N}$$

Laplace smoothing merely adds one to each count (hence its alternate name add-one smoothing). Since there are V words in the vocabulary and each one was incremented, we also need to adjust the denominator to take into account the extra V observations. (What happens to our P values if we don't increase the denominator?)

add-one

$$P_{\text{Laplace}}(w_i) = \frac{c_i + 1}{N + V} \tag{3.21}$$

Instead of changing both the numerator and denominator, it is convenient to describe how a smoothing algorithm affects the numerator, by defining an adjusted count c^*. This adjusted count is easier to compare directly with the MLE counts and can be turned into a probability like an MLE count by normalizing by N. To define this count, since we are only changing the numerator in addition to adding 1 we'll also need to multiply by a normalization factor $\frac{N}{N+V}$:

$$c^*_i = (c_i + 1)\frac{N}{N + V} \tag{3.22}$$

We can now turn c^*_i into a probability P^*_i by normalizing by N.

discounting A related way to view smoothing is as discounting (lowering) some non-zero counts in order to get the probability mass that will be assigned to the zero counts. Thus, instead of referring to the discounted counts c^*, we might describe a smoothing algorithm in terms of a relative discount d_c, the ratio of the discounted counts to the original counts:

discounting

$$d_c = \frac{c^*}{c}$$

Now that we have the intuition for the unigram case, let's smooth our Berkeley Restaurant Project bigrams. Figure 3.6 shows the add-one smoothed counts for the bigrams in Fig. 3.1.

Figure 3.7 shows the add-one smoothed probabilities for the bigrams in Fig. 3.2. Recall that normal bigram probabilities are computed by normalizing each row of counts by the unigram count:

i want to eat chinese food lunch spend i 6 828 1 10 1 1 1 3 want 3 1 609 2 7 7 6 2 to 3 1 5 687 3 1 7 212 eat 1 1 3 1 17 3 43 1 chinese 2 1 1 1 1 83 2 1 food 16 1 16 1 2 5 1 1 lunch 3 1 1 1 1 2 1 1 spend 2 1 2 1 1 1 1 1

Figure 3.6 Add-one smoothed bigram counts for eight of the words (out of $V = 1446$) in the Berkeley Restaurant Project corpus of 9332 sentences. Previously-zero counts are in gray.

$$P(w_n|w_{n-1}) = C(w_{n-1}w_n)$$

$$C(w_{n-1}) \quad (3.23)$$

For add-one smoothed bigram counts, we need to augment the unigram count by the number of total word types in the vocabulary V:

$$P_{\text{Laplace}}(w_n|w_{n-1}) = \frac{C(w_{n-1}w_n)+1}{\sum_w (C(w_{n-1}w)+1)} = \frac{C(w_{n-1}w_n)+1}{C(w_{n-1})+V} \quad (3.24)$$

Thus, each of the unigram counts given in the previous section will need to be augmented by $V = 1446$. The result is the smoothed bigram probabilities in Fig. 3.7.

i want to eat chinese food lunch spend i 0.0015 0.21 0.00025 0.0025 0.00025 0.00025 0.00025 0.00025 0.00075 want 0.0013 0.00042 0.26 0.00084 0.0029 0.0029 0.0025 0.00084 to 0.00078 0.00026 0.0013 0.18 0.00078 0.00026 0.0018 0.055 eat 0.00046 0.00046 0.0014 0.00046 0.0078 0.0014 0.02 0.00046 chinese 0.0012 0.00062 0.00062 0.00062 0.00062 0.052 0.0012 0.00062 food 0.0063 0.00039 0.0063 0.00039 0.00079 0.002 0.00039 0.00039 lunch 0.0017 0.00056 0.00056 0.00056 0.00056 0.0011 0.00056 0.00056 spend 0.0012 0.00058 0.0012 0.00058 0.00058 0.00058 0.00058 0.00058 Figure 3.7 Add-one smoothed bigram probabilities for eight of the words (out of $V = 1446$) in the BeRP corpus of 9332 sentences. Previously-zero probabilities are in gray.

It is often convenient to reconstruct the count matrix so we can see how much a smoothing algorithm has changed the original counts. These adjusted counts can be computed by Eq. 3.25. Figure 3.8 shows the reconstructed counts.

$$c^*(w_{n-1}w_n) = [C(w_{n-1}w_n)+1] \times \frac{C(w_{n-1})}{C(w_{n-1})+V} \quad (3.25)$$

Note that add-one smoothing has made a very big change to the counts. Com paring Fig. 3.8 to the original counts in Fig. 3.1, we can see that $C(want\ to)$ changed from 608 to 238! We can see this in probability space as well: $P(to|want)$ decreases from .66 in the unsmoothed case to .26 in the smoothed case. Looking at the dis count d (the ratio between new and old counts) shows us how strikingly the counts for each prefix word have been reduced; the discount for the bigram *want to* is .39, while the discount for *Chinese food* is .10, a factor of 10!

The sharp change in counts and probabilities occurs because too much probabil ity mass is moved to all the zeros.

i want to eat chinese food lunch spend i 3.8 527 0.64 6.4 0.64 0.64 0.64 1.9 want 1.2 0.39 238 0.78 2.7 2.7 2.3 0.78 to 1.9 0.63 3.1 430 1.9 0.63 4.4 133 eat 0.34 0.34 1 0.34 5.8 1 15 0.34 chinese 0.2 0.098 0.098 0.098 0.098 8.2 0.2 0.098 food 6.9 0.43 6.9 0.43 0.86 2.2 0.43 0.43 lunch 0.57 0.19 0.19 0.19 0.19 0.38 0.19 0.19 spend 0.32 0.16 0.32 0.16 0.16 0.16 0.16 0.16 Figure 3.8 Add-one reconstituted counts for eight words (of $V = 1446$) in the BeRP corpus of 9332 sentences. Previously-zero counts are in gray.

3.6.2 Add-k smoothing

add-k

One alternative to add-one smoothing is to move a bit less of the probability mass from the seen to the unseen events. Instead of adding 1 to each count, we add a frac tional count k (.5? .05? .01?). This algorithm is therefore called add-k smoothing.

$$P^*_{\text{Add-k}}(w_n|w_{n-1}) = \frac{C(w_{n-1}w_n) + k}{C(w_{n-1}) + kV} \quad (3.26)$$

Add-k smoothing requires that we have a method for choosing k; this can be done, for example, by optimizing on a devset. Although add-k is useful for some tasks (including text classification), it turns out that it still doesn't work well for language modeling, generating counts with poor variances and often inappropriate discounts (Gale and Church, 1994).

3.6.3 Backoff and Interpolation

The discounting we have been discussing so far can help solve the problem of zero frequency n-grams. But there is an additional source of knowledge we can draw on. If we are trying to compute $P(w_n|w_{n-2}w_{n-1})$ but we have no examples of a particular trigram $w_{n-2}w_{n-1}w_n$, we can instead estimate its probability by using the bigram probability $P(w_n|w_{n-1})$. Similarly, if we don't have counts to compute $P(w_n|w_{n-1})$, we can look to the unigram $P(w_n)$.

In other words, sometimes using less context is a good thing, helping to general ize more for contexts that the model hasn't learned much about. There are two ways backoff to use this n-gram "hierarchy". In backoff, we use the trigram if the evidence is sufficient, otherwise we use the bigram, otherwise the unigram. In other words, we only "back off" to a lower-order n-gram if we have zero evidence for a higher-order interpolation n-gram. By contrast, in interpolation, we always mix the probability estimates from all the n-gram estimators, weighting and combining the trigram, bigram, and unigram counts.

In simple linear interpolation, we combine different order n-grams by linearly interpolating them. Thus, we estimate the trigram probability $P(w_n|w_{n-2}w_{n-1})$ by mixing together the unigram, bigram, and trigram probabilities, each weighted by a λ:

$$\hat{P}(w_n|w_{n-2}w_{n-1}) = \lambda_1 P(w_n) + \lambda_2 P(w_n|w_{n-1}) + \lambda_3 P(w_n|w_{n-2}w_{n-1}) \quad (3.27)$$

The λs must sum to 1, making Eq. 3.27 equivalent to a weighted average. In a slightly more sophisticated version of linear interpolation, each λ weight is com puted by conditioning on the context. This way, if we have particularly accurate counts for a particular bigram, we assume that the counts of the trigrams based on this bigram will be more trustworthy, so we can make the λs for those trigrams higher and thus give that trigram more weight in the interpolation. Equation 3.28 shows the equation for interpolation with context-conditioned weights:

$$\hat{P}(w_n|w_{n-2}w_{n-1}) = \lambda_1(w_{n-2:n-1})P(w_n) + \lambda_2(w_{n-2:n-1})P(w_n|w_{n-1}) + \lambda_3(w_{n-2:n-1})P(w_n|w_{n-2}w_{n-1}) \quad (3.28)$$

How are these λ values set? Both the simple interpolation and conditional interpo held-out lation λs are learned from a held-out corpus. A held-out corpus is an additional training corpus, so-called because we hold it out from the training data,

that we use to set hyperparameters like these λ values. We do so by choosing the λ values that maximize the likelihood of the held-out corpus. That is, we fix the n-gram probabil ities and then search for the λ values that—when plugged into Eq. 3.27—give us the highest probability of the held-out set. There are various ways to find this optimal set of λs. One way is to use the EM algorithm, an iterative learning algorithm that converges on locally optimal λs (Jelinek and Mercer, 1980).

In a backoff n-gram model, if the n-gram we need has zero counts, we approx imate it by backing off to the (n-1)-gram. We continue backing off until we reach a history that has some counts.

In order for a backoff model to give a correct probability distribution, we have discount to discount the higher-order n-grams to save some probability mass for the lower order n-grams. Just as with add-one smoothing, if the higher-order n-grams aren't discounted and we just used the undiscounted MLE probability, then as soon as we replaced an n-gram which has zero probability with a lower-order n-gram, we would be adding probability mass, and the total probability assigned to all possible strings by the language model would be greater than 1! In addition to this explicit discount factor, we'll need a function α to distribute this probability mass to the lower order n-grams.

Katz backoff This kind of backoff with discounting is also called Katz backoff. In Katz back off we rely on a discounted probability P^*if we've seen this n-gram before (i.e., if we have non-zero counts). Otherwise, we recursively back off to the Katz probabil ity for the shorter-history (n-1)-gram. The probability for a backoff n-gram P_{BO} is thus computed as follows:

$$P_{\text{BO}}(w_n|w_{n-N+1:n-1}) = \begin{cases} P^*(w_n|w_{n-N+1:n-1}), \text{ if } C(w_{n-N+1:n}) > 0 \\ \alpha(w_{n-N+1:n-1}) P_{\text{BO}}(w_n|w_{n-N+2:n-1}), \text{ otherwise.} \end{cases} \quad (3.29)$$

Good-Turing Katz backoff is often combined with a smoothing method called Good-Turing. The combined Good-Turing backoff algorithm involves quite detailed computation for estimating the Good-Turing smoothing and the $P^*$and α values.

3.7 Huge Language Models and Stupid Backoff

By using text from the web or other enormous collections, it is possible to build extremely large language models. The Web 1 Trillion 5-gram corpus released by Google includes various large sets of n-grams, including 1-grams through 5-grams from all the five-word sequences that appear at least 40 times from 1,024,908,267,229
words of text from publicly accessible Web pages in English (Franz and Brants, 2006). Google has also released Google Books Ngrams corpora with n-grams drawn from their book collections, including another 800 billion tokens of n-grams from Chinese, English, French, German, Hebrew, Italian, Russian, and Spanish (Lin et al., 2012a). Smaller but more carefully curated n-gram corpora for English include the million most frequent n-grams drawn from the COCA (Corpus of Contempo rary American English) 1 billion word corpus of American English (Davies, 2020). COCA is a balanced corpus, meaning that it has roughly equal numbers of words from different genres: web, newspapers, spoken conversation transcripts, fiction, and so on, drawn from the period 1990-2019, and has the context of

each n-gram as well as labels for genre and provenance. Some example 4-grams from the Google Web corpus:

4-gram	Count
serve as the incoming	92
serve as the incubator	99
serve as the independent	794
serve as the index	223
serve as the indication	72
serve as the indicator	120
serve as the indicators	45

Efficiency considerations are important when building language models that use such large sets of n-grams. Rather than store each word as a string, it is generally represented in memory as a 64-bit hash number, with the words themselves stored on disk. Probabilities are generally quantized using only 4-8 bits (instead of 8-byte floats), and n-grams are stored in reverse tries.

An n-gram language model can also be shrunk by pruning, for example only storing n-grams with counts greater than some threshold (such as the count threshold of 40 used for the Google n-gram release) or using entropy to prune less-important n-grams (Stolcke, 1998). Another option is to build approximate language models

Bloom filters

using techniques like Bloom filters (Talbot and Osborne 2007, Church et al. 2007). Finally, efficient language model toolkits like KenLM (Heafield 2011, Heafield et al. 2013) use sorted arrays, efficiently combine probabilities and backoffs in a single value, and use merge sorts to efficiently build the probability tables in a minimal number of passes through a large corpus.

Although with these toolkits it is possible to build web-scale language models using advanced smoothing algorithms like the Kneser-Ney algorithm we will see in Section 3.8, Brants et al. (2007) show that with very large language models a much

stupid backoff

simpler algorithm may be sufficient. The algorithm is called stupid backoff. Stupid backoff gives up the idea of trying to make the language model a true probability dis tribution. There is no discounting of the higher-order probabilities. If a higher-order n-gram has a zero count, we simply backoff to a lower order n-gram, weighed by a fixed (context-independent) weight. This algorithm does not produce a probability

distribution, so we'll follow Brants et al. (2007) in referring to it as S:

$$S(w_i|w_{i-N+1:i-1}) = \begin{cases} \text{count}(w_{i-N+1:i}) & \text{if } \text{count}(w_{i-N+1:i-1}) \\ \text{count}(w_{i-N+1:i}) > 0 & \\ \lambda S(w_i|w_{i-N+2:i-1}) & \text{otherwise} \end{cases} \quad (3.30)$$

The backoff terminates in the unigram, which has score $S(w) = \frac{count(w)}{N}$. Brants et al. (2007) find that a value of 0.4 worked well for λ.

Kneser-Ney Smoothing

3.8 Advanced:

Kneser-Ney A popular advanced n-gram smoothing method is the interpolated Kneser-Ney al gorithm (Kneser and Ney 1995, Chen and Goodman 1998).

3.8.1 Absolute Discounting

Kneser-Ney has its roots in a method called absolute discounting. Recall that dis counting of the counts for frequent n-grams is necessary to save some probability mass for the smoothing algorithm to distribute to the unseen n-grams.

To see this, we can use a clever idea from Church and Gale (1991). Consider an n-gram that has count 4. We need to discount this count by some amount. But how much should we discount it? Church and Gale's clever idea was to look at a held-out corpus and just see what the count is for all those bigrams that had count 4 in the training set. They computed a bigram grammar from 22 million words of AP newswire and then checked the counts of each of these bigrams in another 22 million words. On average, a bigram that occurred 4 times in the first 22 million words occurred 3.23 times in the next 22 million words. Fig. 3.9 from Church and Gale (1991) shows these counts for bigrams with c from 0 to 9.

Bigram count in training set	Bigram count in heldout set
0	0.0000270
1	0.448
2	1.25
3	2.24
4	3.23
5	4.21
6	5.23
7	6.21
8	7.21
9	8.26

Figure 3.9 For all bigrams in 22 million words of AP newswire of count 0, 1, 2,...,9, the counts of these bigrams in a held-out corpus also of 22 million words.

Notice in Fig. 3.9 that except for the held-out counts for 0 and 1, all the other bigram counts in the held-out set could be estimated pretty well by just subtracting 0.75 from the count in the training set! Absolute discounting formalizes this intu- absolute discounting ition by subtracting a fixed (absolute) discount d from each count. The intuition is that since we have good estimates already for the very high counts, a small discount

d won't affect them much. It will mainly modify the smaller counts, for which we don't necessarily trust the estimate anyway, and Fig. 3.9 suggests that in practice this discount is actually a good one for bigrams with counts 2 through 9. The equation for interpolated absolute discounting applied to bigrams:

$$P_{\text{AbsoluteDiscounting}}(w_i|w_{i-1}) = \frac{C(w_{i-1}w_i)-d}{\sum_v C(w_{i-1} \quad v)} + \lambda(w_{i-1})P(w_i)$$

(3.31)

The first term is the discounted bigram, with $0 \leq d \leq 1$, and the second term is the unigram with an interpolation weight λ. By inspection of Fig. 3.9, it looks like just setting all the d values to .75 would work very well, or perhaps keeping a separate second discount value of 0.5 for the bigrams with counts of 1. There are principled methods for setting d; for example, Ney et al. (1994) set d as a function of n_1 and n_2, the number of unigrams that have a count of 1 and a count of 2, respectively:

$$d = \frac{n_1}{n_1 + 2n_2} \tag{3.32}$$

3.8.2 Kneser-Ney Discounting

Kneser-Ney discounting (Kneser and Ney, 1995) augments absolute discounting with a more sophisticated way to handle the lower-order unigram distribution. Con sider the job of predicting the next word in this sentence, assuming we are interpo lating a bigram and a unigram model.

I can't see without my reading .

The word *glasses* seems much more likely to follow here than, say, the word *Kong*, so we'd like our unigram model to prefer *glasses*. But in fact it's *Kong* that is more common, since *Hong Kong* is a very frequent word. A standard unigram model will assign *Kong* a higher probability than *glasses*. We would like to capture the intuition that although *Kong* is frequent, it is mainly only frequent in the phrase *Hong Kong*, that is, after the word *Hong*. The word *glasses* has a much wider distribution.

In other words, instead of $P(w)$, which answers the question "How likely is w?", we'd like to create a unigram model that we might call $P_{\text{CONTINUATION}}$, which answers the question "How likely is w to appear as a novel continuation?". How can we estimate this probability of seeing the word w as a novel continuation, in a new unseen context? The Kneser-Ney intuition is to base our estimate of $P_{\text{CONTINUATION}}$ on the *number of different contexts word w has appeared in*, that is, the number of bigram types it completes. Every bigram type was a novel continuation the first time it was seen. We hypothesize that words that have appeared in more contexts in the past are more likely to appear in some new context as well. The number of times a word w appears as a novel continuation can be expressed as:

$$P_{\text{CONTINUATION}}(w) \propto |\{v : C(vw) > 0\}| \tag{3.33}$$

To turn this count into a probability, we normalize by the total number of word bigram types. In summary:

$$P_{\text{CONTINUATION}}(w) = \frac{|\{v : C(vw) > 0\}|}{|\{(u^0, w^0) : C(u^0 w^0) > 0\}|} \tag{3.34}$$

An equivalent formulation based on a different metaphor is to use the number of word types seen to precede w (Eq. 3.33 repeated):

$$P_{\text{CONTINUATION}}(w) \propto |\{v : C(vw) > 0\}| \tag{3.35}$$

normalized by the number of words preceding all words, as

follows: $P_{\text{CONTINUATION}}(w) = |\{v : C(vw) > 0\}|$

$$\sum_{w'} |\{v : C(vw') > 0\}| \quad (3.36)$$

A frequent word (Kong) occurring in only one context (Hong) will have a low con tinuation probability.

The final equation for Interpolated Kneser-Ney smoothing for bigrams is then: Interpolated Kneser-Ney

$$P_{\text{KN}}(w_i|w_{i-1}) = \max(C(w_{i-1}w_i)-d,0)$$

$$C(w_{i-1})+\lambda(w_{i-1})P_{\text{CONTINUATION}}(w_i) \quad (3.37)$$

The λ is a normalizing constant that is used to distribute the probability mass we've discounted:

$$\lambda(w_{i-1}) = d$$

$$\sum_{v} C(w_{i-1}v)|\{w : C(w_{i-1}w) > 0\}| \quad (3.38)$$

The first term, d

$$\sum_{v} C(w_{i-1}v)$$

, is the normalized discount (the discount d, $0 \le d \le 1$, was introduced in the absolute discounting section above). The second term, $|\{w : C(w_{i-1}w) > 0\}|$, is the number of word types that can follow w_{i-1} or, equiva lently, the number of word types that we discounted; in other words, the number of times we applied the normalized discount.

The general recursive formulation is as follows:

$$P_{\text{KN}}(w_i|w_{i-n+1:i-1}) = \max(c_{KN}(w_{i-n+1:i})-d,0)$$

$$\sum_{v} c_{KN}(w_{i-n+1:i-1}v)+\lambda(w_{i-n+1:i-1})P_{KN}(w_i|w_{i-n+2:i-1}) \quad (3.39)$$

where the definition of the count c_{KN} depends on whether we are counting the highest-order n-gram being interpolated (for example trigram if we are interpolating trigram, bigram, and unigram) or one of the lower-order n-grams (bigram or unigram if we are interpolating trigram, bigram, and unigram):

$$c_{KN}(\cdot) = \begin{cases} \text{continuationcount}(\cdot) & \text{for lower orders} \\ \text{count}(\cdot) & \text{for the highest order} \end{cases} \quad (3.40)$$

The continuation count of a string $\cdot$ is the number of unique single word contexts for that string $\cdot$.

At the termination of the recursion, unigrams are interpolated with the uniform distribution, where the parameter is the empty string:

$$P_{\text{KN}}(w) = \max(c_{KN}(w)-d,0) \quad \sum_{w'} c_{KN}(w')+\lambda(\quad)\frac{1}{V} \quad (3.41)$$

If we want to include an unknown word <UNK>, it's just included as a regular vo cabulary entry with count zero, and hence its probability will be a lambda-weighted uniform distribution $\frac{\lambda(\ )}{V}$

$\cdot$

The best performing version of Kneser-Ney smoothing is called modified Kneser Ney smoothing, and is due to Chen and Goodman (1998). Rather than use a single fixed discount d, modified Kneser-Ney uses three different discounts d_1, d_2, and d_{3+} for n-grams with counts of 1, 2 and three or more, respectively. See Chen and Goodman (1998, p. 19) or Heafield et al. (2013) for the details.

modified Kneser-Ney

3.9 Advanced: Perplexity's Relation to Entropy

We introduced perplexity in Section 3.3 as a way to evaluate n-gram models on a test set. A better n-gram model is one that assigns a higher probability to the test data, and perplexity is a normalized version of the probability of the test set. The perplexity measure actually arises from the information-theoretic concept of cross-entropy, which explains otherwise mysterious properties of perplexity (why the inverse probability, for example?) and its relationship to entropy. Entropy is a measure of information. Given a random variable X ranging over whatever we are predicting (words, letters, parts of speech, the set of which we'll call χ) and with a particular probability function, call it $p(x)$, the entropy of the random variable X is:

Entropy

$$H(X) = -\sum_{x \in \chi} p(x)\log_2 p(x) \quad (3.42)$$

The log can, in principle, be computed in any base. If we use log base 2, the resulting value of entropy will be measured in bits.

One intuitive way to think about entropy is as a lower bound on the number of bits it would take to encode a certain decision or piece of information in the optimal coding scheme.

Consider an example from the standard information theory textbook Cover and Thomas (1991). Imagine that we want to place a bet on a horse race but it is too far to go all the way to Yonkers Racetrack, so we'd like to send a short message to the bookie to tell him which of the eight horses to bet on. One way to encode this message is just to use the binary representation of the horse's number as the code; thus, horse 1 would be 001, horse 2 010, horse 3 011, and so on, with horse 8 coded as 000. If we spend the whole day betting and each horse is coded with 3 bits, on average we would be sending 3 bits per race.

Can we do better? Suppose that the spread is the actual distribution of the bets placed and that we represent it as the prior probability of each horse as follows:

Horse 1	$\frac{1}{2}$	Horse 5	$\frac{1}{64}$
Horse 2	$\frac{1}{4}$	Horse 6	$\frac{1}{64}$
Horse 3	$\frac{1}{8}$	Horse 7	$\frac{1}{64}$
Horse 4	$\frac{1}{16}$	Horse 8	$\frac{1}{64}$

The entropy of the random variable X that ranges over horses gives us a lower bound on the number of bits and is

$$H(X) = -\sum_{i=1}^{i=8} p(i)\log_2 p(i)$$

$$= -\frac{1}{2}\log_2\frac{1}{2} - \frac{1}{4}\log_2\frac{1}{4} - \frac{1}{8}\log_2\frac{1}{8} - \frac{1}{16}\log_2\frac{1}{16} - 4(\frac{1}{64}\log_2\frac{1}{64})$$
$$= 2 \text{ bits} \quad (3.43)$$

A code that averages 2 bits per race can be built with short encodings for more probable horses, and longer encodings for less probable horses. For example, we could encode the most likely horse with the code 0, and the remaining horses as 10, then 110, 1110, 111100, 111101, 111110, and 111111.

What if the horses are equally likely? We saw above that if we used an equal length binary code for the horse numbers, each horse took 3 bits to code, so the average was 3. Is the entropy the same? In this case each horse would have a probability of $\frac{1}{8}$. The entropy of the choice of horses is then

$$H(X) = -\sum_{i=1}^{i=8} \frac{1}{8}\log_2\frac{1}{8} = -\log_2\frac{1}{8} = 3 \text{ bits} \quad (3.44)$$

Until now we have been computing the entropy of a single variable. But most of what we will use entropy for involves *sequences*. For a grammar, for example, we will be computing the entropy of some sequence of words $W = \{w_1, w_2, \ldots, w_n\}$. One way to do this is to have a variable that ranges over sequences of words. For example we can compute the entropy of a random variable that ranges over all finite sequences of words of length n in some language L as follows:

$$H(w_1, w_2, \ldots, w_n) = -\sum_{w_{1:n} \in L} p(w_{1:n}) \log p(w_{1:n}) \quad (3.45)$$

entropy rate We could define the entropy rate (we could also think of this as the per-word entropy) as the entropy of this sequence divided by the number of words:

$$\frac{1}{n}H(w_{1:n}) = -\frac{1}{n}\sum_{w_{1:n} \in L} p(w_{1:n}) \log p(w_{1:n}) \quad (3.46)$$

But to measure the true entropy of a language, we need to consider sequences of infinite length. If we think of a language as a stochastic process L that produces a sequence of words, and allow W to represent the sequence of words $w_1, \ldots, w_n$, then L's entropy rate $H(L)$ is defined as

$$H(L) = \lim_{n\to\infty} \frac{1}{n}H(w_1, w_2, \ldots, w_n)$$
$$= -\lim_{n\to\infty} \frac{1}{n} \sum_{W \in L} p(w_1, \ldots, w_n) \log p(w_1, \ldots, w_n) \quad (3.47)$$

The Shannon-McMillan-Breiman theorem (Algoet and Cover 1988, Cover and Thomas 1991) states that if the language is regular in certain ways (to be exact, if it is both stationary and ergodic),

$$H(L) = \lim_{n\to\infty} -\frac{1}{n}\log p(w_1 w_2 \ldots w_n) \quad (3.48)$$

That is, we can take a single sequence that is long enough instead of summing over all possible sequences. The intuition of the Shannon-McMillan-Breiman the orem is that a long-enough sequence of words will contain in it many other shorter sequences and that each of these shorter sequences will reoccur in the longer se quence according to their probabilities.

Stationary A stochastic process is said to be stationary if the probabilities it assigns to a sequence are invariant with respect to shifts in the time index. In other words, the probability distribution for words at time t is the same as the probability distribution at time $t + 1$. Markov models, and hence n-grams, are stationary. For example, in a bigram, $P_i$is dependent only on P_{i-1}. So if we shift our time index by x, P_{i+x} is still dependent on P_{i+x-1}. But natural language is not stationary, since as we show

in Appendix D, the probability of upcoming words can be dependent on events that were arbitrarily distant and time dependent. Thus, our statistical models only give an approximation to the correct distributions and entropies of natural language.

To summarize, by making some incorrect but convenient simplifying assump tions, we can compute the entropy of some stochastic process by taking a very long sample of the output and computing its average log probability.

cross-entropy Now we are ready to introduce cross-entropy. The cross-entropy is useful when we don't know the actual probability distribution p that generated some data. It allows us to use some m, which is a model of p (i.e., an approximation to p). The cross-entropy of m on p is defined by

$$H(p,m) = \lim_{n\to\infty} -\frac{1}{n} \sum_{W\in L} p(w_1,\ldots,w_n)\log m(w_1,\ldots,w_n) \quad (3.49)$$

That is, we draw sequences according to the probability distribution p, but sum the log of their probabilities according to m.

Again, following the Shannon-McMillan-Breiman theorem, for a stationary er godic process:

$$H(p,m) = \lim_{n\to\infty} -\frac{1}{n}\log m(w_1 w_2 \ldots w_n) \quad (3.50)$$

This means that, as for entropy, we can estimate the cross-entropy of a model m on some distribution p by taking a single sequence that is long enough instead of summing over all possible sequences.

What makes the cross-entropy useful is that the cross-entropy $H(p,m)$ is an up per bound on the entropy $H(p)$. For any model m:

$$H(p) \le H(p,m) \quad (3.51)$$

This means that we can use some simplified model m to help estimate the true en tropy of a sequence of symbols drawn according to probability p. The more accurate m is, the closer the cross-entropy $H(p,m)$ will be to the true entropy $H(p)$. Thus, the difference between $H(p,m)$ and $H(p)$ is a measure of how accurate a model is. Between two models m_1 and m_2, the more accurate model will be the one with the lower cross-entropy. (The

cross-entropy can never be lower than the true entropy, so a model cannot err by underestimating the true entropy.)

We are finally ready to see the relation between perplexity and cross-entropy as we saw it in Eq. 3.50. Cross-entropy is defined in the limit as the length of the observed word sequence goes to infinity. We will need an approximation to cross entropy, relying on a (sufficiently long) sequence of fixed length. This approxima tion to the cross-entropy of a model $M = P(w_i|w_{i-N+1:i-1})$ on a sequence of words W is

$$H(W) = -\frac{1}{N}\log P(w_1 w_2 \ldots w_N) \quad (3.52)$$

perplexity The perplexity of a model P on a sequence of words W is now formally defined as 2 raised to the power of this cross-entropy:

$$\begin{aligned}\text{Perplexity}(W) &= 2^{H(W)} \\ &= P(w_1 w_2 \ldots w_N)^{-\frac{1}{N}} \\ &= \sqrt[N]{\frac{1}{P(w_1 w_2 \ldots w_N)}} \\ &= \sqrt[N]{\prod_{i=1}^{N} \frac{1}{P(w_i|w_1 \ldots w_{i-1})}} \quad (3.53)\end{aligned}$$

3.10 Summary

This chapter introduced language modeling and the n-gram, one of the most widely used tools in language processing.

- Language models offer a way to assign a probability to a sentence or other sequence of words, and to predict a word from preceding words.
- n-grams are Markov models that estimate words from a fixed window of pre vious words. n-gram probabilities can be estimated by counting in a corpus and normalizing (the maximum likelihood estimate).
- n-gram language models are evaluated extrinsically in some task, or intrinsi cally using perplexity.
- The perplexity of a test set according to a language model is the geometric mean of the inverse test set probability computed by the model.
- Smoothing algorithms provide a more sophisticated way to estimate the prob ability of n-grams. Commonly used smoothing algorithms for n-grams rely on lower-order n-gram counts through backoff or interpolation.
- Both backoff and interpolation require discounting to create a probability dis tribution.
- Kneser-Ney smoothing makes use of the probability of a word being a

novel continuation. The interpolated Kneser-Ney smoothing algorithm mixes a discounted probability with a lower-order continuation probability.

Bibliographical and Historical Notes

The underlying mathematics of the n-gram was first proposed by Markov (1913), who used what are now called Markov chains (bigrams and trigrams) to predict whether an upcoming letter in Pushkin's *Eugene Onegin* would be a vowel or a con sonant. Markov classified 20,000 letters as V or C and computed the bigram and trigram probability that a given letter would be a vowel given the previous one or two letters. Shannon (1948) applied n-grams to compute approximations to English word sequences. Based on Shannon's work, Markov models were commonly used in engineering, linguistic, and psychological work on modeling word sequences by the 1950s. In a series of extremely influential papers starting with Chomsky (1956) and including Chomsky (1957) and Miller and Chomsky (1963), Noam Chomsky argued that "finite-state Markov processes", while a possibly useful engineering heuristic,

were incapable of being a complete cognitive model of human grammatical knowl edge. These arguments led many linguists and computational linguists to ignore work in statistical modeling for decades.

The resurgence of n-gram models came from Fred Jelinek and colleagues at the IBM Thomas J. Watson Research Center, who were influenced by Shannon, and James Baker at CMU, who was influenced by the prior, classified work of Leonard Baum and colleagues on these topics at labs like IDA. Independently these two labs successfully used n-grams in their speech recognition systems at the same time (Baker 1975b, Jelinek et al. 1975, Baker 1975a, Bahl et al. 1983, Jelinek 1990). The terms "language model" and "perplexity" were first used for this technology by the IBM group. Jelinek and his colleagues used the term *language model* in pretty modern way, to mean the entire set of linguistic influences on word sequence prob abilities, including grammar, semantics, discourse, and even speaker characteristics, rather than just the particular n-gram model itself.

Add-one smoothing derives from Laplace's 1812 law of succession and was first applied as an engineering solution to the zero frequency problem by Jeffreys (1948) based on an earlier Add-K suggestion by Johnson (1932). Problems with the add one algorithm are summarized in Gale and Church (1994).

A wide variety of different language modeling and smoothing techniques were proposed in the 80s and 90s, including Good-Turing discounting—first applied to the n-gram smoothing at IBM by Katz (Nadas ´ 1984, Church and Gale 1991)— Witten Bell discounting (Witten and Bell, 1991), and varieties of class-based n-gram mod-

class-based n-gram els that used information about word classes. Starting in the late 1990s, Chen and Goodman performed a number of carefully controlled experiments comparing differ ent discounting algorithms, cache models, class-based models, and other language model parameters (Chen and Goodman 1999, Goodman 2006, inter alia). They showed the advantages of Modified Interpolated Kneser-Ney, which became the standard baseline for n-gram language modeling, especially because they showed that caches and class-based models provided only minor additional improvement. SRILM (Stolcke, 2002) and KenLM (Heafield 2011, Heafield et al. 2013) are pub licly available toolkits for building n-gram language models.

Modern language modeling is more commonly done with neural network lan guage models, which solve the major problems with n-grams: the number of param eters increases exponentially as the n-gram order increases, and n-grams have no way to generalize from training to test set. Neural language models instead project words into a continuous space in which words with similar contexts have similar representations. We'll introduce feedforward language models (Bengio et al. 2006, Schwenk 2007) in Chapter 7, recurrent language models (Mikolov, 2012) in Chap ter 9, and transformer-based large language models in Chapter 10.

Exercises

3.1 Write out the equation for trigram probability estimation (modifying Eq. 3.11). Now write out all the non-zero trigram probabilities for the I am Sam corpus on page 35.

3.2 Calculate the probability of the sentence i want chinese food. Give two probabilities, one using Fig. 3.2 and the 'useful probabilities' just below it on page 37, and another using the add-1 smoothed table in Fig. 3.7. Assume the additional add-1 smoothed probabilities $P(\text{i}|\text{<s>}) = 0.19$ and $P(\text{</s>}|\text{food}) = 0.40$.

3.3 Which of the two probabilities you computed in the previous exercise is higher, unsmoothed or smoothed? Explain why.

3.4 We are given the following corpus, modified from the one in the chapter:

<s> I am Sam </s>
<s> Sam I am </s>
<s> I am Sam </s>
<s> I do not like green eggs and Sam </s>

Using a bigram language model with add-one smoothing, what is P(Sam | am)? Include <s> and </s> in your counts just like any other token.

3.5 Suppose we didn't use the end-symbol </s>. Train an unsmoothed bigram grammar on the following training corpus without using the end-symbol </s>:

<s> a b
<s> b b
<s> b a
<s> a a

Demonstrate that your bigram model does not assign a single probability dis tribution across all sentence lengths by showing that the sum of the probability of the four possible 2 word sentences over the alphabet *{*a,b*}* is 1.0, and the sum of the probability of all possible 3 word sentences over the alphabet *{*a,b*}* is also 1.0.

3.6 Suppose we train a trigram language model with add-one smoothing on a given corpus. The corpus contains V word types. Express a formula for esti mating P(w3|w1,w2), where w3 is a word which follows the bigram (w1,w2), in terms of various n-gram counts and V. Use the notation c(w1,w2,w3) to denote the number of times that trigram (w1,w2,w3) occurs in the corpus, and so on for bigrams and unigrams.

3.7 We are given the following corpus, modified from the one in the chapter:

<s> I am Sam </s>

<s> Sam I am </s>
<s> I am Sam </s>
<s> I do not like green eggs and Sam </s>

If we use linear interpolation smoothing between a maximum-likelihood bi gram model and a maximum-likelihood unigram model with $\lambda_1 = \frac{1}{2}$ and $\lambda_2 = \frac{1}{2}$, what is P(Sam|am)? Include <s> and </s> in your counts just like any other token.

3.8 Write a program to compute unsmoothed unigrams and bigrams.

3.9 Run your n-gram program on two different small corpora of your choice (you might use email text or newsgroups). Now compare the statistics of the two corpora. What are the differences in the most common unigrams between the two? How about interesting differences in bigrams?

3.10 Add an option to your program to generate random sentences.

3.11 Add an option to your program to compute the perplexity of a test set.

3.12 You are given a training set of 100 numbers that consists of 91 zeros and 1 each of the other digits 1-9. Now we see the following test set: 0 0 0 0 0 3 0 0 0 0. What is the unigram perplexity?

CHAPTER

4 Naive Bayes, Text Classification, and Sentiment

Classification lies at the heart of both human and machine intelligence. Deciding what letter, word, or image has been presented to our senses, recognizing faces or voices, sorting mail, assigning grades to homeworks; these are all examples of assigning a category to an input. The potential challenges of this task are highlighted by the fabulist Jorge Luis Borges (1964), who imagined classifying animals into:

(a) those that belong to the Emperor, (b) embalmed ones, (c) those that are trained, (d) suckling pigs, (e) mermaids, (f) fabulous ones, (g) stray dogs, (h) those that are included in this classification, (i) those that tremble as if they were mad, (j) innumerable ones, (k) those drawn with a very fine camel's hair brush, (l) others, (m) those that have just broken a flower vase, (n) those that resemble flies from a distance.

Many language processing tasks involve classification, although luckily our classes are much easier to define than those of Borges. In this chapter we introduce the naive Bayes algorithm and apply it to text categorization, the task of assigning a label or category to an entire text or document.

text categorization

We focus on one common text categorization task, sentiment analysis, the extraction of sentiment, the positive or negative orientation that a writer expresses toward some object. A review of a movie, book, or product on the web expresses the author's sentiment toward the product, while an editorial or political text expresses sentiment toward a candidate or political action. Extracting consumer

sentiment analysis

or public sen timent is thus relevant for fields from marketing to politics.

The simplest version of sentiment analysis is a binary classification task, and the words of the review provide excellent cues. Consider, for example, the follow ing phrases extracted from positive and negative reviews of movies and restaurants. Words like *great*, *richly*, *awesome*, and *pathetic*, and *awful* and *ridiculously* are very informative cues:

\+ *...zany characters and richly applied satire, and some great plot twists*
− *It was pathetic. The worst part about it was the boxing scenes...*
\+ *...awesome caramel sauce and sweet toasty almonds. I love this place!*
− *...awful pizza and ridiculously overpriced...*

spam detection Spam detection is another important commercial application, the binary clas sification task of assigning an email to one of the two classes *spam* or *not-spam*. Many lexical and other features can be used to perform this classification. For ex ample you might quite reasonably be suspicious of an email containing phrases like "online pharmaceutical" or "WITHOUT ANY COST" or "Dear Winner".

Another thing we might want to know about a text is the language it's written in. Texts on social media, for example, can be in any number of languages and
language id we'll need to apply different processing. The task of language id is thus the first step in most language processing pipelines. Related text classification tasks like au thorship attribution— determining a text's author— are also relevant to the digital
authorship attribution humanities, social sciences, and forensic linguistics.

classification is the task for which the naive Bayes algorithm was invented in 1961 Maron (1961).

Classification is essential for tasks below the level of the document as well. We've already seen period disambiguation (deciding if a period is the end of a sen tence or part of a word), and word tokenization (deciding if a character should be a word boundary). Even language modeling can be viewed as classification: each word can be thought of as a class, and so predicting the next word is classifying the context-so-far into a class for each next word. A part-of-speech tagger (Chapter 8) classifies each occurrence of a word in a sentence as, e.g., a noun or a verb.

The goal of classification is to take a single observation, extract some useful features, and thereby classify the observation into one of a set of discrete classes. One method for classifying text is to use rules handwritten by humans. Handwrit ten rule-based classifiers can be components of state-of-the-art systems in language processing. But rules can be fragile, as situations or data change over time, and for some tasks humans aren't necessarily good at coming up with the rules.

The most common way of doing text classification in language processing is instead via supervised machine learning, the subject of this chapter. In supervised

supervised machine

Finally, one of the oldest tasks in text classification is assigning a library sub ject category or topic label to a text. Deciding whether a research paper concerns epidemiology or instead, perhaps, embryology, is an important component of infor mation retrieval. Various sets of subject categories exist, such as the MeSH (Medical Subject Headings) thesaurus. In fact, as we will see, subject category

learning learning, we have a data set of input observations, each associated with some

correct output (a 'supervision signal'). The goal of the algorithm is to learn how to map from a new observation to a correct output.

Formally, the task of supervised classification is to take an input x and a fixed set of output classes $Y = \{y_1, y_2, ..., y_M\}$ and return a predicted class $y \in Y$. For text classification, we'll sometimes talk about c (for "class") instead of y as our output variable, and d (for "document") instead of x as our input variable. In the supervised situation we have a training set of N documents that have each been hand labeled with a class: $\{(d_1, c_1),...,(d_N, c_N)\}$. Our goal is to learn a classifier that is capable of mapping from a new document d to its correct class $c \in C$, where C is some set of useful document classes. A probabilistic classifier additionally will tell us the probability of the observation being in the class. This full distribution over the classes can be useful information for downstream decisions; avoiding making discrete decisions early on can be useful when combining systems.

Many kinds of machine learning algorithms are used to build classifiers. This chapter introduces naive Bayes; the following one introduces logistic regression. These exemplify two ways of doing classification. Generative classifiers like naive Bayes build a model of how a class could generate some input data. Given an ob servation, they return the class most likely to have generated the observation. Dis criminative classifiers like logistic regression instead learn what features from the input are most useful to discriminate between the different possible classes. While discriminative systems are often more accurate and hence more commonly used, generative classifiers still have a role.

4.1 Naive Bayes Classifiers

In this section we introduce the multinomial naive Bayes classifier, so called be- naive Bayes classifier cause it is a Bayesian classifier that makes a simplifying (naive) assumption about

how the features interact.

The intuition of the classifier is shown in Fig. 4.1. We represent a text document bag of words as if it were a bag of words, that is, an unordered set of words with their position ignored, keeping only their frequency in the document. In the example in the figure, instead of representing the word order in all the phrases like "I love this movie" and "I would recommend it", we simply note that the word *I* occurred 5 times in the entire excerpt, the word *it* 6 times, the words *love*, *recommend*, and *movie* once, and so on.

it
6
I
5
the
4
I love this movie! It's sweet,
fairy
it
to
3
always
love
but with satirical humor. The
to
it
and
3
whimsical
it
dialogue is great and the
I
and
are anyone
seen
seen
2
adventure scenes are fun...
friend
dialogue
yet
1
happy
It manages to be whimsical
recommend
would
1
adventure
and romantic while laughing
satirical
whimsical
1
sweet
of
at the conventions of the
who
it
movie
I
to
times
1
fairy tale genre. I would
it
but
romantic
I
yet
sweet
1
several
recommend it to just about
humor
again
the
satirical
1
it
anyone. I've seen it several
the

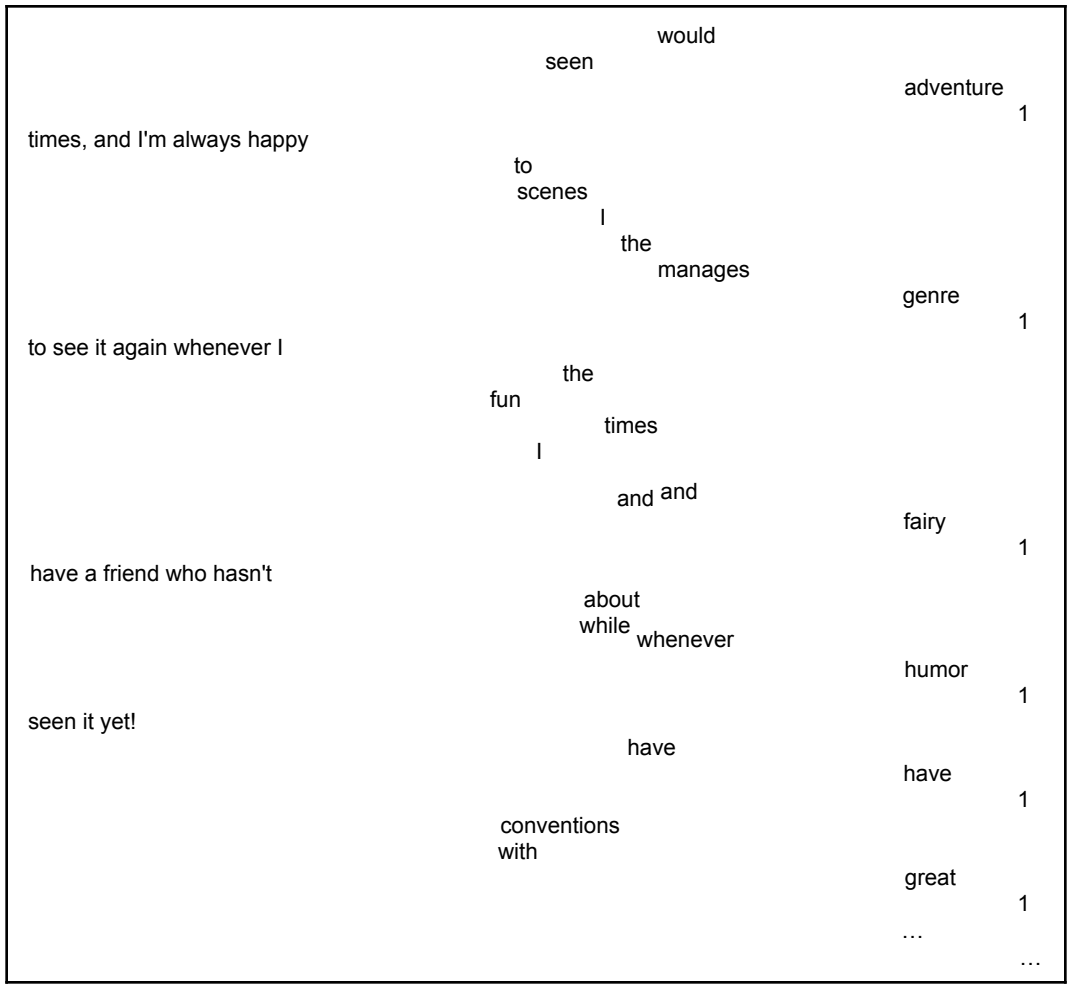


Figure 4.1 Intuition of the multinomial naive Bayes classifier applied to a movie review. The position of the words is ignored (the *bag-of-words* assumption) and we make use of the frequency of each word.

Naive Bayes is a probabilistic classifier, meaning that for a document d, out of all classes $c \in C$ the classifier returns the class $\hat{c}$ which has the maximum posterior probability given the document. In Eq. 4.1 we use the hat notation ˆ to mean "our estimate of the correct class".

$$\hat{c} = \operatorname{argmax}_{c \in C} P(c|d) \quad (4.1)$$

This idea of Bayesian inference has been known since the work of Bayes (1763), Bayesian inference and was first applied to text classification by Mosteller and Wallace (1964). The intuition of Bayesian classification is to use Bayes' rule to transform Eq. 4.1 into other probabilities that have some useful properties. Bayes' rule is presented in Eq. 4.2; it gives us a way to break down any conditional probability $P(x|y)$ into three other probabilities:

$$P(x|y) = \frac{P(y|x)P(x)}{P(y)} \quad (4.2)$$

We can then substitute Eq. 4.2 into Eq. 4.1 to get Eq. 4.3:

$$\hat{c} = \operatorname{argmax}_{c \in C} P(c|d) = \operatorname{argmax}_{c \in C} \frac{P(d|c)P(c)}{P(d)} \quad (4.3)$$

We can conveniently simplify Eq. 4.3 by dropping the denominator $P(d)$. This is possible because we will be computing $\frac{P(d|c)P(c)}{P(d)}$ for each possible class. But $P(d)$ doesn't change for each class; we are always asking about the most likely class for the same document d, which must have the same probability $P(d)$. Thus, we can choose the class that maximizes this simpler formula:

$$\hat{c} = \operatorname*{argmax}_{c \in C} P(c|d) = \operatorname*{argmax}_{c \in C} P(d|c)P(c) \quad (4.4)$$

Chapter 5.

To return to classification: we compute the most probable class $\hat{c}$ given some document d by choosing the class which has the highest product of two probabilities: the prior probability of the class $P(c)$ and the likelihood of the document $P(d|c)$:

prior probability likelihood

We call Naive Bayes a generative model because we can read Eq. 4.4 as stating a kind of implicit assumption about how a document is generated: first a class is sampled from $P(c)$, and then the words are generated by sampling from $P(d|c)$. (In fact we could imagine generating artificial documents, or at least their word counts, by following this process). We'll say more about this intuition of generative models in

prior likelihood

$$\hat{c} = \operatorname*{argmax}_{c \in C} \overbrace{P(d|c)}^{\text{likelihood}} \overbrace{P(c)}^{\text{prior}} \quad (4.5)$$

Without loss of generalization, we can represent a document d as a set of features $f_1, f_2, ..., f_n$:

$$\hat{c} = \operatorname*{argmax}_{c \in C} \overbrace{P(f_1, f_2, ..., f_n|c)}^{\text{likelihood}} \overbrace{P(c)}^{\text{prior}} \quad (4.6)$$

Unfortunately, Eq. 4.6 is still too hard to compute directly: without some simplifying assumptions, estimating the probability of every possible combination of features (for example, every possible set of words and positions) would require huge numbers of parameters and impossibly large training sets. Naive Bayes classifiers therefore make two simplifying assumptions.

The first is the *bag-of-words* assumption discussed intuitively above: we assume position doesn't matter, and that the word "love" has the same effect on classification whether it occurs as the 1st, 20th, or last word in the document. Thus we assume that the features $f_1, f_2, ..., f_n$ only encode word identity and not position.

naive Bayes assumption

The second is commonly called the naive Bayes assumption: this is the conditional independence assumption that the probabilities $P(f_i|c)$ are independent given the class c and hence can be 'naively' multiplied as follows:

$$P(f_1, f_2, ..., f_n|c) = P(f_1|c) \cdot P(f_2|c) \cdot ... \cdot P(f_n|c) \quad (4.7)$$

The final equation for the class chosen by a naive Bayes classifier is thus:

$$c_{NB} = \text{argmax}_{c \in C} P(c) \prod_{f \in F} P(f|c) \quad (4.8)$$

To apply the naive Bayes classifier to text, we need to consider word positions, by simply walking an index through every word position in the document:

positions ← all word positions in test document

$$c_{NB} = \text{argmax}_{c \in C} P(c) \prod_{i \in positions} P(w_i|c) \quad (4.9)$$

Naive Bayes calculations, like calculations for language modeling, are done in log space, to avoid underflow and increase speed. Thus Eq. 4.9 is generally instead expressed[1] as

$$c_{NB} = \text{argmax}_{c \in C} \log P(c) + \sum_{i \in positions} \log P(w_i|c) \quad (4.10)$$

4.10 computes the predicted class as a linear function of input features. Classifiers that use a linear combination of the inputs to make a classification decision —like naive Bayes and also logistic regression— are called linear classifiers.

linear classifiers

By considering features in log space, Eq.

4.2 Training the Naive Bayes Classifier

How can we learn the probabilities $P(c)$ and $P(f_i|c)$? Let's first consider the maximum likelihood estimate. We'll simply use the frequencies in the data. For the class prior $P(c)$ we ask what percentage of the documents in our training set are in each class c. Let N_c be the number of documents in our training data with class c and N_{doc} be the total number of documents. Then:

$$\hat{P}(c) = \frac{N_c}{N_{doc}} \quad (4.11)$$

To learn the probability $P(f_i|c)$, we'll assume a feature is just the existence of a word in the document's bag of words, and so we'll want $P(w_i|c)$, which we compute as the fraction of times the word w_i appears among all words in all documents of topic c. We first concatenate all documents with category c into one big "category c" text. Then we use the frequency of w_i in this concatenated document to give a maximum likelihood estimate of the probability:

$$\hat{P}(w_i|c) = \frac{count(w_i, c)}{\sum_{w \in V} count(w, c)} \quad (4.12)$$

Here the vocabulary V consists of the union of all the word types in all classes, not just the words in one class c.

There is a problem, however, with maximum likelihood training. Imagine we are trying to estimate the likelihood of the word "fantastic" given class *positive*, but suppose there are no training documents that both contain the word "fantastic" and are classified as *positive*. Perhaps the word "fantastic"

happens to occur (sarcasti cally?) in the class *negative*. In such a case the probability for this feature will be zero:

$$\hat{P}(\text{“fantastic”}|\text{positive}) = \frac{count(\text{“fantastic”},\text{positive})}{\sum_{w \in V} count(w,\text{positive})} = 0 \quad (4.13)$$

But since naive Bayes naively multiplies all the feature likelihoods together, zero probabilities in the likelihood term for any class will cause the probability of the class to be zero, no matter the other evidence!

The simplest solution is the add-one (Laplace) smoothing introduced in Chap ter 3. While Laplace smoothing is usually replaced by more sophisticated smoothing

[1] In practice throughout this book, we'll use log to mean natural log (ln) when the base is not specified.

algorithms in language modeling, it is commonly used in naive Bayes text catego rization:

$$\hat{P}(w_i|c) = \frac{count(w_i, c) + 1}{\sum_{w \in V}(count(w, c) + 1) = count(w_i, c) + 1} = \frac{count(w_i, c) + 1}{\sum_{w \in V} count(w, c) + |V|} \quad (4.14)$$

Note once again that it is crucial that the vocabulary V consists of the union of all the word types in all classes, not just the words in one class *c* (try to convince yourself why this must be true; see the exercise at the end of the chapter).

What do we do about words that occur in our test data but are not in our vocab ulary at all because they did not occur in any training document in any class? The unknown word solution for such unknown words is to ignore them—remove them from the test document and not include any probability for them at all.

Finally, some systems choose to completely ignore another class of words: stop stop words words, very frequent words like *the* and *a*. This can be done by sorting the vocabu lary by frequency in the training set, and defining the top 10–100 vocabulary entries as stop words, or alternatively by using one of the many predefined stop word lists available online. Then each instance of these stop words is simply removed from both training and test documents as if it had never occurred. In most text classifica tion applications, however, using a stop word list doesn't improve performance, and so it is more common to make use of the entire vocabulary and not use a stop word list.

Fig. 4.2 shows the final algorithm.

function TRAIN NAIVE BAYES(D, C) returns log $P(c)$ and log $P(w|c)$

for each class $c \in C$ # Calculate $P(c)$ terms
  N_{doc} = number of documents in D
  N_c = number of documents from D in class c
  $logprior[c] \leftarrow \log \frac{N_c}{N_{doc}}$
  $V \leftarrow$ vocabulary of D
  $bigdoc[c] \leftarrow$ append(d) for d $\in$ D with class c
  for each word w in V # Calculate $P(w|c)$ terms
    $count(w,c) \leftarrow$ # of occurrences of w in $bigdoc[c]$
    $loglikelihood[w,c] \leftarrow \log \frac{count(w, c) + 1}{\sum_{w' \text{ in } V} (count(w', c) + 1)}$
return *logprior*, *loglikelihood*, *V*

function TEST NAIVE BAYES(*testdoc*, *logprior*, *loglikelihood*, C, V) returns best *c*

for each class $c \in C$
  $sum[c] \leftarrow logprior[c]$
  for each position i in *testdoc*
    $word \leftarrow testdoc[i]$
    if $word \in V$
      $sum[c] \leftarrow sum[c] + loglikelihood[word,c]$
return $\text{argmax}_c sum[c]$

Figure 4.2 The naive Bayes algorithm, using add-1 smoothing. To use add-α smoothing instead, change the +1 to +α for loglikelihood counts in training.

4.3 Worked example

Let's walk through an example of training and testing naive Bayes with add-one smoothing. We'll use a sentiment analysis domain with the two classes positive (+) and negative (-), and take the following miniature training and test documents simplified from actual movie reviews.

	Cat Documents
Training	- just plain boring
	- entirely predictable and lacks energy
	- no surprises and very few laughs
	+ very powerful
	+ the most fun film of the summer
Test	? predictable with no fun

The prior $P(c)$ for the two classes is computed via Eq. 4.11 as $\frac{N_c}{N_{doc}}$:

$$P(-) = \frac{3}{5} \quad P(+) = \frac{2}{5}$$

The word *with* doesn't occur in the training set, so we drop it completely (as mentioned above, we don't use unknown word models for naive Bayes). The like lihoods from the training set for the remaining three words "predictable", "no", and "fun", are as follows, from Eq. 4.14 (computing the

probabilities for the remainder of the words in the training set is left as an exercise for the reader):

$$P(\text{“predictable”}|-) = \frac{1+1}{14+20} \qquad P(\text{“predictable”}|+) = \frac{0+1}{9+20}$$

$$P(\text{“no”}|-) = \frac{1+1}{14+20} \qquad P(\text{“no”}|+) = \frac{0+1}{9+20}$$

$$P(\text{“fun”}|-) = \frac{0+1}{14+20} \qquad P(\text{“fun”}|+) = \frac{1+1}{9+20}$$

For the test sentence S = “predictable with no fun”, after removing the word ‘with’, the chosen class, via Eq. 4.9, is therefore computed as follows:

$$P(-)P(S|-) = \frac{3}{5} \times \frac{2\times2\times1}{34^3} = 6.1\times10^{-5}$$

$$P(+)P(S|+) = \frac{2}{5} \times \frac{1\times1\times2}{29^3} = 3.2\times10^{-5}$$

The model thus predicts the class *negative* for the test sentence.

4.4 Optimizing for Sentiment Analysis

While standard naive Bayes text classification can work well for sentiment analysis, some small changes are generally employed that improve performance.

First, for sentiment classification and a number of other text classification tasks, whether a word occurs or not seems to matter more than its frequency. Thus it often improves performance to clip the word counts in each document at 1 (see the end

of the chapter for pointers to these results). This variant is called binary multino mial naive Bayes or binary naive Bayes. The variant uses the same algorithm as binary naive Bayes in Fig. 4.2 except that for each document we remove all duplicate words before con catenating them into the single big document during training and we also remove duplicate words from test documents. Fig. 4.3 shows an example in which a set of four documents (shortened and text-normalized for this example) are remapped to binary, with the modified counts shown in the table on the right. The example is worked without add-1 smoothing to make the differences clearer. Note that the results counts need not be 1; the word *great* has a count of 2 even for binary naive Bayes, because it appears in multiple documents.

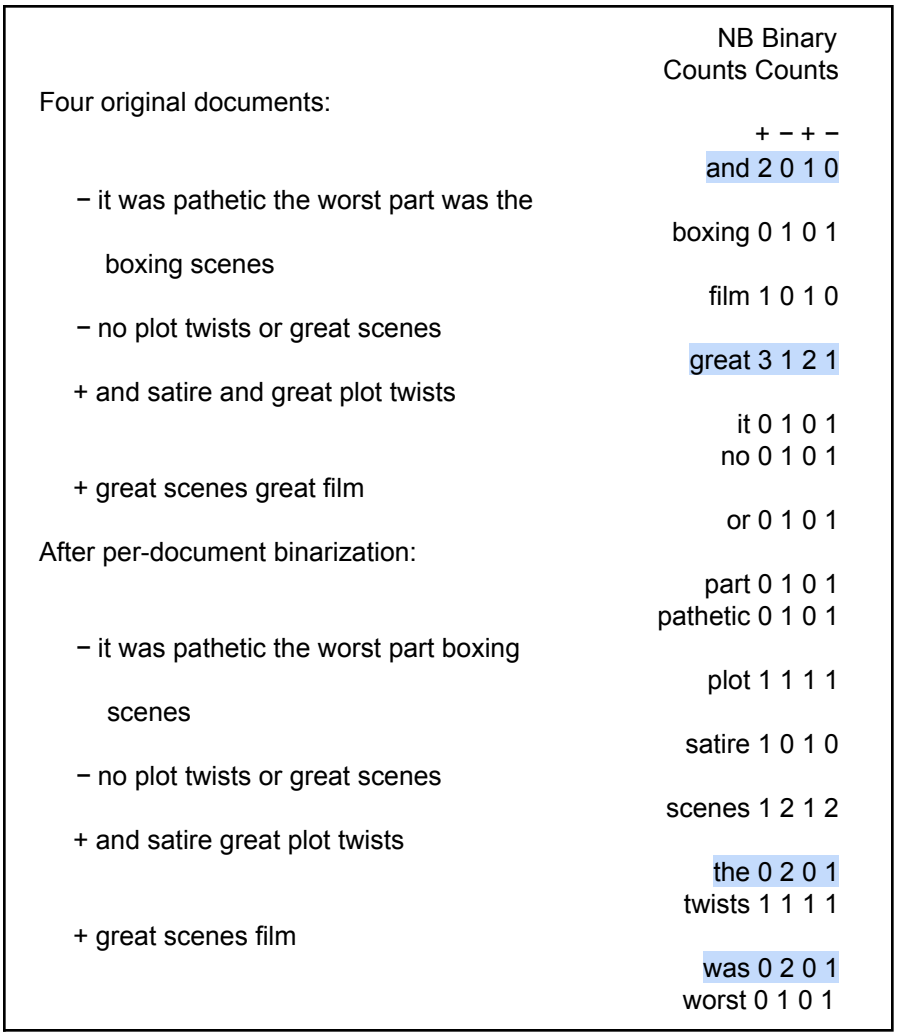


Figure 4.3 An example of binarization for the binary naive Bayes algorithm.

A second important addition commonly made when doing text classification for sentiment is to deal with negation. Consider the difference between *I really like this movie* (positive) and *I didn't like this movie* (negative). The negation expressed by *didn't* completely alters the inferences we draw from the predicate *like*. Similarly, negation can modify a negative word to produce a positive review (*don't dismiss this film*, *doesn't let us get bored*).

A very simple baseline that is commonly used in sentiment analysis to deal with negation is the following: during text normalization, prepend the prefix *NOT* to every word after a token of logical negation (*n't, not, no, never*) until the next punc tuation mark. Thus the phrase

didn't like this movie , but I

becomes

didn't NOT_like NOT_this NOT_movie , but I

Newly formed 'words' like *NOT like*, *NOT recommend* will thus occur more of ten in negative document and act as cues for negative sentiment, while words like *NOT bored*, *NOT dismiss* will acquire positive associations. We will return in Chap ter 20 to the use of parsing to deal more accurately with the scope relationship be tween these negation words and the predicates they modify, but this simple baseline works quite well in practice.

Finally, in some situations we might have insufficient labeled training data to train accurate naive Bayes classifiers using all words in the training set to estimate positive and negative sentiment. In such cases we can instead derive the positive and negative word features from sentiment lexicons, lists of words that are pre-annotated with positive or negative sentiment. Four popular lexicons are the General Inquirer (Stone et al., 1966), LIWC (Pennebaker et al., 2007), the opinion lexicon of Hu and Liu (2004a) and the MPQA Subjectivity Lexicon (Wilson et al., 2005). For example the MPQA subjectivity lexicon has 6885 words each marked for whether it is strongly or weakly biased positive or negative. Some examples:

sentiment lexicons
General Inquirer
LIWC

+ : *admirable, beautiful, confident, dazzling, ecstatic, favor, glee, great*
− : *awful, bad, bias, catastrophe, cheat, deny, envious, foul, harsh, hate*

A common way to use lexicons in a naive Bayes classifier is to add a feature that is counted whenever a word from that lexicon occurs. Thus we might add a feature called 'this word occurs in the positive lexicon', and treat all instances of words in the lexicon as counts for that one feature, instead of counting each word separately. Similarly, we might add as a second feature 'this word occurs in the negative lexicon' of words in the negative lexicon. If we have lots of training data, and if the test data matches the training data, using just two features won't work as well as using all the words. But when training data is sparse or not representative of the test set, using dense lexicon features instead of sparse individual-word features may generalize better.

We'll return to this use of lexicons in Chapter 25, showing how these lexicons can be learned automatically, and how they can be applied to many other tasks be yond sentiment classification.

4.5 Naive Bayes for other text classification tasks

In the previous section we pointed out that naive Bayes doesn't require that our classifier use all the words in the training data as features. In fact features in naive Bayes can express any property of the input text we want.

spam detection

Consider the task of spam detection, deciding if a particular piece of email is an example of spam (unsolicited bulk email)—one of the first applications of naive Bayes to text classification (Sahami et al., 1998).

A common solution here, rather than using all the words as individual features, is to predefine likely sets of words or phrases as features, combined with features that are not purely linguistic. For example the open-source SpamAssassin tool[2] predefines features like the phrase "one hundred percent guaranteed", or the feature *mentions millions of dollars*, which is a regular expression that matches suspiciously large sums of money. But it also includes features like *HTML has a low ratio of text to image area*, that aren't purely linguistic and might require some sophisticated computation, or totally non-linguistic features about, say, the path that the email took to arrive. More sample SpamAssassin features:

- Email subject line is all capital letters
- Contains phrases of urgency like "urgent reply"
- Email subject line contains "online pharmaceutical"
- HTML has unbalanced "head" tags

[2] https://spamassassin.apache.org

• Claims you can be removed from the list

language id For other tasks, like language id—determining what language a given piece of text is written in—the most effective naive Bayes features are not words at all, but character n-grams, 2-grams ('zw') 3-grams ('nya', ' Vo'), or 4-grams ('ie z', 'thei'), or, even simpler byte n-grams, where instead of using the multibyte Unicode character representations called codepoints, we just pretend everything is a string of raw bytes. Because spaces count as a byte, byte n-grams can model statistics about the beginning or ending of words. A widely used naive Bayes system, langid.py (Lui and Baldwin, 2012) begins with all possible n-grams of lengths 1-4, using fea ture selection to winnow down to the most informative 7000 final features.

Language ID systems are trained on multilingual text, such as Wikipedia (Wiki pedia text in 68 different languages was used in (Lui and Baldwin, 2011)), or newswire. To make sure that this multilingual text correctly reflects different regions, dialects, and socioeconomic classes, systems also add Twitter text in many languages geo tagged to many regions (important for getting world English dialects from countries with large Anglophone populations like Nigeria or India), Bible and Quran transla tions, slang websites like Urban Dictionary, corpora of African American Vernacular English (Blodgett et al., 2016), and so on (Jurgens et al., 2017).

4.6 Naive Bayes as a Language Model

As we saw in the previous section, naive Bayes classifiers can use any sort of fea ture: dictionaries, URLs, email addresses, network features, phrases, and so on. But if, as in the previous section, we use only individual word features, and we use all of the words in the text (not a subset), then naive Bayes has an important similar ity to language modeling. Specifically, a naive Bayes model can be viewed as a set of class-specific unigram language models, in which the model for each class instantiates a unigram language model.

Since the likelihood features from the naive Bayes model assign a probability to each word $P(word|c)$, the model also assigns a probability to each sentence:

$$P(s|c) = \prod_{i \in positions} P(w_i|c) \quad (4.15)$$

Thus consider a naive Bayes model with the classes *positive* (+) and *negative* (-) and the following model parameters:

w	P(w|+)	P(w|-)
I	0.1	0.2
love	0.1	0.001
this	0.01	0.01
fun	0.05	0.005
film	0.1	0.1
...	...	...

Each of the two columns above instantiates a language model that can assign a probability to the sentence "I love this fun film":

P("I love this fun film"|+) = 0.1×0.1×0.01×0.05×0.1 = 0.0000005 P("I love this fun film"|−) = 0.2×0.001×0.01×0.005×0.1 = .0000000010

As it happens, the positive model assigns a higher probability to the sentence: $P(s|pos) > P(s|neg)$. Note that this is just the likelihood part of the naive Bayes model; once we multiply in the prior a full naive Bayes model might well make a different classification decision.

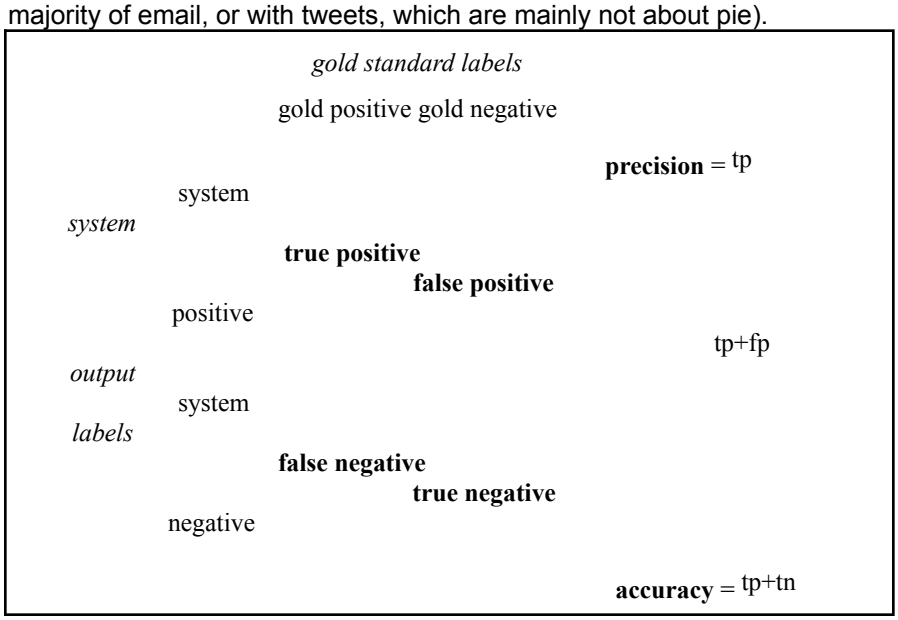
4.7 Evaluation: Precision, Recall, F-measure

To introduce the methods for evaluating text classification, let's first consider some simple binary *detection* tasks. For example, in spam detection, our goal is to label every text as being in the spam category ("positive") or not in the spam category ("negative"). For each item (email document) we therefore need to know whether our system called it spam or not. We also need to know whether the email is actually spam or not, i.e. the human-defined labels for each document that we are trying to gold labels match. We will refer to these human labels as the gold labels.

Or imagine you're the CEO of the *Delicious Pie Company* and you need to know what people are saying about your pies on social media, so you build a system that detects tweets concerning Delicious Pie. Here the positive class is tweets about Delicious Pie and the negative class is all other tweets.

In both cases, we need a metric for knowing how well our spam detector (or pie-tweet-detector) is doing. To evaluate any system for detecting things, we start by building a confusion matrix like the one shown in Fig. 4.4. A confusion matrix confusion matrix is a table for visualizing how an algorithm performs with respect to the human gold labels, using two dimensions (system output and gold labels), and each cell labeling a set of possible outcomes. In the spam detection case, for example, true positives are documents that are indeed spam (indicated by human-created gold labels) that our system correctly said were spam. False negatives are documents that are indeed spam but our system incorrectly labeled as non-spam.

To the bottom right of the table is the equation for *accuracy*, which asks what percentage of all the observations (for the spam or pie examples that means all emails or tweets) our system labeled correctly. Although accuracy might seem a natural metric, we generally don't use it for text classification tasks. That's because accuracy doesn't work well when the classes are unbalanced (as indeed they are with spam, which is a large majority of email, or with tweets, which are mainly not about pie).



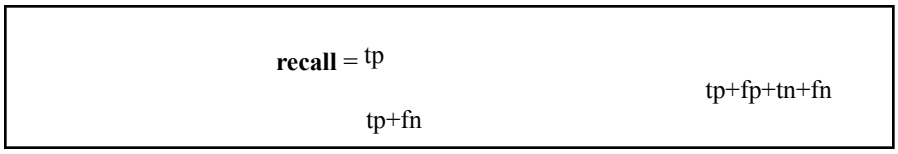


Figure 4.4 A confusion matrix for visualizing how well a binary classification system per forms against gold standard labels.

To make this more explicit, imagine that we looked at a million tweets, and let's say that only 100 of them are discussing their love (or hatred) for our pie,

while the other 999,900 are tweets about something completely unrelated. Imagine a simple classifier that stupidly classified every tweet as "not about pie". This classifier would have 999,900 true negatives and only 100 false negatives for an accuracy of 999,900/1,000,000 or 99.99%! What an amazing accuracy level! Surely we should be happy with this classifier? But of course this fabulous 'no pie' classifier would be completely useless, since it wouldn't find a single one of the customer comments we are looking for. In other words, accuracy is not a good metric when the goal is to discover something that is rare, or at least not completely balanced in frequency, which is a very common situation in the world.

That's why instead of accuracy we generally turn to two other metrics shown in Fig. 4.4: precision and recall. Precision measures the percentage of the items that the system detected (i.e., the system labeled as positive) that are in fact positive (i.e., are positive according to the human gold labels). Precision is defined as

precision

$$\text{Precision} = \frac{\text{true positives}}{\text{true positives + false positives}}$$

Recall measures the percentage of items actually present in the input that were correctly identified by the system. Recall is defined as

recall

$$\text{Recall} = \frac{\text{true positives}}{\text{true positives + false negatives}}$$

Precision and recall will help solve the problem with the useless "nothing is pie" classifier. This classifier, despite having a fabulous accuracy of 99.99%, has a terrible recall of 0 (since there are no true positives, and 100 false negatives, the recall is 0/100). You should convince yourself that the precision at finding relevant tweets is equally problematic. Thus precision and recall, unlike accuracy, emphasize true positives: finding the things that we are supposed to be looking for.

There are many ways to define a single metric that incorporates aspects of both precision and recall. The simplest of these combinations is the F-measure (van Rijsbergen, 1975) , defined as:

F-measure

$$F_\beta = \frac{(\beta^2+1)PR}{\beta^2 P + R}$$

The β parameter differentially weights the importance of recall and precision, based perhaps on the needs of an application. Values of $\beta > 1$ favor recall, while values of $\beta < 1$ favor precision. When $\beta = 1$, precision and recall are equally bal

F1

anced; this is the most frequently used metric, and is called $F_{\beta=1}$ or just F_1:

$$F_1 = \frac{2PR}{P+R} \quad (4.16)$$

F-measure comes from a weighted harmonic mean of precision and recall. The harmonic mean of a set of numbers is the reciprocal of the arithmetic mean of recip rocals:

$$\text{HarmonicMean}(a_1, a_2, a_3, a_4, \ldots, a_n) = \frac{n}{\frac{1}{a_1}+\frac{1}{a_2}+\frac{1}{a_3}+\ldots+\frac{1}{a_n}} \quad (4.17)$$

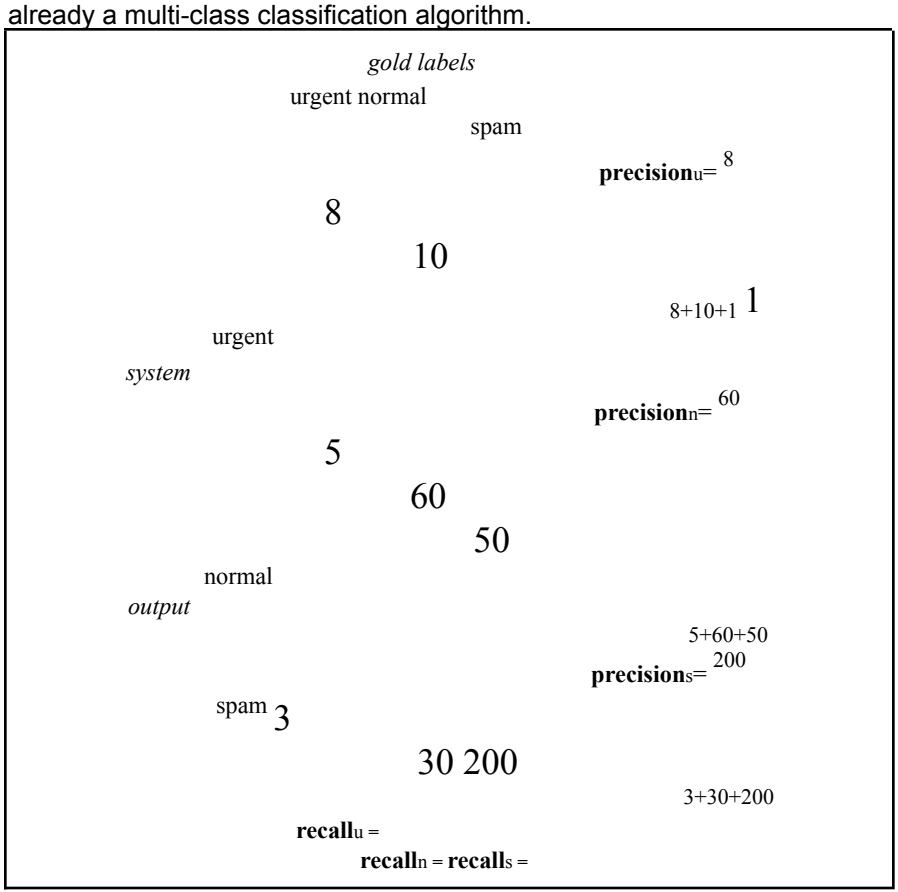
and hence F-measure is $F = \frac{1}{\alpha\frac{1}{P} + (1-\alpha)\frac{1}{R}}$ with $\beta^2 = \frac{1-\alpha}{\alpha}$ or

$$F = \frac{(\beta^2+1)PR}{\beta^2 P+R} \quad (4.18)$$

Harmonic mean is used because the harmonic mean of two values is closer to the minimum of the two values than the arithmetic mean is. Thus it weighs the lower of the two numbers more heavily, which is more conservative in this situation.

4.7.1 Evaluating with more than two classes

Up to now we have been describing text classification tasks with only two classes. But lots of classification tasks in language processing have more than two classes. For sentiment analysis we generally have 3 classes (positive, negative, neutral) and even more classes are common for tasks like part-of-speech tagging, word sense disambiguation, semantic role labeling, emotion detection, and so on. Luckily the naive Bayes algorithm is already a multi-class classification algorithm.



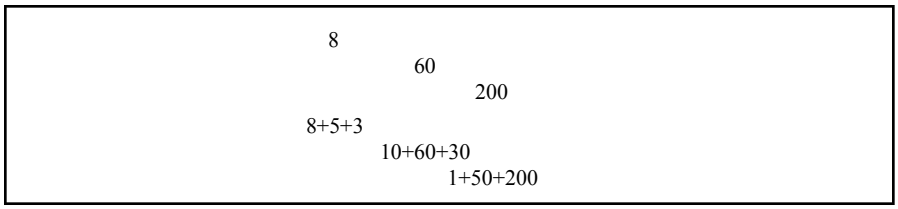


Figure 4.5 Confusion matrix for a three-class categorization task, showing for each pair of classes (c_1, c_2), how many documents from c_1 were (in)correctly assigned to c_2.

But we'll need to slightly modify our definitions of precision and recall. Con sider the sample confusion matrix for a hypothetical 3-way *one-of* email catego rization decision (urgent, normal, spam) shown in Fig. 4.5. The matrix shows, for example, that the system mistakenly labeled one spam document as urgent, and we have shown how to compute a distinct precision and recall value for each class. In order to derive a single metric that tells us how well the system is doing, we can com

macroaveraging bine these values in two ways. In macroaveraging, we compute the performance microaveraging for each class, and then average over classes. In microaveraging, we collect the de cisions for all classes into a single confusion matrix, and then compute precision and recall from that table. Fig. 4.6 shows the confusion matrix for each class separately, and shows the computation of microaveraged and macroaveraged precision. As the figure shows, a microaverage is dominated by the more frequent class (in this case spam), since the counts are pooled. The macroaverage better reflects the statistics of the smaller classes, and so is more appropriate when performance on all the classes is equally important.

4.8 Test sets and Cross-validation

The training and testing procedure for text classification follows what we saw with language modeling (Section 3.2): we use the training set to train the model, then use the development test set (also called a devset) to perhaps tune some parameters,

development

test set
devset