2 正则表达式，分词，编辑距离

> 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.

上面的对话来自一个早期的自然语言处理系统ELIZA，它能够模仿罗杰斯派心理治疗师（Rogerian psychotherapist）与用户进行有限的对话（Weizenbaum, 1966）。ELIZA是一个出人意料的简单程序，它使用模式匹配来识别诸如“我需要xxx”样这样的短语，并将其转化为合适的输出，比如“如果你得到了X，这对你意味着什么？” 这种简单的技术在人机对话领域取得了一定成功，因为ELIZA实际上并不需要了解任何事情就能模仿Rogerian psychotherapist。正如Weizenbaum指出的那样，这是少数几种对话体裁之一，听者可以表现得仿佛他们对世界一无所知。ELIZA对人类对话的模仿非常成功：许多与ELIZA互动的人认为它真的理解他们和他们的问题，即使在程序的运作方式被解释之后，许多人仍然继续相信ELIZA有这样的能力（Weizenbaum, 1976）。甚至直到今天，这类聊天机器人仍然是一种有趣的消遣。

当然，现代对话智能体远不止是一种消遣；它们可以回答问题、预订航班或查找餐厅。这些功能依赖于对用户意图更为复杂的理解。尽管如此，驱动ELIZA和其他聊天机器人的**简单模式匹配方法**在自然语言处理领域仍然发挥着至关重要的作用。

我们将从描述文本模式最重要的工具——**正则表达式**开始。正则表达式可以用来指定我们希望从文档中提取的字符串，例如在ELIZA系统中的对“我需要X”字符串的转换，或定义
像“$199”或“$24.99”这样的字符串以从文档中提取价格表。

接下来，我们将介绍**文本规范化任务**，其中正则表达式起着重要作用。文本规范化意味着将文本转换为更方便、标准的形式。例如，我们处理语言的大多数操作都依赖于首先将连贯的文本拆分成单独的词，称为**分词（tokenization）**。英语单词通常通过空格彼此分开，但仅仅关注空格是不够的。例如，虽然*New York*和 *rock 'n' roll* 包含空格，但有时需要将其视为一个整体。相反的，有时我们需要将 *I'm* 拆分成两个词 *I* 和 *am*。处理推文（tweets）或短信时，我们则需要将表情符号（如 *:)*）或标

签（如 *#nlproc*）处理为单独的词元（token）。除英语之外，一些语言（如中文）单词之间没有空格，这将使分词变得更加困难。

文本规范化的另一部分是**词形还原（lemmatization）**——确定两个词是否具有相同词根，尽管它们在表面上有所不同。例如，单词 *sang*、*sung*和 *sings* 都是动词 *sing* 的不同形式，但*sing*是这些词的共同词形，那么词形还原器会将这些词都映射为 *sing*。词形还原对于处理像阿拉伯语这样形态复杂的语言至关重要。**词干提取（stemming）**是词形还原的一种简化形式，其主要是去掉词尾的后缀。文本规范化还包括句子分割：根据句号或感叹号等标志将文本分割成单独的句子。

最后，我们还需要比较单词和其他字符串。我们将介绍一种称为**编辑距离（edit distance）**的度量标准，它根据将一个字符串转换为另一个字符串所需的编辑次数（插入、删除、替换）来衡量两个字符串的相似程度。编辑距离是一种广泛应用于从拼写纠正到语音识别，再到指代消解等语言处理任务的算法。

2.1 正则表达式

在计算机科学中，正则表达式（regular expression，通常缩写为regex）是最有用的文本处理工具之一，它是一种用于指定文本搜索字符串的语言。这种实用的语言在每种计算机语言中都有使用，在文本处理工具（如Unix工具中的grep）以及编辑器（如vim或Emacs）中都能见到。形式上，正则表达式是一种用于描述一组字符串的代数符号。正则表达式在进行文本搜索时特别有用，当我们有一个需要搜索的模式和一组待搜索的文本时，正则表达式搜索功能会遍历整个文本集，返回与该模式匹配的所有文本。这个文本集可以是单个文档或一组文档。例如，Unix命令行工具grep接收一个正则表达式，并返回输入文档中所有与表达式匹配的行。

搜索可以设计为返回每行中的所有匹配项（如果有多个匹配项），或者只返回第一个匹配项。在下面的示例中，我们通常会强调与正则表达式匹配的模式的确切部分，并且只展示第一个匹配。我们将用斜线来表示正则表达式的界限（但请注意，**斜线本身并不是正则表达式的一部分**）。

我们将介绍扩展正则表达式（extended regular expressions）。由于正则表达式有很多变体，不同的正则表达式解析器可能只识别扩展正则表达式的子集，或者对某些表达式有稍微不同的处理方式。另外，使用在线正则表达式测试器是测试表达式和探索这些变体的一种方便方法。

2.1.1 基本的正则表达式模式

最简单的正则表达式是由一串简单字符组成的序列，将字符按顺序排列称为**连接**。要搜索 *woodchuck*，我们可以输入 **/woodchuck/**。表达式 **/Buttercup/** 匹配任何包含子字符串*Buttercup* 的字符串，例如在grep中使用该表达式会返回包含*I’m called little Buttercup*。搜索字符串可以是单个字符（如 **/!/**）或一串字符（如 **/urgl/**）（见图2.1）。

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

图 2.1 一些简单的正则表达式搜索

正则表达式是**区分大小写**的，**/s/**与**/S/**不同（/s/ 匹配小写*s*，但不匹配大写*S*）。这意味着模式 **/woodchucks/** 将不会匹配字符串*Woodchucks*。我们可以通过使用方括号 **[ ]** 来解决这个问题。方括号内的字符表示要匹配的字符的“或”关系。例如，图2.2显示了模式 **/[wW]/** 可以匹配包含*w*或*W*的字符串。

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”

图 2.2 使用方括号 [] 来对字符串表示析取

正则表达式 **/[1234567890]/** 用于匹配任意一个数字字符。虽然数字或字母等字符类是表达式中的重要构建块，但它们有时会显得繁琐或很不方便（例如，使用 **/[ABCDEFGHIJKLMNOPQRSTUVWXYZ]/** 来表示任意大写字母）。在字符集有明确的顺序时，可以使用方括号和短划线 **-** 来指定一个范围内的任意字符。模式 **/[2-5]/** 表示任意一个字符2、3、4或5。模式 **/[b-g]/** 表示字符 b、c、d、e、f 或 g 中的一个。图2.3中展示了其他一些示例。

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”

图 2.3 使用方括号 [] 和短划线 - 来表示一个范围

方括号还可以用来指定某个字符不匹配的内容，通过使用插入符号**^**。如果插入符号**^**是左方括号后的第一个符号，那么生成的模式就是取反的。例如，模式 **/[^a]/** 匹配任意一个除去 *a*的字符（包括特殊字符）。只有当插入符号是左方括号后的第一个符号时，这种情况才成立。如果插入符号出现在其他地方，它通常表示插入符号本身。图2.4展示了一些示例。

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

图 2.4 插入符号 ^ 表示取反与自我表示

我们如何处理可选元素，比如 **woodchuck** 和 **woodchucks** 中的可选 *s* 呢？我们不能使用方括号，因为方括号表达“s 或 S”，但不能表达“s 或 没有”。为此我们使用问号 **/?/**，它表示“有该问号前面的一个表达式或没有”，如图2.5所示。

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

图 2.5？ 标记前一个表达式的可选性。

我们可以将问号理解为“前一个字符出现零次或一次”。也就是说，它是一种指定我们需要某个字符出现次数的方式，而这一特性这在正则表达式中非常重要。例如，考虑某些绵羊的语言，其字符串看起来如下：

baa!、

baaa!、

baaaa!

……

该绵羊语言的字符串由一个 *b* 开头，后面跟着至少两个 *a*，再接一个感叹号。允许我们表达“某个数量的a”的操作符为 *，通常称为克林星号（Kleene star，通一般发音为“cleany star”）。* 的意思是“前一个字符或表达式出现零次或多次”。因此 **/a*/** 意味着“任意数量的a（包括零次）”，匹配的结果可为 *a* 或 *aaaaaa*，但它也会匹配空字符串，因为字符串 *Off Minor* 以零个 *a* 开头。因此，匹配一个或多个 *a* 的正则表达式是 **/aa*/**，意思是一个 *a*后跟零个或多个 *a*。而且该操作符还可以重复更复杂的表达式而不仅仅是一个字符。**/[ab]*/**表示“零个或多个 *a* 或 *b*”（切记不是“零个或多个右方括号”）。这可以匹配到 *aaaa*、*ababab* 或 *bbbb* 之类的字符串，也会匹配空字符串。

为了指定多个数字（例如在查找价格时），我们可以扩展表示单个数字的正则表达式 **/[0-9]/**。一个整数（由数字组成的字符串）可以表示为 **/[0-9][0-9]*/**。（为什么不是直接写成 **/[0-9]*/**？）

有时需要两次写出表示数字的正则表达式（为了防止匹配空字符串的情况）会让人觉得麻烦，因此有一种更短的方式来指定“至少一个”字符。这就是克林加号（Kleene +），它表示“前一个字符或正则表达式出现一次或多次”。因此，**/[0-9]+/** 是指定“一串数字”的常规方式。那么页有两种方式来指定上述绵羊语言：**/baaa*!/**或 **/baa+!/**。

一个非常重要的特殊字符是句点（英文句号） **/./**，它是一个通配符表达式，用以匹配除回车符外的任意单个字符，如图2.6所示。

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

图 2.6 使用句点来匹配任意字符

通配符 **.** 通常与克林星号 ***一起使用来表示“任意字符串”。例如，假设我们想找到任何一行中某个特定单词（例如 *aardvark*）出现两次的情况，可以用正则表达式 ***/aardvark.*aardvark/*** 来指定。

锚点 是将正则表达式固定在字符串特定位置的特殊字符。最常见的锚点是插入符号 **^** 和美元符号 **$**。插入符号 **^** 匹配行的开头。模式 **/^The/** 仅匹配位于行首的单词 *The*。总结一下，插入符号 **^** 有三种用途：匹配行的开头、在方括号内表示否定，以及表示插入符号本身。（grep 或 Python 是如何根

据上下文判断给定的插入符号应该执行哪个功能的？）美元符号 **$** 用于匹配行尾。因此，模式 **_$** 是匹配行尾空格的一个有用模式，**/^The dog\.$/** 匹配仅包含短语 *The dog.* 的一行（我们在这里必须使用反斜杠，因为我们希望 **.** 表示句号，而不是通配符）。

此外，还有两个锚点：**\b** 匹配单词边界，**\B** 匹配非单词边界。因此，**/\bthe\b/** 匹配单词 *the*，但不匹配单词 *other*。在正则表达式中，**“单词”** 是基于编程语言中的定义，指的是由数字、下划线或字母组成的序列。因此，**/\b99\b/** 将匹配字符串 *There are 99 bottles of beer on the wall* 中的 *99*（因为 *99* 后跟空格，即该*99*是一个单独的词），但不会匹配字符串 *There are 299 bottles of beer on the wall* 中的 *99*（因为 *99* 前有数字）。不过，它会匹配 *$99* 中的 *99*（因为 *99* 后跟的美元符号不是数字、下划线或字母）。

2.1.2 选择、分组与优先级

假设我们需要搜索关于宠物的文本，并且特别对猫和狗感兴趣，我们可能去搜索字符串*cat*或字符串*dog*。由于我们不能使用方括号来搜索“cat 或 dog”（为什么不能写 **/[catdog]/** 呢？），我们需要一个新的运算符，即选择运算符，也叫做管道符号 **|**。模式 **/cat|dog/** 匹配字符串“cat”或字符串“dog”。

有时我们需要在更长的序列中使用这种选择运算符。例如，假设我想为我的堂弟大卫搜索关于宠物鱼的信息。如何同时指定 *guppy* 和 *guppies*？我们不能简单地写 **/guppy|ies/**，因为那只会匹配 *guppy* 和 *ies* 两个字符串。这是因为像*guppy*这样的序列优先于选择运算符 **|**。为了让选择运算符只应用于特定模式，我们需要使用括号运算符 **( )**。将一个模式包在括号中，使其对于像 **|** 和克林星号 ***** 这样的邻近运算符表现得像一个单一字符。因此，模式 **/gupp(y|ies)/** 指定选择运算符仅适用于后缀 *y* 和 *ies*。

括号运算符在使用像克林星号 ***** 这样的计数器时也很有用。与 **|** 运算符不同，克林星号默认只应用于单个字符，而不是整个序列。假设我们想匹配重复出现的字符串。可能我们有一行列标签，如 *Column 1 Column 2 Column 3*。表达式 ***/Column_[0-9]+_*/*** 不会匹配任意数量的列；它只会匹配一个列标签，后面跟任意数量的空格！这里的星号只适用于它前面的空格，而不是整个序列。使用括号后，我们可以写出 **/(Column_[0-9]+_*)*/**来匹配 *Column* 这个单词，后跟数字和可选的空格，整个模式可以重复零次或多次。

一个运算符可能优先于另一个运算符，因此需要我们使用括号来明确指定我们的目的。这一概念通过正则表达式的运算符优先级层次结构形式化了。下表给出了正则表达式运算符的优先级顺序（最高到最低）。

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

因此，由于计数器的优先级高于序列，**/the*/** 匹配 *theeeee*，但不匹配 *thethe*。由于序列的优先级高于选择，**/the|any/** 匹配 *the* 或 *any*，但不匹配 thany 或 *theny*。

模式还存在另一种方式的歧义。考虑表达式 **/[a-z]*/** 在匹配文本 *once upon a time* 时的情况。由于 **/[a-z]*/** 匹配零个或多个字母，该表达式可能匹配空字符串，或只是第一个字母 *o* 、 *on* 、 *onc* 或 *once* 。在这些情况下，正则表达式总是匹配它们能匹配到的最长字符串；我们称这种模式为**贪婪**的，即尽不断扩展以覆盖尽可能多的字符串。

然而，确实有办法强制**非贪婪匹配**，通过使用 **?** 修饰符的另一种含义。运算符 ***?** 是一个克林星号，匹配尽可能少的文本。运算符 **+?** 是一个克林加号，匹配尽可能少的文本。

2.1.3 一个简单的例子

假设我们想写一个正则表达式来查找英语冠词 *the*。一个简单（但不正确）的模式可能是：

/the/。

但问题是这个模式会错过在句首时被大写的单词（例如 *The*）。这可能会让我们想到以下模式：

/[tT]he/。

但这个模式仍然会错误地返回嵌入在其他单词中的 *the*（例如 *other* 或 *theology*）。因此，我们需要指定我们要查找的是单词边界两侧都有空格的情况：

/\b[tT]he\b/。

因为 **/\b/** 不会将下划线和数字视为单词边界，所以匹配的结果有 *the_* 或 *the25*（旁边有下划线或数字）。但我们想要更“纯净”一点的，因此我们需要指定我们想要的是 *the* 两侧没有字母：

/[^a-zA-Z][tT]he[^a-zA-Z]/。

但这个模式还有一个问题：它不会找到位于行首的 *the*，因为我们使用的正则表达式 **[^a-zA-Z]**（用于避免嵌入式的 *the*）意味着在 *the* 前面必须有某个非字母字符。我们可以通过指定在 *the* 前面要求要么是行的开头，要么是非字母字符，在 *the* 结束时同样如此，来避免这个问题：

/(^|[^a-zA-Z])[tT]he([^a-zA-Z]|$)/。

我们刚刚经历的过程是基于修正两类错误：**假阳性**（即我们错误匹配的字符串，如 *other* 或 *there*），以及**假阴性**（即我们错误遗漏的字符串，如 *The*）。在语言处理过程中，不断解决这两类错误是反复出现的。减少应用程序的整体错误率涉及两个对立的努力：

- **提高精确率**（即最小化假阳性）
- **提高召回率**（即最小化假阴性）

我们将在第4章中以更精确的定义回到精确率和召回率的问题。

2.1.4 更多的操作符

图2.8展示了一些常用范围的别名，可以节省输入时间。除了克林星号*和克林加号+，我们还可以使用明确的数字作为计数器，通过将它们放在**大括号**中来实现。运算符 **/{3}/** 表示“前一个字符或表达式恰好出现3次”。例如，**/a\.{24}z/** 匹配一个 *a* 后跟24个点，再跟一个 *z*（但不匹配前面是23个或25个点的 *a* 和 *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

图 2.8 常见字符集的别名

我们还可以指定一个数字范围。**/{n,m}/** 表示前一个字符或表达式出现 *n* 到 *m* 次，**/{n,}/** 表示前一个表达式至少出现 **n** 次。图2.9总结了用于计数的正则表达式。

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

图 2.9 用于计数的正则表达式运算符

最后，某些特殊字符通过基于反斜杠 \ 的特殊符号来引用（见图2.10）。其中最常见的是换行符 \n 和制表符 \t。如果要引用那些本身是特殊字符的符号（如 .、*、[和 \），需要在它们前面加上反斜杠。

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	

图 2.10 一些需要转义的字符（通过反斜杠）

2.1.5 一个更复杂的例子

让我们以一个更具代表性的例子来展示正则表达式的强大功能。假设我们的目标是帮助一个用户在网上选购电脑，他想要“至少6 GHz处理器和500 GB硬盘，价格低于1000美元”。为了实现这种检索，我们首先需要能够查找类似于 *6 GHz*、*500 GB* 或 *$999.99* 这样的表达式。让我们为这个任务设计一些正则表达式。

首先，让我们完成价格的正则表达式。以下是一个表示美元符号后跟数字字符串的正则表达式：

/$[0-9]+/。

注意，**$** 字符在这里的作用与我们之前讨论的行尾符号不同。大多数正则表达式解析器足够智能，可以判断此处的 **$** 并不是指行尾。（作为一个思维实验，想一想正则表达式解析器如何根据上下文判断 **$** 的功能。）

现在我们只需要处理美元的小数部分。我们将添加一个小数点和两个数字：

/$[0-9]+\.[0-9][0-9]/。

这个模式只允许 *$199.99*，但不允许 *$199.*。我们需要使小数点及其后面部分这一整体变为可选，并确保我们在一个单词边界：

/(^|\W)$[0-9]+(\.[0-9][0-9])?\b/。

还有一个问题！这个模式允许像 *$199999.99* 这样的价格，而这显然太贵了！我们需要限制美元部分的位数：

/(^|\W)$[0-9]{0,3}(.[0-9][0-9])?\b/。

进一步的修正（例如避免匹配只有美元符号却没有价格的情况 **/(^|\W)\$[0-9]+(\.[0-9]{2})?\b/**）留给读者作为练习。

那么，硬盘空间如何表示呢？我们需要再次允许可选的小数部分（如 *5.5 GB*）；注意 **?** 用于使结尾的 s 可选，并使用 **/_ */** 表示“零个或多个空格”，因为可能会有多余的空格存在：

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

修改此正则表达式以仅匹配大于500 GB的情况（**/\b(5[0-9]{2,}|[6-9][0-9]{2,}|[1-9][0-9]{3,})(\.[0-9]+)? *(GB|[Gg]igabytes?)\b/**），同样留给读者作为练习。

2.1.6 替换、捕获组和 ELIZA

正则表达式的一个重要用途是进行**替换（Substitution，s）**。例如，替换运算符 **s/regexp1/pattern/** 在 Python 和 Unix 命令（如 vim 或 sed）中用于将匹配正则表达式的字符串替换为另一个字符串，例如：

s/colour/color/

通常，我们希望引用匹配第一个模式的字符串的某个子部分。假设我们想在文本中的所有整数周围加上尖括号，例如将 **35 boxes** 改为 **<35> boxes**。我们希望有一种方法引用找到的整数，以便轻松地添加括号。为此，我们在第一个模式周围加上括号 **( )**，并在第二个模式中使用数字运算符 **\1** 来引用前面的匹配的结果。如下所示：

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

括号和数字运算符还可以指定某个字符串或表达式必须在文本中出现两次。例如，假设我们正在查找这样的模式：“the Xer they were, the Xer they will be”，并且我们希望两个 X 是相同的字符串。我们通过将第一个 X 用括号包围，并用 **\1** 代替第二个 X，如下：

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

这里的 **\1** 将被替换为与第一个括号中的内容匹配的字符串。因此，这将匹配 *the bigger they were, the bigger they will be*，但不匹配 *the bigger they were, the faster they will be*。

这种使用括号将模式存储在内存中的方式称为**捕获组（capture group）**。每次使用捕获组（即，括号包围一个模式）时，匹配的结果会存储在一个编号的**寄存器**中。如果你匹配了两个不同的括号组，**\2** 表示匹配第二个捕获组的内容。因此，

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

将匹配 *the faster they ran, the faster we ran*，但不匹配 *the faster they ran, the faster we ate*。同样，第三个捕获组存储在 **\3**，第四个捕获组存储在 **\4**，以此类推。

因此，**括号在正则表达式中有双重功能**：它们用于对术语进行分组，以指定运算符应用的顺序，还用于将匹配的内容捕获到寄存器中。有时我们可能希望只使用括号进行分组，但不希望将其捕获到寄存器中。这种情况下，我们使用**非捕获组(non-capturing group)**，通过在开括号后添加特殊命令 **?:**，形式如下：**(?: pattern)**。

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

将匹配 *some cats like some cats*，但不匹配 *some cats like some some*。

替换和捕获组在实现像 ELIZA 这样的简单聊天机器人中非常有用（Weizenbaum, 1966）。在上面我们提到，ELIZA 模拟了一位罗杰斯派心理学家，通过类似于下面的对话进行交流：

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 的工作原理是使用一系列或级联的正则表达式替换，每一个都匹配并更改输入文本的某些部分。在输入内容被转换为大写后，替换会将所有 *MY* 替换为 *YOUR*，将 *I'M* 替换为 *YOU ARE*，等等。当 ELIZA 重复用户的部分话语时，似乎就能够正确地指代用户。下图所示的一组替换会匹配并替换输入中的其他模式。

```
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/
```

由于多个替换可以应用于同一输入，所以ELIZA会为各种替换赋予等级并按顺序应用。创建模式是练习 2.3 的主题，我们将在第 15 章详细讨论 ELIZA 的架构。

2.1.7 前瞻断言

有时候我们需要“预测未来”：在文本中提前查看是否有某个模式匹配，但不移动我们在文本中当前指针的位置，以便当该模式出现时我们可以处理它；如果没有出现，则可以检查其他内容。

这些**前瞻断言**使用我们在上一节中提到的非捕获组的 **(?** 语法。

- 正向预查（**?= pattern**）：它检查在当前位置之后是否出现某个模式，但不会将指针向前移动。如果模式匹配，则返回 true。
- 负向前瞻（ **?! pattern** ）：它检查在当前位置之后是否没有出现某个模式。如果模式不存在，它将返回 true，与正向前瞻一样，它不会移动指针。

例如，假设我们希望匹配行首的任意一个不以“Volcano”开头的单词。我们可以使用负前瞻来实现这一点：

/^(?!Volcano)[A-Za-z]+/

简单来说，lookahead 允许你前瞻即将到来的文本来决定是否匹配某些内容，但不会超过当前位置或将“窥视”的部分包含在最终结果中。

2.2 Word

在讨论处理单词之前，我们需要决定什么算作单词。先来看一个特定的**语料库（corpus，复数为corpora）**，即一个计算机可读的文本或语音集合。例如，布朗语料库（Brown corpus）是由布朗大学在1963-1964年间（Kučera和Francis, 1967）收集的，包含500个不同类型（报纸、小说、非小说、学术等）英文书面文本的样本，总共约100万字。下面是布朗语料库中的一个句子，其中有多少个单词呢？

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

如果不算标点符号，这个句子有13个单词；如果算上标点符号，则有15个单词。是否将句号（“.”）、逗号（“,”）等视为单词取决于具体任务。标点符号在寻找边界（如逗号、句号、冒号）和识别某些语义方面（如问号、感叹号、引号）非常重要。有时在某些任务如词性标注、句法分析或语音合成中，我们会把标点符号当作独立的单词来处理。

Switchboard语料库是关于陌生人之间的美国英语电话对话，收集于20世纪90年代初；它包含2430个对话，每个对话平均时长6分钟，总计240小时的语音和约300万单词（Godfrey等，1992年）。此类口语语料库在定义单词时引入了其他复杂因素。让我们来看Switchboard中的一个话语（utterance，话语对应口语中的句子）：

I do uh main- mainly business data processing

这个话语中有两种语音不流利现象。被截断的单词*main-*称为**片段（fragment）**。像*uh*和*um*这样的词称为**填充词**或**填充停顿**。我们应该把这些当作单词吗？这仍然取决于应用场景。如果我们在构建语音转录系统，可能最终会去掉这些不流利现象。

但有时我们也会保留这些不流利现象。像*uh*或*um*这样的不流利现象在语音识别中实际上很有帮助，它们可以预测即将出现的单词，因为它们可能表明说话者正在重新启动句子或想法，因此在语音识别中，它们被视为常规单词。由于不同的人使用不同的不流利现象，它们也可以作为识别说话者的线索。Clark和Fox Tree（2002年）研究表明*uh*和*um*有不同的含义。你认为它们的不同含义是什么？

或许最重要的是，我们需要在考虑什么是单词时区分两种讨论单词的方式，这在整本书中都将非常有用。**词型（word types）**是语料库中不同的单词数量；如果词汇表的集合是V，词型的数量就是词汇表的大小|V|。**词实例（word instances）**是“运行”中所有单词的总数量 *N*。如果忽略标点符号，下面的布朗语料库句子有14个词型和16个词实例。

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

我们还有决定需要做！例如，我们是否应该把一个大写字符串（如*They*）和一个小写字符串（如*they*）视为同一个词型？答案是，这取决于具体任务！在某些任务中，如语音识别，我们更关心单词的顺序，而不太关注格式，因此*They*和*they*可能会被归为同一个词型，而在其他任务中，例如判断某个词是否为人名或地名（命名实体识别任务），大写是一个有用的特征而因此会保留。**有时我们会保留某个特定NLP模型的两个版本，一个保留大小写，一个不保留。**

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

图 2.11 一些英语语料库的词形类型和实例的粗略数量。最大的 Google n-gram 语料库包含 1300 万种类型，但此计数仅包括出现 40 次或更多次的类型，因此真实数量会大得多。

英语中有多少个单词？当我们讨论语言中的单词数量时，通常指的是词型数量。图2.11展示了从一些英语语料库中计算出的词型和词实例的大致数量。研究的语料库越大，发现的词型越多，实际上词型数量|V|与词实例数量N之间的关系被称为**赫尔丹定律**（Herdan’s Law，1960）或**希普斯定律**（Heaps’ Law，1978），以其发现者（分别在语言学和信息检索领域）命名。式2.18展示了这一关系，其中k和β是正常数，且0 < β < 1。

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

β的值取决于语料库的大小和类型，但至少对于图2.11中的大语料库，β范围在0.67到0.75之间。因此，我们可以粗略地说，词型数量增长速度明显快于词示例数量的平方根。

有时我们还会进行进一步区分。考虑诸如*cats*和*cat*之类的词的变化形式。我们说这两个词是不同的**词形（wordforms）**，但它们有相同的**词干（lemma）**。**词干**是一组具有相同词干、相同主要词性和相同词义的词汇形式。**词形**是词的变化或派生形式。因此，*cats*和*cat*这两个词形具有相同的词干，可以表示为*cat*。

对于像阿拉伯语这样形态复杂的语言，我们经常需要进行**词干化（lemmatization）**处理。然而，对于大多数英语任务，词形已经足够了，在本书中当我们谈论单词时，我们几乎总是指词形（尽管我们将在2.6节讨论词干化及相关的词干提取算法）。但当我们衡量词典中的单词数量时是基于词干的。字典条目或粗体形式是词元数量（上限）非常粗略的近似值（因为有些词元有多个粗体形式）。1989 年版的《牛津英语词典》有 615,000 个词条。

最后，我们应该注意到，在许多实际的NLP应用中（例如神经语言建模），我们实际上并不使用单词作为内部表示的单元！我们将输入字符串**分词**成**标记（tokens）**，这些标记可以是单词，但也可以是单词的一部分。当我们介绍第2.5.2节的**BPE**算法时，我们将回到这个分词问题。

2.3 语料库

单词不会凭空出现。我们研究的任何特定文本都是由一个或多个特定的说话者或作者在特定时间、特定地点、使用特定语言的特定方言为特定目的所产生的。

也许文本最重要的变异是其使用的语言。NLP 算法在适用于多种语言时最有用。根据 Ethnologue 在线目录（Simons 和 Fennig, 2018），截至本文撰写时，全球有 7097 种语言。因此，测试算法时应超越单一语言，尤其是在具有不同特性的语言上进行测试；遗憾的是，当前的 NLP 算法倾向于只在英语上开发或测试（Bender, 2019）。即使算法不局限于英语，它们也通常用于大型工业化国家的官方语言（如中文、西班牙语、日语、德语等），但我们不应仅限于这些少数语言。此外，大多数语言也有多种变体，通常由不同地区或不同社会群体使用。例如，如果我们处理使用非洲裔美国英语（**AAE**）或非洲裔美国白话英语（**AAVE**）特征的文本——数百万非洲裔美国社区成员使用的英语变体（King 2020）——我们必须使用能够处理这些变体特征的NLP工具。推特帖子可能使用非洲裔美国英语使用者常用的特征，比如 *iont*（对应主流美国英语中的 *I don't*）或 *talmbout*（对应主流美国英语中的 *talking about*），这些例子都影响到词语分割（Blodgett et al. 2016, Jones 2015）。

说话者或作者在一次交流中使用多种语言的现象也很常见，这被称为**编码转换**。编码转换在全球范围内非常普遍，以下是西班牙语和（音译的）印地语与英语编码转换的例子（Solorio et al., 2014；Jurgens et al., 2017）：

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

另一个变异维度是文本的**体裁**。我们的算法需要处理的文本可能来自新闻通讯、小说或非小说类书籍、科学文章、维基百科或宗教文本。它可能来自口语类的文本，如电话交谈、商务会议、警察随身摄像机录音、医学访谈或电视节目或电影的转录文本。它还可能来自工作场景，如医生的笔记、法律文本或议会或国会的记录。

文本还反映了作者（或说话者）的**人口特征**：年龄、性别、种族、社会经济阶层都会影响我们处理的文本的语言特性。

最后，**时间**也很重要。语言会随着时间而变化，对于某些语言，我们拥有不同历史时期的优质语料库。

由于语言是有情境的，因此在从语料库中开发语言处理的计算模型时，考虑是谁产生了这些语言、在什么情境下、为了什么目的非常重要。数据集的用户如何了解所有这些细节？最好的方法是让语料库创建者为每个语料库制作一个**数据表**（Gebru et al., 2020）或**数据声明**（Bender et al., 2021）。数据表规定了数据集的属性，例如：

- **动机**：语料库是由谁收集的，为什么收集，谁资助的？

- **情境**：文本是在何时何地编写或口述的？例如，是否有任务存在？语言是原本的口语对话、经过编辑的文本、社交媒体交流，还是独白与对话？
- **语言变体**：语料库使用的是什么语言（包括方言/地区）？
- **说话者人口特征**：例如，文本作者的年龄或性别是什么？
- **收集过程**：数据的规模多大？如果是子样本，如何采样的？数据是否经过同意后收集？数据如何进行预处理，有哪些可用的元数据？
- **注释过程**：注释内容是什么？注释者的人口特征是什么？他们如何培训，数据是如何注释的？
- **分发**：是否有版权或其他知识产权限制？

2.4 用于单词分词的简单 Unix 工具

在对文本进行几乎任何自然语言处理之前，必须对文本进行规范化，这项任务称为**文本规范化**。通常，文本规范化过程包含至少三项任务：

1. **分词**（对单词进行分割）
2. **规范化单词格式**
3. **分割句子**

在接下来的章节中，我们将逐步介绍每一项任务，但首先我们从一个简单朴素的英文分词、规范化（以及频率计算）开始，通过单个 Unix 命令行就可以完成。我们将使用一些 Unix 命令：**tr**，用于系统地改变输入中的特定字符；**sort**，按字母顺序对输入行进行排序；以及 **uniq**，用于合并和统计相邻的相同行。

例如，假设我们将莎士比亚的“完整作品”放在一个文件 **sh.txt** 中。我们可以使用 **tr** 来进行分词，将每个非字母字符序列替换为换行符（‘A-Za-z’ 表示字母，-c 选项表示补充非字母，因此两者结合意味着将每个非字母字符替换为换行符。-s（‘squeeze’）选项用于将多个连续替换结果合并为单个输出，因此一系列连续的非字母字符将被‘压缩’为一个换行符）【这段话的意思是，我们使用 `tr` 命令将所有非字母字符（如标点、数字、空格等）转换为换行符，并且如果有连续的非字母字符，最终会只产生一个换行符。这有助于将文本中的单词逐个分隔开，从而实现分词。】：

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

输出结果为：

```
THE
SONNETS
by
William
Shakespeare
From
fairest
```

creatures

We

.....

现在每行有一个单词，我们可以对这些行进行排序，并将它们传递给 uniq -c，它将合并相同的行并进行计数：

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

输出如下：

1945 A

72 AARON

19 ABBESS

25 Aaron

6 Abate

1 Abates

5 Abbess

6 Abbey

3 Abbot

.....

或者，我们可以将所有大写字母与小写字母进行合并：

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

输出结果为：

14725 a

97 aaron

1 abaissiez

10 abandon

2 abandoned

2 abase

1 abash

14 abate

3 abated

3 abatement

.....

现在我们可以再次排序以查找常用词。用于排序的 **-n** 选项表示按数字而不是字母顺序排序，而 **-r** 选项表示按相反顺序（从最高到最低）排序：

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

结果表明，与任何其他语料库一样，莎士比亚语料库中最常见的词是短虚词，如冠词、代词、介词：

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

这种 Unix 工具在为任何英文语料库快速构建词频统计时非常方便。虽然在某些 Unix 版本中，这些命令行工具也能正确处理 Unicode 字符，从而可以用于多种语言，但通常对于处理英语以外的大多数语言，我们需要使用更复杂的分词算法。

2.5 单词和子词分词

上面提到的简单 Unix 工具仅适用于获取粗略的单词统计信息，但我们通常需要更复杂的算法来进行分词。大致来说，分词算法分为两类。

在自上而下的分词中，我们定义一个标准并通过规则实现这种类型的分词。

但更常见的是，NLP 算法的输入并不是完整的单词，而是将单词分解为**子词标记（subword tokens）**，这些标记可以是完整单词、单词的一部分，甚至是单个字母。我们利用字母序列的简单统计数据，生成子词标记的词汇表，并将输入以自下而上的方式分解为子词标记。

2.5.1 自上而下（基于规则的）分词

虽然前面提到的 Unix 命令序列去除了所有数字和标点符号，但在大多数自然语言处理（NLP）应用中，我们需要在分词时保留这些字符。我们通常希望将标点符号分离出来作为独立的标记；逗号对于解析器来说是有用的信息，而句号有助于指示句子的边界。然而，在某些情况下，我们希望保留出现在单词内部的标点符号，例如 *m.p.h.*、*Ph.D.*、*AT&T* 和 *cap’n*，价格中的特殊字符和数字（如 *$45.55*）以及日期（如 *01/02/06*）也需要保留；我们不希望将这些价格分解为“45”和“55”两个独

立的标记。此外，还有 URL（如 https://www.stanford.edu）、推特标签（如 #nlproc）或电子邮件地址（如 someone@cs.colorado.edu）。

数字表达式引入了一些复杂性；除了出现在单词边界外，逗号在英文数字中每三位出现一次，如 *555,500.50*。分词在不同语言中有所不同；例如，西班牙语、法语和德语使用逗号表示小数点，并使用空格（有时是句号）来标记英文中使用逗号的地方，例如*555 500,50*。

分词器还可以用于展开带有撇号的**缩略**形式，将 *what're* 转换为两个标记 *what are*，将 *we're* 转换为 *we are*。**附加词素**是一种不能独立存在的词素，只有附加在另一个单词上时才能出现。这类缩略形式也出现在其他字母语言中，包括法语中的代词（如 *j'ai* 和冠词 *l'homme*）。

根据具体的应用，分词算法还可以将多词表达式（如 *New York* 或 *rock 'n' roll*）作为一个单一的标记进行分词，这需要某种多词表达式词典。因此，分词与命名实体识别（检测名称、日期、组织等任务，第17章）紧密相关。

一个常用的分词标准是 **Penn Treebank 分词（Penn Treebank tokenization）**标准，用于由语言数据联盟（Linguistic Data Consortium, LDC）发布的已解析语料库（treebanks，树库），LDC 是许多有用数据集的来源。该标准将附加词素分离出来（*doesn't* 变为 *does* 加 *n't*），将连字符单词保持在一起，并将所有标点符号分离出来（下面的例子中，为了节省空间，我们在此展示的输出的标记之间是可见的空格'␣'，但实际情况中换行符是更常见的输出格式）：

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

输出："␣The␣San␣Francisco-based␣restaurant␣,␣"␣they␣said␣,␣"␣does␣n't␣charge␣$␣10␣"␣.

在实际应用中，由于分词是在任何其他处理之前运行的，因此它必须非常快速。对于单词分词，我们通常使用基于正则表达式的确定性算法，这些正则表达式被编译成高效的有限状态自动机。例如，图2.12展示了一个基本的正则表达式，它可以与 Python 中的 Natural Language Toolkit（NLTK）中的 **nltk.regexp_tokenize** 函数一起使用来分词英文文本（Bird 等, 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', '...']
```

图 2.12 基于Python的自然语言处理工具包NLTK(Bird et al , 2009)中的正则表达式的Python路径；(? x)标志告诉Python去掉注释和空格。图来自Bird等人(2009)的第3章。

精心设计的确定性算法可以处理出现的歧义，例如，撇号在不同情况下需要进行不同的分词处理：作为所有格标记（如 *the book’s cover*）、引用符号（如 *‘The other class’, she said*）或附加词素（如 *they’re*）时，处理方式不同。

一些语言的分词比英语复杂得多，比如书面中文、日语和泰语，这些语言不使用空格来标记单词边界。例如，在中文中，词是由汉字组成的。每个汉字通常表示一个独立的语义单位（称为**词素，morpheme**），并且可以发音为一个音节。中文词的平均长度大约为2.4个字。但是，决定什么算作中文词是很复杂的。例如，考虑以下句子：

姚明进入总决赛　　yáo míng jìn rù zǒng jué sài
“Yao Ming reaches the finals”

正如Chen et al (2017)指出的那样，这可以理解为3个词("中国树库"分词)：

姚明　　进入　　总决赛
YaoMing reaches finals

或如5字("北京大学"分词)：

姚　明　　进入　　总　　决赛
Yao Ming reaches overall finals

还可以简单地忽略单词，使用字符作为基本元素，将句子视为7个字符的序列：

姚　明　　进　　入　　总　　决　　赛
Yao Ming enter enter overall decision game

实际上，对于大多数中文 NLP 任务，使用字符（汉字）而不是词作为输入效果更好，因为字符在大多数应用中处于合理的语义层次，而且与词相比，字符标准不会产生大量极为罕见的词汇（Li 等，2019）。

然而，对于日语和泰语来说，字符是一个过小的单位，因此需要算法进行分割。在某些需要区分单词而不是字符边界的罕见情况下，这些算法也对中文有用。在这些情况下，我们可以使用下一节中介绍的子词分词算法。

2.5.2 字节对编码：自下而上的分词算法

分词还有第三种方法，通常用于大语言模型（large language models）中。这种方法不是将标记定义为单词（无论是通过空格分隔还是更复杂的算法），也不是像中文那样将标记定义为字符，而是利用数据自动生成标记。这在处理未知单词（语言处理中的一个重要问题）时尤其有用。正如我们将在下一章看到的，NLP算法通常会从一个语料库（训练语料库）中学习一些语言知识，然后利用这些知识对另一个独立的测试语料库进行决策。如果我们的训练语料库包含单词*low*、*new*、*newer*，但不包含 *lower*，那么当*lower*出现在测试语料库中时，系统将不知道如何处理它。

为了应对这个未知单词的问题，现代分词器会自动生成一组包含比单词更小的标记，称为**子词（subwords）**。子词可以是任意的子字符串，也可以是具有意义的单位，例如词素*-est*或*-er*（词素是语言中最小的意义单位；例如，单词*unwashable*包含*un-*、*wash*和 *-able* 这几个词素。）在现代分词

方案中，大多数标记是单词，但一些标记是频繁出现的词素或其他子词，例如*-er*。因此，像*lower*这样的未知单词可以通过已知的子词单位序列表示出来，或者如果有必要，甚至可以作为单个字母序列来表示。

大多数分词方案包含两个部分：**标记学习器**和**标记分段器**。标记学习器处理原始训练语料库（有时粗略地通过空格分隔成单词），并生成一个词汇表，即一组标记。标记分段器则处理原始的测试句子，并将其分段为词汇表中的标记。有两种算法被广泛使用：**字节对编码**（**Byte-Pair Encoding，BPE**，Sennrich et al., 2016），以及**一元语言建模**（**unigram language modeling**，Kudo, 2018）。此外，还有一个名为**SentencePiece**的库，其中包含这两种算法的实现（Kudo和Richardson, 2018），但人们通常使用SentencePiece这个名称来指代一元语言建模分词。

在本节中，我们介绍这三种方法中最简单的一种：BPE算法（Sennrich et al., 2016）；参见图2.13。BPE标记学习器首先从仅包含所有单个字符的词汇表开始。然后它检查训练语料库，选择最常见的相邻符号（例如‘A’和‘B’），将新合并的符号‘AB’添加到词汇表中，并将语料库中所有相邻的‘A’‘B’替换为新的‘AB’。它继续统计并合并，创建越来越长的字符串，直到完成k次合并，生成k个新标记（k即算法的一个参数）。最终生成的词汇表包含最初的一组字符加上k个新符号。该算法通常在单词内部运行（不会跨单词边界进行合并），因此首先通过空格分隔语料库，生成一组字符串，每个字符串对应于一个单词的字符，并加上一个特殊的词尾符号 _ 及其计数。

让我们看一下它在一个18个单词标记地小型语料库上的操作。下面给出了每个单词的计数（例如，单词*low*出现5次，*newer*出现6次，依此类推），起始词汇表共有11个字母。

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 _	

BPE算法首先统计所有相邻符号对：最常见的是符号对‘er’，因为它出现在*newer*（频率为6次）和*wider*（频率为3次）中，共出现了9次。接着，我们合并这些符号，将‘er’视为一个符号，再次统计。

现在，最常见的符号对是‘er’，我们将其合并；我们的系统学习到，应该为单词结尾的‘er’创建一个标记，表示为‘er’。

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 _	

现在出现频率最高的一对是‘er_’，我们将其合并；我们的系统已经了解到词尾er应该有一个token，表示为‘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 _	

接下来将 n e (总计数8)合并为 ‘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 _	

后续的合并是：

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_

一旦我们学习了词汇表，标记分段器就用于对测试句子进行分词。标记分段器只是根据我们从训练数据中学到的合并规则，对测试数据进行贪婪地分割（即按照我们学习的顺序应用这些规则）。因此，测试数据中的频率并不起作用，起作用的是训练数据中的频率。首先，我们将每个测试句子的单词分割成字符。然后我们应用第一个规则：将测试语料库中所有的‘e r’替换为‘er’，接着应用第二个规则：将所有‘er _’替换为‘er_’，依此类推。到最后，如果测试语料库中包含字符序列‘n e w e r _’，它将被标记为一个完整单词。但像*lower*这样的新（未知）单词的字符‘l o w e r _’会被合并成两个标记：*low* 和 *er_*。

当然，在实际环境中，BPE通常在非常大的输入语料库上进行成千上万次合并，可以让大多数单词将作为完整的符号表示，只有极少数非常罕见的单词（和未知单词）才需要通过它们的部分来表示。

2.6 词规范化、词形还原与词干提取

词规范化（Word normalization）是将单词或标记转化为标准格式的任务。最简单的词汇规范化是**大小写折叠（case folding）**。将所有字母映射为小写意味着“Woodchuck”和“woodchuck”将被相同地表示，这对于许多任务（如信息检索或语音识别）中的泛化非常有帮助。相反，在情感分析、文本分类任务、信息抽取和机器翻译中，保留大小写信息通常很有帮助，因此一般不进行大小写折叠。这是因为在某些情况下，保留大小写的差异可能比泛化带来的好处更重要，例如区分“US”（指国家）和“us”（指代词）。有时，我们会同时生成区分大小写（即包含大小写字母的词汇或标记）和不区分大小写的语言模型版本。

使用字节对编码（BPE）或其他自下而上分词方式的系统，可能不会再进行进一步的词汇规范化。在其他 NLP 系统中，我们可能希望进一步规范化，例如为多个形式的单词选择一个标准形式，

如“USA”和“US”或“uh-huh”和“uhhuh”。尽管在规范化过程中会丢失拼写信息，但这种标准化可能是有价值的。比如，在进行关于“USA”的信息检索或信息抽取时，无论文档中提到的是“US”还是“USA”，我们都希望能找到相关信息。

2.6.1 词形还原

在某些自然语言处理任务中，我们希望词形上不同的两个词能够表现得相似。例如，在网页搜索中，用户可能会输入*woodchucks*这个单词，此时一个有用的系统也会返回那些只提到*woodchuck*而没有*s*的页面。这种情况在形态复杂的语言中尤其常见，比如波兰语， *Warsaw* 这个单词在不同语法情境下会有不同的结尾：作为主语时是 *Warszawa* ，在介词后是*in Warsaw*（ *w Warszawie*），“*to Warsaw*”（ *do Warszawy*）等等。 词形还原的任务是确定两个单词具有相同的词根，尽管它们在表面上有所不同。比如， *am* 、 *are* 和 *is* 都有相同的词根 *be*； *dinner* 和 *dinners* 的词根都是 *dinner* 。将这些词形还原到同一个词根，可以让我们找到波兰语中所有像 Warsaw 一样的词形。比如，句子 *He is reading detective stories* 的词形还原形式是 *He be read detective story*。

词形还原是如何完成的？最复杂的词形还原方法涉及对单词的**完整形态分析（morphological parsing）**。**形态学（Morphology）**是研究单词如何由称为词素的小意义单位构成的学科。词素可以分为两大类： 词干 ——单词的核心词素，提供主要的词义；和词缀 ——为单词添加各种“附加”意义。例如，单词 *fox* 由一个词素（ fox ）组成，而单词 *cats* 由两个词素组成： *cat* 和 *-s* 。一个形态分析器可以将 *cats* 这个单词解析为两个词素 *cat* 和 *s* ，或者将西班牙语单词 *amaren* （中文意思为‘如果他们在未来会爱’）解析为词素 *amar* （中文意思为‘爱’）以及形态特征 *3PL* （第三人称复数和虚拟将来时 。

词干提取：Porter Stemmer

词形还原算法可能很复杂，因此我们有时使用一种更简单但粗糙的方法，主要是截去词尾的词缀。这种朴素的形态分析版本称为 词干提取 。例如，经典的 Porter词干提取器 （Porter, 1980）在以下段落中可以应用：

> *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.*

产生结果如下：

> *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*

该算法基于一系列的重写规则，逐步运行，每个步骤的输出作为下一个步骤的输入。以下是一些示例规则（更多规则请参见 [https://tartarus.org/martin/PorterStemmer/]）。

$$
\begin{aligned}
\text{ATIONAL} &\rightarrow \text{ATE} \quad \text{(e.g., relational} \rightarrow \text{relate)} \\
\text{ING} &\rightarrow \epsilon \quad \text{if the stem contains a vowel (e.g., motoring} \rightarrow \text{motor)} \\
\text{SSES} &\rightarrow \text{SS} \quad \text{(e.g., grasses} \rightarrow \text{grass)}
\end{aligned}
$$

简单的词干提取器在需要处理同一词根的不同变体时可能会有用。然而，现代系统中较少使用它们，因为它们存在两类错误： 过度泛化 （例如将 policy 还原为 police ），以及泛化不足 （例如未能将 European 还原为 Europe ）(Krovetz, 1993)。

2.7 句子分割

句子分割（Sentence segmentation）是文本处理中的另一个重要步骤。用于将文本分割成句子的最有用的线索是标点符号，例如句号、问号和感叹号。问号和感叹号是相对明确的句子边界标记。而句号（ “.” ）则有点模棱两可。句号既可以表示句子边界，也可以表示缩写标记，如 *Mr.* 或 *Inc.* 。你刚刚阅读的前一句（即‘如 *Mr.* 或 *Inc.* 。’ ）展示了这种歧义的一个更复杂的例子，其中 Inc. 的句末句号既表示缩写，又标记了句子的结束。因此，句子分词和单词分词可能需要一起处理。

通常，句子分词方法的工作原理是首先根据规则或机器学习决定句号是单词的一部分，还是句子的边界标记。 缩写词典可以帮助判断句号是否是常用缩写的一部分；这些词典可以手工构建或通过机器学习生成（Kiss 和 Strunk, 2006），最终的句子分割器也可以通过这两种方式获得。例如，在 Stanford CoreNLP 工具包（Manning 等, 2014）中，句子分割是基于规则的，是分词的确定性结果；当句末标点符号（如 .、！或 ？）没有与其他字符（如缩写或数字）组合成一个标记时，句子就会结束，标点符号后面还可以选择性地跟随最后的引号或括号。

2.8 最小编辑距离

许多自然语言处理任务都涉及测量两个字符串的相似度。例如，在拼写纠正中，用户输入了一个错误的字符串，比如“graffe”，我们想知道用户的真正意图。用户可能想输入的是一个与graffe相似的单词。在候选的相似单词中，giraffe只与graffe相差一个字母，直觉上比与graffe相差更多字母的单词（如grail或graf）更相似。另一个例子是共指（coreference）消解，即判断两个字符串是否指代同一实体的任务：

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

同样，这两个字符串非常相似（仅有一个单词的差异），这为判断它们可能是共指的提供了有用的依据。

编辑距离（Edit distance）为我们提供了一种量化字符串相似性的方式。形式上，两个字符串之间的最小编辑距离可定义为将一个字符串转换为另一个字符串所需的最少编辑操作（如插入、删除、替换）的次数。

例如，字符串intention和execution之间的编辑距离是5（删除一个i，将n替换为e，将t替换为x，插入一个c，将n替换为u）。通过观察字符串之间的对齐（如图2.14所示），更容易理解这一点。给定两个序列，对齐是两个序列的子串之间的对应关系。例如，I与空字符串对齐，N与E对齐，依此类推。在对齐字符串下方是另一种表示形式：一系列符号表示将顶部字符串转换为底部字符串的操作列表：d表示删除，s表示替换，i表示插入。

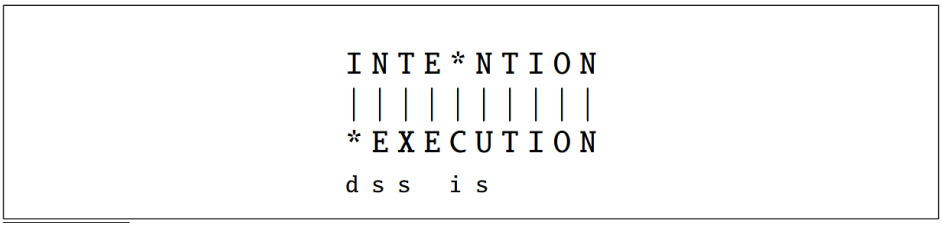

图 2.14 表示作为对齐的两个字符串之间的最小编辑距离。最后一行给出了将顶部字符串转换为底部字符串的操作列表：d表示删除，s表示替换，i表示插入。

我们还可以为每个操作分配特定的成本或权重。**Levenshtein**距离是最简单的加权方式，其中每种操作的成本都是1（Levenshtein, 1966）——我们假设字母替换为自身的成本为0，例如t替换为t的成本为0。intention和execution之间的Levenshtein距离是5。Levenshtein还提出了另一种度量方法，在这种方法中，每次插入或删除的成本为1，且不允许替换。（这相当于允许替换，但将每次替换的成本设为2，因为任何替换都可以通过一次插入和一次删除来表示）。使用这种版本，intention和execution之间的Levenshtein距离是8。

2.8.1 最小编辑距离算法

如何找到最小编辑距离？我们可以将其视为一个搜索任务，即从一个字符串到另一个字符串的最短路径——一系列编辑操作。

可能的所有编辑路径空间非常大，因此我们不能暴力地搜索。因为许多不同的编辑路径最终会到达相同的状态（即相同的字符串），所以我们可以避免重复计算这些路径，而是在每次看到某个状态时，记录到达该状态的最短路径。我们可以通过使用**动态规划（dynamic programming）**来实现这一点。动态规划是一类算法的名称，由 Bellman（1957）首次提出，使用一种图表驱动的方法，通过组合子问题的解决方案来解决问题。自然语言处理中一些常用的算法也利用了动态规划，例如 **Viterbi** 算法（第17章）和用于句法解析的 **CKY** 算法（第18章）。

动态规划问题的直观理解是，通过适当组合各种子问题的解决方案可以解决大问题。我们以表示intention和execution之间最小编辑距离的单词转换路径为例（如图2.16所示）。

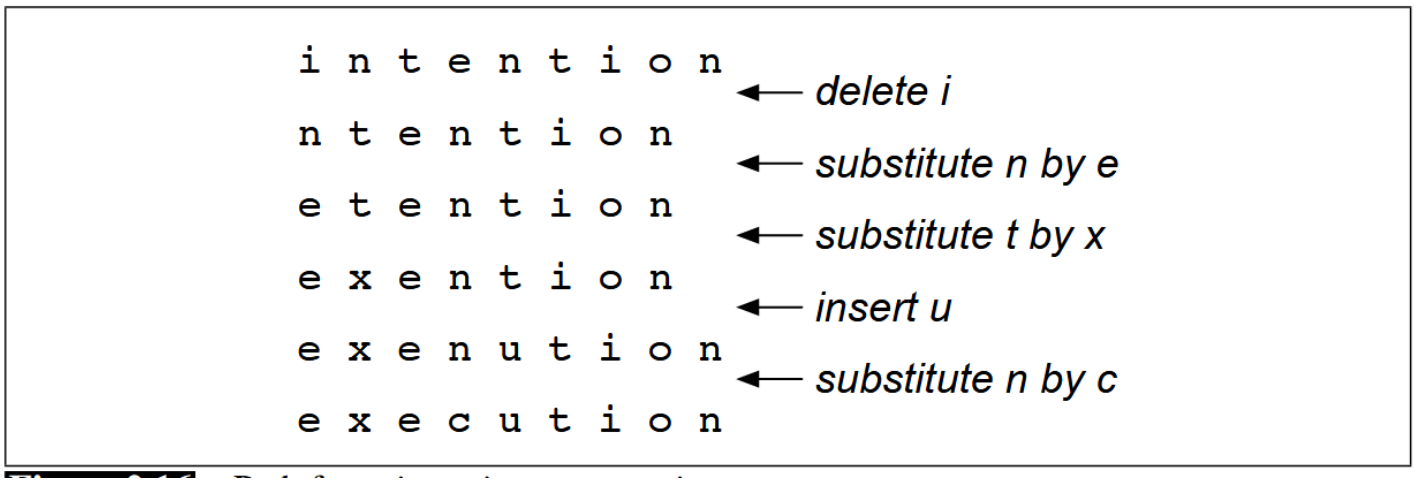

图 2.16 从 intention 到 excution 的编辑路径

假设某个字符串（例如 exention）在这个最优路径上（无论这个路径是什么）。动态规划的直觉是，如果 *exention* 在最优操作列表中，那么最优序列也必须包括从*intention*到*exention*的最优路径。为什么？如果从*intention*到*exention*有更短的路径，我们可以使用它，从而使整体路径更短，这样最优序列就不会是最优的，导致矛盾。

最小编辑距离算法由 Wagner 和 Fischer（1974）命名，但许多人独立发现了该算法（详见第17章的历史注释部分）。

首先，我们定义两个字符串之间的最小编辑距离。给定两个字符串，源字符串 X 长度为 n，目标字符串 Y 长度为 m，我们将 **D[i, j]** 定义为 X**[1..i]** 和 Y**[1..j]** 之间的编辑距离，即 X 的前 i 个字符和 Y 的前 j 个字符之间的编辑距离。因此，X 和 Y 之间的编辑距离是 D[n, m]。

我们将使用动态规划自下而上地计算 D[n, m]，通过组合子问题的解决方案来计算。在基准情况下，对于长度为 ***i*** 的源子串而目标字符串为空的情况，从 i 个字符变为 0 需要 i 次删除。对于长度为 **j** 的目标子串但源字符串为空的情况，从 0 个字符变为 j 个字符需要 **j** 次插入。计算出较小的 **D[i, j]** 后，我们基于之前计算的小值计算更大的 D[i, j]。D[i, j] 的值是通过取矩阵中三条可能路径中的最小值计算得出的：

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

前面提到的 Levenshtein 距离有两个版本：一个是替换的成本为 1，另一个是替换的成本为 2（即相当于一次插入加一次删除）。我们在这里使用第二个版本的 Levenshtein 距离，其中插入和删除的成本各为 1（ins-cost(·) = del-cost(·) = 1），替换的成本为 2（相同字母的替换成本为 0）。在该背景下，D[i, j] 的计算变为：

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

该算法的总结见图2.17；图2.18显示了使用该版本的 Levenshtein 距离计算*intention*和*execution*之间距离的结果。

```
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]
```

图 2.17 最小编辑距离算法，动态规划算法类的一个例子。各种成本可以是固定的(e.g. , ∂ x , ins-cost (x) = 1)，也可以是特定于字母(来模拟某些字母比其他字母更有可能被插入的事实)。我们假设用一个字母代替它本身没有成本(即子成本(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

图 2.18 使用图2.17的算法计算意图与执行之间的最小编辑距离，插入或删除使用代价为1的Levenshtein距离，替换使用代价为2的Levenshtein距离。

对齐

知道最小编辑距离对于拼写错误校正等算法很有用。但编辑距离算法还有另一种重要用途；通过小的改变，它还可以提供两个字符串之间的最小成本对齐。字符串对齐在语音和语言处理中非常有用。在语音识别中，最小编辑距离对齐用于计算词错误率（第16章）。对齐在机器翻译中也起着作用，在平行语料库（包含两种语言文本的语料库）中，需要将句子相互匹配。

为了将编辑距离算法扩展到对齐，我们可以将对齐视为通过编辑距离矩阵的一条路径。图2.19显示了用粗体标记的路径。每个粗体单元格代表两个字符串中一对字母的对齐。如果两个粗体单元格位于同一行，则表示从源字符串到目标字符串的过程中有一个插入操作；如果两个粗体单元格位于同一列，则表示有一个删除操作。

	#	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**

图 2.19 当在每个单元格中输入一个值时，我们用最多三个箭头标记我们来自的三个相邻单元格中的哪一个。表满后，我们通过使用回溯，从右下角的8开始，并在箭头后方计算对齐(最小编辑路径)。黑体单元格序列代表两个字符串之间的一种可能的最小成本比对，同样使用Levenshtein距离，成本为1用于插入或删除，2用于替换。

图2.19还展示了计算对齐路径的直觉。计算分两步进行。第一步，我们扩展最小编辑距离算法，在每个单元格中存储回指。每个单元格中的回指指向进入当前单元格时来自的前一个（或多个）单元格。图2.19展示了这些回指的示意图。某些单元格有多个回指，因为最小扩展可能来自多个先前的单元格。第二步，我们进行**回溯**。在回溯中，我们从最后一个单元格（位于最后一行和最后一列）开始，沿着指针回溯穿过动态规划矩阵。每条从最后一个单元格到初始单元格的完整路径就是最小距离对齐。练习 2.7 要求你修改最小编辑距离算法，以存储指针并计算回溯以输出对齐。

虽然我们的示例使用的是简单的 Levenshtein 距离，但图2.17中的算法允许操作具有任意权重。例如，在拼写校正中，替换更有可能发生在键盘上相邻的字母之间。**Viterbi**算法是最小编辑距离的概率扩展。它不是计算两个字符串之间的“最小编辑距离”，而是计算一个字符串与另一个字符串的“最大概率对齐”。我们将在第17章中进一步讨论这一点。

2.9 总结

本章介绍了语言处理中的一个基本工具——**正则表达式**，并展示了如何执行基本的文本规范化任务，包括单词分割和规范化、句子分割以及词干提取。我们还介绍了用于比较字符串的重要算法——**最小编辑距离算法**。以下是我们对这些概念所涵盖的主要内容的总结：

- **正则表达式**是一个强大的模式匹配工具。
- 正则表达式中的基本操作包括符号的**连接**、符号的**分离**（`[]`，`|`）、**计数器**（`*`，`+`，`{n,m}`）、**定位符**（`^`，`$`）和**优先级操作符**（`(`，`)`）。
- **单词分词和规范化**通常通过简单的正则表达式替换或有限自动机的级联完成。
- **Porter算法**是一种简单而有效的进行词干提取的方法，主要是去除词缀。虽然准确性不高，但可能对某些任务有用。
- 两个字符串之间的**最小编辑距离**是将一个字符串编辑为另一个所需的最少操作次数。最小编辑距离可以通过动态规划计算，这也会产生两个字符串之间的对齐。