

2. 正则表达式，文本规范化，编辑距离

用户：我不快乐。

ELIZA：您是否认为来到这里会帮助您得到快乐？

用户：我需要一些帮助，这似乎可以肯定。

ELIZA：如果您得到帮助，对您有什么意义？

用户：也许我可以学会和妈妈相处。

ELIZA：告诉我有关您家庭的更多信息。

用户：我妈妈照顾我。

ELIZA：家庭中的还有其他人照料您吗？

用户：我父亲。

ELIZA：你的父亲？

用户：从某些方面来说，你就像我的父亲。

魏岑鲍姆(1966)

上面的对话来自 **ELIZA(伊丽莎)**，这是一种早期的自然语言处理系统，可以通过模仿罗杰式(Rogierian)心理治疗师的回答与用户进行有限的对话(Weizenbaum, 1966)。ELIZA 是一个非常简单的程序，它使用模式匹配来识别“我需要 X”之类的短语并进行翻译将它们转换为合适的输出，例如“如果您得到 X 对您意味着什么？”。这种简单的技术在该领域中很成功，因为 ELIZA 实际上不需要了解任何模仿罗杰式心理治疗师的知识。

正如 Weizenbaum 所指出的那样，这是为数不多的对话类型之一，听众可以表现得好像对世界一无所知。ELIZA 的模仿人类对话非常成功：许多与 ELIZA 互动的人开始相信，ELIZA 确实理解了他们以及他们的问题，即使在向他们解释了程序的操作之后，许多人仍然相信 ELIZA 的能力(Weizenbaum, 1976)，并且即使在今天，这样的**聊天机器人(chatbots)**还是一个有趣的消遣。

当然，现代的对话主体不只是转移注意力。更重要的是，他们可以回答问题、预定航班或寻找餐馆，依靠这些功能可以对用户的意图进行更为复杂的了解，我们将在第 24 章中看到。尽管如此，为 ELIZA 和其他聊天机器人提供支持的基于模式的简单方法，在自然语言处理中起着至关重要的作用。

我们将从描述文字模式的最重要工具开始：正则表达式。正则表达式可用于：指定我们可能想从文档中提取的字符串、转换上述 Eliza 中的“我需要 X”、定义文档价格表中的诸如\$ 199 或\$ 24.99 之类的字符串。

然后，我们转到一组统称为**文本规范化(text normalization)**的任务，其中文本规范化正则表达式起着重要的作用。规范化文本意味着将其转换为更方便的标准格式。例如，我们将要使用的大多数语言都依赖于首先从运行文本中**分离出单词(tokennizing)**，即**符记化(tokennization)**的任务。英语单词经常用空格将彼此分开标记化，但仅有空格是不够的。尽管“纽约(New York)”和“摇滚(rock 'n' roll)”有时包含空格，但有时仍会被视为大词，而有时我们需要将“I'm”分为“I”和“am”两个词。为了处理推特(tweets)或文字，我们需要作符记像 :) 的表情符号，或像# n1proc 的主题标签。某些语言(例如日语)在单词之间没有空格，因此单词符记化变得更加困难。

文本规范化的另一任务是**词形还原(Lemmatization)**，确定两个单词是否有相同的词根，尽管它们表面不同。例如，单词 sang, sung 和 sings 是动词 sing 的形式。单词 sing 是这些单词的常见词缀，而词缀生成器则将所有这些词映射为 sing。词形还原对于处理诸如阿拉伯语之类的形态复杂的语言至关重要。**词干化(stemming)**是简化词形还原的一种形式，其中我们主要从词的末尾去除后缀。文本规范化还包括句子**分段(segmentation)**：使用句点或感叹号等线索将文本分解为单个句子。

最后，我们需要比较单词和其他字符串。我们将引入一个称为“编辑距离”的度量，该度量基于将一个

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。编辑距离是一种算法, 可应用于从拼写校正到语音识别再到共指解析的整个语言处理过程。

2.1. 正则表达式

正则表达式(Regular Expression, RE)是一种专门用于在文本中搜索字符串的语言, 它是计算机科学标准化领域鲜为人知的成功之一。每种计算机语言, 文字处理器和文本处理工具(例如 Unix 工具 `grep` 或 `Emacs`)都使用此实用语言。形式上, 正则表达式是用于表征一组字符串的代数符号。当我们有一种搜索模式和一整套文本要搜索时, 它们对于搜索文本特别有用。正则表达式搜索功能将搜索 **语料库(corpus)**, 返回所有与模式匹配的文本。语料库可以是单个文档或集合。例如, Unix 命令行工具 `grep` 接受一个正则表达式, 并返回与该表达式匹配的输入文档的每一行。

可以将搜索设计为返回一行中的每个匹配项(如果有多个匹配项), 或者仅返回第一个匹配项。在以下示例中, 我们通常在与正则表达式匹配的模式的确切部分作下划线并仅显示第一个匹配项。我们将显示用斜杠分隔的正则表达式, 但请注意, 斜杠不是正则表达式的一部分。

正则表达式有许多变体。我们将描述扩展的正则表达式; 不同的正则表达式解析器可能仅识别其中的子集, 或者对某些表达式的处理略有不同。使用在线正则表达式测试器是测试表达式并探索这些变化的便捷方法。

2.1.1. 正则表达式的基本模式

最简单的正则表达式是一系列简单字符。要搜索土拨鼠(`woodchuck`), 我们就键入 `/woodchuck/`。表达式 `/Buttercup/` 匹配包含子字符串 `Buttercup`(蝴蝶杯)的任何字符串; 具有该表达式的 `grep` 将返回这样一行字符“`I'm called little Buttercup`”。搜索字符串可包含单个字符(如 `/!/`)或一个字符序列(如 `/urgl/`), 见图 2.1。

RE	Example Patterns Matched
<code>/woodchucks/</code>	“interesting links to <u>woodchucks</u> and lemurs”
<code>/a/</code>	“ <u>M</u> ary Ann stopped by <u>M</u> ona’s”
<code>/!/</code>	“You’ve left the burglar behind again <u>!</u> ” said Nori

图 2-1: 一些简单的正则表达式搜索

正则表达式区分大小写: 小写字母 `s` 与大写字母 `S` 不同(`/s/` 匹配小写字母 `s`, 但不匹配大写字母 `S`)。这意味着模式 `/woodchucks/` 将不匹配字符串 `Woodchucks`。我们可以使用方括号 `[]` 解决此问题。方括号内的字符串指定了要匹配的字符的析取(disjunction)。例如, 图 2.2 显示模式 `/[wW]/` 包含 `w` 或 `W` 的模式匹配。

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

图 2-2: 使用方括号 `[]` 指定字符的析取

正则表达式 `/[1234567890]/` 指定任何一位数字。虽然数字或字母这类字符是表达式中的重要组成部分, 但它们可能会很笨拙(例如, 指定 `/[ABCDEFGHJKLMNOPQRSTUVWXYZ]/` 来表示“任何大写字母”, 就显得不方便)。如果存在与一组字符关联的定义明确的序列, 则可以将方括号与破折号(-)一起使用以指定 **范围(rang)** 中的任何一个字符。模式 `/[2-5]/` 指定字符 `2`、`3`、`4` 或 `5` 中的任何一个。模式 `/[b-g]/` 指定字符 `b`, `c`, `d`, `e`, `f` 或 `g` 中的一个。其他一些示例如图 2.3 所示。

RE	Match	Example Patterns Matched
<code>/[A-Z]/</code>	an upper case letter	“we should call it ‘ <u>D</u> renched Blossoms’”
<code>/[a-z]/</code>	a lower case letter	“ <u>m</u> y beans were impatient to be hoed!”
<code>/[0-9]/</code>	a single digit	“Chapter <u>1</u> : Down the Rabbit Hole”

图 2-3: 用方括号 `[]` 加破折号-来指定范围

方括号也可以通过插入符号 `^` 来指定单个字符不能使用的字符。如果插入符号 `^` 是方括号 `[` 之后的第一个符号，则结果模式将取反。例如，模式 `/[^a]/` 匹配除 `a` 之外的任何单个字符(包括特殊字符)。仅当插入符号是方括号后的第一个符号时，才如此。如果它发生在其他任何地方，通常代表插入符号。图 2.4 显示了一些示例。

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

图 2-4: 插入符号`^`表示否定或仅表示`^`

图注: 请参见以下 re: 反斜杠, 以避免句点。¹

我们该如何谈论可选元素, 就像 `woodchuck` 和 `woodchucks` 中可选的 `s`? 我们不能使用方括号, 因为虽然方括号允许我们说“`s` 或 `S`”, 但不允许我们说“`s` 或 `nothing`”。为此, 我们使用问号 `/?`, 它表示“前面的字符或 `nothing`”, 如图 2.5 所示。

RE	Match	Example Patterns Matched
<code>/woodchucks?/</code>	woodchuck or woodchucks	" <u>woodchuck</u> "
<code>/colou?r/</code>	color or colour	" <u>color</u> "

图 2-5: 问号`?`标记前一个表达式的可选性

我们可以认为问号的意思是“前一个字符的零个或一个实例”。也就是说, 这是一种指定我们想要多少东西的方法, 这在正则表达式中非常重要。例如, 考虑某些绵羊(`sheep`)的语言, 该语言由类似于以下内容的字符串组成:

```
baa!
baaa!
baaaa!
baaaaa! ...
```

该语言由一个 `b` 后跟至少两个 `a` 的字符串组成, 再加上感叹号。允许我们说“某些数量的 `a`”之类的一组操作符是基于星号(`*`)的, 通常称为 **Kleene ***(发音“cleany star”)。Kleene `*` 表示“前一个字符或正则表达式出现零次或多次”。因此, `/a*/` 表示“任何零个或多个 `a` 的字符串”。这将匹配 `a` 或 `aaaaa`, 但也将匹配 `Off Minor`, 因为字符串 `Off Minor` 的 `a` 为零。因此, 匹配一个或多个 `a` 的正则表达式为 `/aa*/`, 表示一个 `a` 后跟零个或多个 `a`。也可以重复更复杂的模式。因此, `/[ab]*/` 的意思是“零个或多个 `a` 或 `b`”(不是“零个或多个右方括号”)。这将匹配 `aaaa` 或 `ababab` 或 `bbbb` 之类的字符串。为了指定多个数字(可用于查找价格), 我们可以扩展 `/[0-9]/`, 这是单个数字的正则表达式。因此, 整数(数字字符串)为 `/[0-9][0-9]*/`。(为什么不只是 `/[0-9]*/?`)

有时必须为数字写两次正则表达式很烦人, 因此可以使用一种较短的方法来指定某个字符的“至少一个”。这就是 **Kleene +**, 表示“前一个字符或正则表达式出现一次或多次”。因此, 表达式 `/[0-9]+/` 是指定“数字序列”的正常方法。因此, 有两种方法可以指定绵羊语言: `/baaa*/` 或 `/baa+!`。一个非常重要的特殊字符是句号(`.`), 它是一个与任何单个字符(回车符除外)匹配的通配符表达式, 如图 2.6 所示。

RE	Match	Example Matched
<code>/beg.n/</code>	any character between beg and n	<u>begin</u> , <u>beg'n</u> , <u>begun</u>

图 2-6: 用句号指定任何字符

¹ 译者注: 为了对图进行题注, 将原书中较长的说明文本, 分成两部分: 第一部分是简短的摘要, 当作图的标签; 第二部分是图注, 用 5 号仿宋字体编辑, 放在图的下方。

通配符通常与 Kleene 星一起使用, 表示“任何字符串”。例如, 假设我们要查找某个特定单词(例如 `aardvark`)出现两次的任意行(line), 那么我们可以使用正则表达式 `/aardvark.*aardvark/` 进行指定。

定位符(anchors)是将正则表达式定位到字符串中特定位置的特殊字符。最常见的定位符是插入符号 `^` 和美元符号 `$`。插入符号 `^` 与行的开头匹配。模式 `/^The /` 与仅在行首的单词 `The` 匹配。因此, 插入符号 `^` 具有三种用途: 匹配行的开头, 表示方括号内的否定, 仅表示插入符号。(什么是允许 `grep` 或 `Python` 知道给定插入符号应该具有的功能的上下文?) 美元符号 `$` 与行尾匹配。因此模式 `$` 是匹配行尾空格的有用模式, `/^The dog \. $/` 匹配仅包含短语 `The dog` 的行。(我们必须在这里使用反斜杠, 因为我们希望 `.` 表示“句点”而不是通配符。)

RE	Match
<code>^</code>	start of line
<code>\\$</code>	end of line
<code>\b</code>	word boundary
<code>\B</code>	non-word boundary

图 2-7: 正则表达式中的定位符

另外还有两个定位符: `\b` 匹配单词边界, 而 `\B` 匹配非边界。因此, `/\bthe\b/` 与单词 `the` 匹配, 但与其他单词不匹配。从技术上讲, 将用于正则表达式的“单词”定义为数字、下划线或字母的任何序列; 这是基于编程语言中“单词”的定义。例如, `/\b99\b/` 将匹配“`There are 99 bottles of beer on the wall`”中的字符串 `99`(因为 `99` 跟在空格后面), 而不匹配“`There are 299 bottles of beer on the wall`”中的字符串 `99`(因为 `99` 跟在数字 `2` 后面)。但它将与 `$99` 中的 `99` 匹配(因为 `99` 跟在美元符号 `$` 后面, 而不是跟在数字、下划线或字母后面)。

2.1.2. 析取, 分组和优先

假设我们需要搜索有关宠物的文字: 也许我们对 `cat` 和 `dog` 特别感兴趣。在这种情况下, 我们可能要搜索字符串 `cat` 或 `dog`。由于我们不能使用方括号来搜索“`cat or dog`”(为什么我们不能说 `/[catdog]/?`), 因此我们需要一个新的操作符, **析取(disjunction)**操作符, 也称为管道符号 `|`。模式 `/cat|dog/` 与字符串 `cat` 或字符串 `dog` 匹配。有时我们需要在更大的序列中使用这个析取操作符。例如, 假设我要为我的堂兄 `David` 搜索有关宠物鱼的信息。如何同时指定 `guppy` 和 `guppies`? 我们不能简单地讲 `/guppy|ies/`, 因为那只会匹配字符串 `guppy` 和 `ies`。这是因为像 `guppy` 这样的序列**优先(precedence)**于析取操作符 `|`。为了使析取操作符仅适用于特定模式, 我们需要使用两个括号(parenthesis)²操作符: 一个是左括号 `(`, 另一个是右括号 `)`。将模式括在括号中, 可以使它像单个字符一样作用于相邻操作符, 例如管道(pipe)符号 `|` 和 Kleene `*`。因此, 模式 `/gupp(y|ies)/` 指定的意思是析取仅适用于后缀 `y` 和 `ies`。

左括号操作符(在我们使用像 Kleene `*` 这样的计数器时也很有用。与 `|` 操作符不同, 默认 Kleene `*` 操作符仅适用于单个字符, 而不适用于整个序列。也许我们有一行, 其行标签的格式为 `Column 1 Column 2 Column3`。表达式 `/(Column [0-9] + *)/` 将不匹配任何数量的列; 相反, 它将匹配单个列, 后跟任意数量的空格! 这里的星号仅适用于其前面的空格 `[]`, 而不适用于整个序列。使用括号, 我们可以编写表达式 `/(Column [0-9] + []*)/` 以匹配单词 `Column`, 后跟数字和可选空格, 整个模式重复零次或多次。

一个操作符可能优先于另一个操作符的思想(要求我们有时使用括号来指定我们的意思), 由正则表达式的**操作符优先级(operator precedence)**层次结构做了形式化。下表给出了正则表达式操作符优先级的顺序, 从最高优先级到最低优先级。

括号	<code>()</code>
计数器	<code>* + ? { }</code>
序列和定位符	<code>the ^my end\$</code>
析取	<code> </code>

² 译者注: `parenthesis` 是圆括号, 简称括号。后文提及的括号基本默认是圆括号。

因此，由于计数器的优先级高于序列，`/the */`与`theeee`匹配，但与`thethe`不匹配。因为序列的优先级高于析取，所以`/the | any /`匹配`the`或`any`而不是`thany`或`theny`。

模式可能会以另一种方式变得模棱两可。当与文本“once upon a time”进行匹配时，请考虑表达式`/[a-z]* /`。由于`/[a-z]* /`匹配零个或多个字母，因此该表达式不能匹配任何内容，或者仅匹配第一个字母`o`、`on`、`onc`或`once`。在这些情况下，正则表达式始终匹配其可以匹配的最大字符串；我们说模式是**贪心的 (greedy)**，不断扩展以覆盖尽可能多的字符串。

但是，可以使用限定符`?`的另一种含义来强制执行非贪心匹配。

操作符`*?`是一个 Kleene 星号，它匹配的文字越少越好。

操作符`+?`是一个 Kleene 加号，它匹配的文字越少越好。

2.1.3. 一个简单例子

假设我们想编写一个 RE 案例来查找英语文章中`the`。一个简单(但不正确)的模式可能是：

```
/the/
```

一个问题是，该模式在句子开头时会漏掉单词，因此将其大写(即`The`)。这可能会导致进入以下模式：

```
/[tT]he/
```

但是，我们仍然会错误地返回包含其他字词的文本(例如，`other`或`theology`)。因此，我们需要指定我们希望实例的两面都带有单词边界：

```
^b[tT]he\b/
```

假设我们想在不用`^b`的情况下执行此操作。我们可能想要这样做，因为`^b`不会将下划线和数字视为单词边界；但是我们可能想在某些情况下找到“`the`”，而在附近也可能有下划线或数字(`the_`或`the25`)。我们需要指定我们想要的实例中，“`the`”的两边都没有字母：

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

但是，这种模式还有一个问题：在开始一行时，找不到“`the`”一词。这是因为正则表达式`[^a-zA-Z]`(我们用来避免嵌入“`the`”的实例)意味着在“`the`”之前必须有一些单个(尽管不是字母)字符。我们可以通过指定在“`the`”之前需要行首字符或非字母字符，并在行尾使用相同字符来避免这种情况：

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

我们刚经历的过程基于修复两种错误：

假阳性(False positives)，我们错误匹配的字符串，例如“`other`”或“`there`”；

假阴性(False negatives)，我们错误漏掉的字符串，例如“`The`”。

在实现语音和语言处理系统时，一次又一次地解决这两种错误。因此，降低应用程序的总体错误率涉及两个反作用的努力：

- 提高精度(最大程度减少假阳性)
- 增加召回率(最大限度地减少假阴性)

我们将在第 4 章中更精确地定义精度和召回率。

2.1.4. 更多操作符

图 2.8 显示了一些通用范围的别名，它们可以主要用于保存键入。除了 Kleene `*`和 Kleene `+`，我们还可以使用显式数字作为计数器，方法是将它们括在大括号中。正则表达式`/ {3} /`的意思是“前一个字符或表达式恰好出现 3 次”。因此，`/a\{24}z /`将匹配“`a`”，再跟 24 个点，最后跟 `z`。

RE	Expansion	Match	First Matches
<code>\d</code>	<code>[0-9]</code>	any digit	Party_of_5
<code>\D</code>	<code>[^0-9]</code>	any non-digit	Blue moon
<code>\w</code>	<code>[a-zA-Z0-9_]</code>	any alphanumeric/underscore	Daiyu
<code>\W</code>	<code>[^\w]</code>	a non-alphanumeric	!!!!
<code>\s</code>	<code>[\r\n\t\f]</code>	whitespace (space, tab)	
<code>\S</code>	<code>[^\s]</code>	Non-whitespace	in_Concord

图 2-8: 常见字符集的别名

RE	Match
*	zero or more occurrences of the previous char or expression
+	one or more occurrences of the previous char or expression
?	exactly zero or one occurrence of the previous char or expression
{n}	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操作符

也可以指定数字范围。因此, `/ {n, m} /` 指定从 `n` 到 `m` 的前一个字符或表达式的出现, `/ {n, } /` 表示前一个表达式至少出现 `n` 次。用于计数的 RE 汇总在图 2.9 中。

最后, 某些特殊字符通过基于反斜杠(`\`)的特殊符号来引用(请参见图 2.10)。其中最常见的是换行(`\n`)和制表符(`\t`)。要引用本身特殊的字符(例如 `.,*,[,and\`), 请在它们前面加反斜杠(例如 `\/, \.\/, *\/, \[/, and \\\`)。

RE	Match	First Patterns Matched
<code>*</code>	an asterisk “*”	“K*A*P*L*A*N”
<code>\.</code>	a period “.”	“Dr. Livingston, I presume”
<code>\?</code>	a question mark	“Why don’t they come and lend a hand?”
<code>\n</code>	a newline	
<code>\t</code>	a tab	

图 2-10: 一些需要反斜杠的字符

2.1.5. 更复杂的例子

让我们尝试一个 RE 威力的重要示例。假设我们要构建一个应用程序来帮助用户购买 Web 上的计算机。用户可能希望“任何具有至少 6 GHz 频率和 500 GB 磁盘空间且价格低于 \$ 1000 的机器”。要进行这种检索, 我们首先需要能够查找 6 GHz 或 500 GB 或 Mac 或 \$ 999.99 的表达式。在本节的其余部分, 我们将为该任务制定一些简单的正则表达式。

首先, 让我们完成价格的正则表达式。这是一个美元符号的 RE, 后跟一串数字:

```
/$[0-9]+/
```

注意, 这里的 `$` 字符与我们前面讨论的行尾函数具有不同的功能。大多数正则表达式解析器都很聪明, 足以意识到这里的 `$` 并不意味着行尾。(作为一个思想实验, 请考虑正则表达式解析器如何从上下文中了解 `$` 的功能。)

现在我们只需要处理小数美元。之后, 我们将添加小数点和两位数:

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

此模式仅允许 \$ 199.99, 但不允许 \$ 199。我们需要使美分成为可选, 并确保我们处于字边界:

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

最后一招! 这种模式允许像 \$ 199999.99 这样的价格, 价格太昂贵了! 我们需要限制美元:

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

磁盘空间如何? 我们将需要再次允许可选部分(5.5 GB): 注意: 用 `?` 使最后的 `s` 可选, 用 `/*` 表示“零个或多个空格”, 因为周围总会有多余的空格:

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

留给读者练习: 修改上述正则表达式, 使其仅匹配大于 500 GB 的情况。

2.1.6. 替代, 捕获组和 ELIZA

正则表达式的重要用途是替换(substitution)。例如, Python 和 Unix 命令(如 vim 或 sed)中使用的替

换操作符 `s / regexp1 / pattern /` 允许将以正则表达式为特征的字符串替换为另一个字符串：

```
s/colour/color/
```

能够引用匹配第一个模式的字符串的特定子部分通常很有用。

假设我们要在文本中的所有整数周围放置尖括号，例如，将 `the 35 boxes` 更改为 `the <35> boxes`。我们希望有一种方法来引用找到的整数，以便我们可以轻松地添加方括号。为此，我们在第一个模式周围加上圆括号，并在第二个模式中使用数字操作符 `\1` 进行引用，形式如下：

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

括号和数字操作符还可以指定某个字符串或表达式在文本中必须出现两次。例如，假设我们正在寻找“`the Xer they were, the Xer they will be`”的模式，我们希望将两个 `X` 约束为相同的字符串。为此，将第一个 `X` 括在括号操作符中，然后将第二个 `X` 替换为数字操作符 `\1`，如下所示：

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

在这里，`\1` 将替换为括号中与第一项匹配的任何字符串。因此，这将匹配“`the bigger they were, the bigger they will be`”，但是不匹配“`the bigger they were, the faster they will be`”。

使用括号将模式存储在内存中称为捕获组(capture group)。每次使用捕获组时(即，括号括起一个模式)，结果匹配项都存储在一个编号寄存器(Register)中。如果匹配两个不同的括号，`\2` 表示匹配第二个捕获组的任何内容。从而

```
/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` 中，第四个捕获组存储在 `\4` 中，依此类推。

因此，括号在正则表达式中具有双重功能。它们用于对术语进行分组，以指定操作符应采用的顺序，并且它们用于捕获寄存器中的内容。有时，我们可能希望使用括号进行分组，但不想在寄存器中捕获结果模式。在这种情况下，我们使用一个非捕获组，该组是通过将命令 `?:` 放在打开的括号后，以 `(?: pattern)` 形式指定的。

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

可以匹配“`some cats like some cats`”，但不能匹配“`some cats like some a few`”。

替换和捕获组在实现简单的聊天机器人(如 `ELIZA`)中非常有用(Weizenbaum, 1966)。回想一下，`ELIZA` 通过进行如下对话来模拟罗杰氏心理学家：

```
用户1：男人们都一样。
ELIZA1：以什么方式不一样？
用户2：他们总是在骚扰我们。
ELIZA2：您能否想到一个具体的例子。
用户3：好，我的男朋友让我来这里。
ELIZA3：您的男朋友来了。
用户4：他说我很多时候都很沮丧。
ELIZA4：很抱歉听到您感到沮丧。
```

`ELIZA` 的工作方式是使用一系列或级联的正则表达式替换，每个替换都匹配和更改输入行的某些部分。输入行首先是大写的。然后，第一个替换将“`MY`”的所有实例更改为“`YOUR`”，将“`I'M`”更改为“`YOU ARE`”，依此类推。下一组替换匹配并替换输入中的其他模式。这里有些例子：

```
s/. * I'M (depressed|sad) . * /I AM SORRY TO HEAR YOU ARE \1/
s/. * I AM (depressed|sad) . * /WHY DO YOU THINK YOU ARE \1/
s/. * all . * /IN WHAT WAY/
s/. * always . * /CAN YOU THINK OF A SPECIFIC EXAMPLE/
```

由于可以将多个替换应用于给定输入，因此将为替换分配一个等级并按顺序应用。创建模式是练习 2.3 的主题，我们将在第 24 章返回 `ELIZA` 体系结构的详细信息。

2.1.7. 前瞻断言

最后, 有时我们需要预测未来: 在文本中向前看, 看是否有某些模式匹配, 但不提前匹配光标(cursor), 以便我们可以在模式出现时对其进行处理。

这些**前瞻(lookahead)**断言使用上一节中针对非捕获组使用的 (? 句法。如果模式出现, 则操作符(? = pattern)为 true, 但是为**零宽度(zero-width)**, 即匹配指针不前进。操作符(? ! pattern)仅在模式不匹配时返回 true, 但再次为零宽度且不使光标前移。当我们解析某些复杂的模式但要排除特殊情况时, 通常使用负向前瞻。例如, 假设我们要在一行的开头匹配任何以“Volcano”开头的单词。我们可以使用负向前瞻进行此操作:

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

2.2. 单词

在讨论处理单词之前, 我们需要确定什么才是单词。让我们首先看一个特定的**语料库(corpus, 其复数为 corpora)**, 它是计算机可读的文本或语音集合。例如, 布朗(Brown)语料库是一百万个单词的样本集, 这些样本来自 500 种不同体裁(报纸, 小说, 非小说, 学术等)的英语书面文本, 于 1963–64 年在布朗大学完成集合(Kucera and Francis, 1967)。下面的布朗句子中有多少个单词?

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

如果我们不将标点符号视为单词, 则此句子包含 13 个单词; 如果我们将标点符号视为单词, 则包含 15 个单词。我们是否将句点(“.”), 逗号(“,”)等视作单词取决于任务。标点符号对于查找事物的边界(逗号, 句号, 冒号)和识别含义的某些方面(问号, 感叹号, 引号)至关重要。对于某些任务, 例如词类标记或语法分析或语音合成, 有时我们将标点符号视为单独的单词。

电话总机(Switchboard)语料库收集了 1990 年代初陌生人之间的美式英语电话会话; 它包含 2430 个会话, 每个会话平均 6 分钟, 总计 240 个小时的语音和大约 300 万个单词(Godfrey 等人, 1992)。这样的口语语料没有标点符号, 但是在定义单词方面确实带来了其他麻烦。看看电话总机的**语段(utterance)**, 它是句子的口头相关词:

I do uh main- mainly business data processing

这种语段有两种**不流利(disfluency)**。断开的单词 main-称为片段, 诸如 uh 和 um 之类的单词称为填充符或**填充停顿符(filled pause)**。我们应该把这些当作词语吗? 同样, 这取决于应用程序。如果我们要建立语音转录系统, 我们可能最终希望去除不流利。

但是我们有时也会保持不流利。像 uh 或 um 这样的不流利, 在语音识别中对预测即将到来的单词是有帮助的, 因为它们可能表明说话者正在重新启动从句或构想, 因此对于语音识别, 它们被视为常规单词。因为人们使用不同的不流利, 它们也可能是识别说话者的线索。实际上, Clark 和 Fox Tree(2002)表明, uh 和 um 有不同的含义。您认为他们是什么?

大写的符记 They 和非大写的符记 they 是一个词吗? 在某些任务(语音识别)中, 它们被集中在一起, 而对于词类(POS)或命名实体标记(NET), 大写是一个有用的特性并被保留下来。

cats 和 cat 之类的变形形式怎么样? 这两个词具有相同的词元 cat, 但具有不同的词形。**词元(lemma)**是一组具有相同词干、相同主要词类和相同词义的词汇形式。**词形(wordform)**是单词的全部变形或派生形式。对于形态复杂的语言(例如阿拉伯语), 我们经常需要处理词形还原。但是, 对于许多英语任务来说, 词形就足够了。

英语里有多少个单词? 要回答这个问题, 我们需要区分两种谈论单词的方式。**类型(types)**是语料库中不同单词的数量; 如果词汇表中的单词集为 V, 则类型数量为词汇表大小|V|。**符记(token)**是运行(running)单词的总数 N。如果我们忽略标点符号, 则下面的布朗句子有 16 个符记和 14 种类型:

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

当我们谈论语言中的单词数时, 通常是指单词类型。

图 2.11 显示了从一些流行的英语语料库中计算出的类型和符记的大致数量。我们查看的语料库越大,

我们发现的类型越多，实际上类型数量 $|V|$ 和符记数量 N 之间的这种关系，以其发现者(分别在语言学和信息检索中)命名为 Herdan 定律(Herdan 1960)或 Heaps 定律(Heaps 1978)。它显示在等式 2.1 中，其中 k 和 β 为正常数，且 $0 < \beta < 1$ 。

$$|V|=kN^\beta \quad (2.1)$$

β 的值取决于语料库的大小和体裁，但至少对于图 2.11 中的大语料库， β 的范围是 0.67 至 0.75。大概可以说，文本的词汇量增长速度明显快于单词长度的平方根。

语言中单词数量的另一种度量是词元数量而不是类型。词典可以帮助确定词元数量。词典条目或黑体字形式是词元数量的上限(因为某些词元具有多种黑体字形式)。1989 版的牛津英语词典有 615,000 个条目。

Corpus	Tokens = 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

图 2-11: 某些英语语料库的类型和符记的大致数量

图注: 最大的 Google N-grams 语料库包含 1300 万个类型，其中仅包含出现 ≥ 40 次的类型，因此真实数字会更大。

2.3. 语料库

文字不是凭空出现的。我们学习的任何一段文字是由一个或多个特定的讲话者或作家，用特定语言的特定方言，在特定的时间，特定的地点，用于特定的功能。

语言变化可能是最重要的方面。当 NLP 算法适用于多种语言时，它们最有用。根据在线民族志目录 (Simons 和 Fennig, 2018)，撰写本文时，世界上共有 7097 种语言。在多种语言上测试算法非常重要，尤其是在具有不同属性的语言上；相比之下，目前不幸的是 NLP 算法仅以英语开发或测试的趋势(Bender, 2019)。即使算法的开发超出英语，它们也倾向于针对大型工业化国家的官方语言开发(中文，西班牙文，日文，德文等)，但我们不想将工具限制为仅这几种语言。此外，大多数语言还具有多种变体，经常在不同地区或不同社会群体中使用。因此，例如，如果我们正在处理的文本使用**非裔美国语言(African American Language, AAL)**的特征，则我们必须使用具有这些变体特征的 NLP 工具。Twitter 帖子可能使用非裔美国人演讲者经常使用的功能，例如 *iont*(即**主流美国英语(Mainstream American English, MAE)**中的 *I don't*)之类的构造，或 *talmbout* 相对应主流美国英语中的 *talk about*，都是两个影响分词的示例(Blodgett 等人 2016, Jones 2015)。

演讲者或作家在一次交流中使用多种语言也很普遍，这种现象称为**代码转换(code switching)**。代码转换在世界范围内极为普遍。这是显示西班牙语和(音译)印地语与英语的代码转换的示例(Solorio 等人 2014, Jurgens 等人 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 worry ... but dherya rakhe
["he was and will remain a friend ... don ' t worry ... but have faith "]

变化的另一个维度是**体裁(genre)**。我们的算法必须处理的文本可能来自新闻专线、小说或非小说类书籍、科学文章、维基百科(Wikipedia)或宗教文本。它可能来自于口头类型，如电话谈话、商务会议、警察穿戴式摄像机、医疗采访或电视节目或电影的文本。它可能来自工作场合，比如医生的笔记、法律文本或议会或国会程序。

文本也反映了作者(或说话者)的人口统计学特征:他们的年龄、性别、种族、社会经济阶层都可以影响我们正在处理的文本的语言属性。

最后，时间也很重要。语言会随时间而变，对于某些语言，我们拥有来自不同历史时期的大量文本。

由于语言的位置如此, 在开发用于从语料库进行语言处理的计算模型时, 重要的是要考虑谁在什么背景下, 出于什么目的制作了语言。数据集(dataset)的用户如何知道所有这些详细信息? 最好的方法是使语料库创建者为每个语料库建立一个**数据表 (datasheet)** (Gebru 等人 2020) 或**数据声明 (statement)** (Bender 和 Friedman 2018)。数据表指定数据集的属性, 例如:

动机: 为什么要收集语料库、由谁收集、由谁提供资金?

情况: 在什么时候、什么情况下写/说过文本? 例如, 有任务吗? 语言最初是口语对话, 编辑文本, 社交媒体交流, 独白还是对话?

语言种类: 语料库使用哪种语言(包括方言/地区)?

演讲者个人背景: 例如, 文字作者的年龄或性别是什么?

收集过程: 数据有多大? 如果是子样本, 则如何采样? 是否经同意收集数据? 数据是如何进行预处理的, 可以使用哪些元数据?

注释过程: 注释是什么, 注释者的个人背景是什么, 他们是如何训练的, 数据是如何注释的?

发行: 有版权或其他知识产权限制吗?

2.4. 文本规范化

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。通常, 至少要执行三个任务, 作为任何规范化过程的一部分:

1. 符记化(分段)单词
2. 规范单词格式
3. 分割句子

在下一节中, 我们将逐步完成所有这些任务。

2.4.1. 用于粗糙符记化和规范化的 Unix 工具

让我们从一个简单的、甚至有点天真的单词标记化和规范化(以及频率计算)的版本开始, 受 Church(1994)的启发, 它只能在一个 UNIX 命令行中用英语完成。我们将使用一些 Unix 命令: `tr`, 用于系统地更改输入中的特定字符; `sort`, 按字母顺序对输入行进行排序; 和 `uniq`, 它折叠并计算相邻的相同行。

例如, 让我们从一个文本文件 `sh.txt` 中的莎士比亚“complete words”开始。通过将非字母字符的每个序列更改为换行符, 我们可以使用 `tr` 符记化单词(“A-Za-z”表示字母, `-c` 选项补全为非字母, `-s` 选项压缩所有字母) 序列成一个字符):

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

该命令的输出将是:

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

现在每行有一个单词, 我们可以对行进行排序, 然后将它们传递给 `uniq -c`, 它将折叠并计算它们:

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

具有以下输出:

```

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折叠为小写字母：

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

其输出是：

```

14725  a
97     aaron
1      abaissez
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见单词。

`-n` 选项的排序方式是按数字而不是字母顺序排序；

`-r` 选项的意思是按相反顺序(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，与其他语料库一样，莎士比亚语中最常见的单词是短功能单词，例如冠词，代词，介词：

```

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

```

...

此类 Unix 工具可以非常方便地为任何语料库建立快速的字数统计。

2.4.2. 单词符记化

上面的简单 UNIX 工具可以很好地获取单词的粗略统计信息，但是对于**符记化(tokenization)** (将运行中的文本分割成单词的任务)来说，通常更复杂的算法是必需的。

尽管 Unix 命令序列仅删除了所有数字和标点符号, 但对于大多数 NLP 应用程序, 我们需要将其保留在标记中。我们经常想将标点符号作为一个单独的符记来分离; 逗号是解析器的有用信息, 句点有助于指示句子边界。但是我们经常希望保留内部出现的标点符号, 例如 m.p.h., Ph.D., AT&T 和 cap'n。特殊字符和数字将需要保留在价格(\$ 45.55)和日期(06/02/01)中; 我们不想将该价格细分为“45”和“55”的单独符记。还有 URL(<http://www.stanford.edu>), Twitter 标签 (#nlproc) 或电子邮件地址 (someone@cs.colorado.edu)。

数字表达式也引入了其他复杂的情况;

虽然逗号通常出现在单词边界, 但在英语中, 逗号用于数字内部, 每三个数字使用一次:555,500.50。在这一点上, 语言以及因此而产生的符记化要求是不同的;相比之下, 许多欧洲大陆语言, 如西班牙语、法语和德语, 使用逗号来标记小数点, 在英语中放置逗号的地方使用空格(有时是句号), 例如, 555 500,50。

标记器还可以用省略符号来扩展附着词(clitic)的缩略形式, 例如, 将“what're”转换为两个符记“what are”, 将“we're”转换为两个字符“we are”。附着词是一个单词的一部分, 不能独立存在, 只能在附加到另一个单词时出现。一些这样的缩略以其他字母语言出现, 包括法语的文章和代词(j'ai, l'homme)。

取决于应用程序, 符记化算法还可以将多词表达式(例如 New York 或 rock'n'roll)符记化为单个标记, 这需要某种类型的多词表达字典。因此, 符记化与命名实体识别, 检测名称, 日期和组织的任务密切相关(第 8 章)。

一种常用的符记化标准称为宾州树库(Penn Treebank)符记化标准, 用于语言数据联盟(LDC)发布的已解析的语料库(树库), 该语言库是许多有用数据集的来源。该标准将附着词分隔开(doesn't 变成 does n't), 将带连字符的单词保持在一起, 并分隔所有标点符号(为了节省空间, 我们在符记之间显示可见的空格 ' _ ', 尽管换行是更常见的输出):

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 _ " _

实际上, 由于符记化需要在任何其他语言处理之前运行, 因此它必须非常快。因此, 符记化的标准方法是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为非常有效的有限状态自动机。例如, 图 2.12 显示了一个基本正则表达式的示例, 该示例可用于基于 Python 的自然语言工具包(NLTK)的 nltk.regexp_tokenize 符记化功能进行符记化(Bird 等人 2009; <http://www.nltk.org>)

```
>>> text = 'That U.S.A. poster-print costs $12.40...'
>>> pattern = r"""(?x)
... ([A-Z]\.)+           # abbreviations, e.g. U.S.A.
... | \w+(-\w+)*        # words with optional internal hyphens
... | \$?\d+(\.\d+)?%?   # currency and 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: NLP 工具包中 RE 符记化的 Python 跟踪

图注: NLTK 基于 Python 的 NLP 工具包中 RE 符记化的 Python 跟踪。为可读性而注释; (?x) 冗长标志告诉 Python 删除注释和空格。摘自 chapter 3 of Bird 等人(2009)

精心设计的确定性算法可以处理产生的歧义, 例如, 省略符号在用作所有格标记(如书的封面)、引号(如 “The other class”)或附着词(如 they're)时需要进行不同的符记化。

在诸如中文, 日文和泰文这样的语言中, 单词符记化更为复杂, 这些语言不使用空格来标记潜在的单词边界。例如, 在中文中, 单词由字符组成(中文称为汉字(hanzi))。每个字符通常代表一个意思单元(称为语素), 并且可以作为一个音节发音。单词的平均长度约为 2.4 个字符。但是, 决定什么才是中文中的单词是很复杂的。例如, 考虑以下句子:

(2.4) 姚明进入总决赛

“Yao Ming reaches the finals”

正如 Chen 等人(2017)指出的那样, 这可以被视为 3 个词(“中国树库”分割):

(2.5) 姚明 进入 总决赛

YaoMing reaches finals

或作为 5 个单词(“北京大学”分词):

(2.6) 姚 明 进入 总 决赛

Yao Ming reaches overall finals

最后, 在中文中, 完全可以完全忽略单词并使用字符作为基本元素, 将句子视为一系列 7 个字符:

(2.7) 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 任务而言, 采用字符而不是单词作为输入效果更好, 因为对于大多数应用程序而言, 字符处于合理的语义级别, 相比之下, 这导致词汇量巨大, 包含大量非常罕见的单词(Li 等人 2019)。

但是, 对于日语和泰语, 字符的单位太小, 因此需要用于分词(word segmentation)的算法。对于在少数情况下需要单词而不是字符边界的情况, 这些功能对于中文也很有用。这些语言的标准分割算法使用经过人工分割训练集的有监督机器学习训练的神经序列模型; 我们将在第 8 章和第 9 章中介绍序列模型。

2.4.3. 符记化的字节对编码

对文本进行符记化还有第三种选择。我们可以使用数据自动地告诉我们应该使用什么符记, 而不是将符记定义为单词(无论是由空格或更复杂的算法分隔)或字符(如在中文中)。这在处理未知单词(语言处理中的重要问题)时特别有用。

正如我们将在下一章中看到的那样, NLP 算法通常会从一个语料库(一个训练语料库)中学习一些关于语言的事实, 然后使用这些事实来决定一个单独的测试语料库及其语言。因此, 如果我们的训练语料库包含单词“low”, “new”, “newer”而不是“lower”, 又如果“lower”一词出现在我们的测试语料库中, 则我们的系统将不知道如何处理。

为了处理这个未知的单词问题, 现代的符记器经常自动地归纳出一组符记, 其中包含比单词更小的字符, 称为子单词(subword)。子单词可以是任意的子字符串, 也可以是带有含义的单位, 例如词素-est 或 -er。(词素是一种语言中最小的含意单元; 例如, 单词“unlikeliest”的词素是 un-, likely 和 -est。)在现代符记化方案中, 大多数符记是单词, 但是某些符记是经常出现的语素或其他子词, 例如 -er。因此, 每个看不见的单词(例如“lower”)都可以由已知的子单词单元(例如 low 和 er)的某些序列表示, 或者在必要时甚至表示为单个字母的序列。

大多数符记化方案有两个部分: 符记学习器和符记分段器。符记学习器采用原始的训练语料库(有时通过空格将其粗略地分离成单词), 并产生一个词汇集(一组符记)。符记分段器采用原始测试语句, 并将其分段为词汇表中的符记。三种算法被广泛使用: 字节对编码(Sennrich 等人 2016), 一元语法(unigram)语言建模(Kudo, 2018)和 WordPiece(Schuster 和 Nakajima, 2012); 还有一个 SentencePiece 库, 其中包含三个算法中的前两个的实现(Kudo 和 Richardson, 2018)。

在本节中, 我们介绍三种中最简单的一种, 即字节对编码(BPE)算法(Sennrich 等人 2016), 见图 2.13。

```

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 til k times
  tL, tR ← Most frequent pair of adjacent tokens in C
  tNEW ← tL + tR                                           # make new token by concatenating
  V ← V + tNEW                                               # update the vocabulary
  Replace each occurrence of tL, tR in C with tNEW         # and update the corpus
return V

```

图 2-13: BPE 算法中的符记学习器部分

图注: 将语料库分解成单个字符或字节, 通过迭代合并符记来学习词汇。图改编自 Bostrom 和 Durrett (2020)。

BPE 符记学习器从一个词汇开始, 该词汇只是所有单个字符的集合。然后, 它检查训练语料库, 选择最经常相邻的两个符号(例如“**A**”, “**B**”), 在词汇表中添加新的合并符号“**AB**”, 并用新“**AB**”替换语料库中的每个相邻的“**A**”, “**B**”。它继续计数和合并, 创建越来越长的新字符串, 直到完成 k 个合并为止, 创建了 k 个新颖的符记。因此, k 是算法的参数。产生的词汇表由原始字符集和 k 个新符号组成。

该算法通常在单词内部运行(不跨单词边界合并), 因此输入语料库首先用空格分隔, 以给出一组字符串, 每个字符串对应于一个单词的字符, 再加上一个特殊的单词结尾符号_及其计数。让我们看看它在以下由 18 个单词符记组成的小输入语料库上的操作, 每个单词都有计数(单词 **low** 出现 5 次, 单词 **new** 出现 6 次, 依此类推), 其起始词汇为 11 个字母:

corpus	vocabulary
5 low_	_ , d, e, i, l, n, o, r, s, t, w
2 lowest_	
6 newer_	
3 wider_	
2 new_	

BPE 算法首先对所有相邻符号对进行计数: 最频繁的是 **er**, 因为它出现在“**newer**”(频率为 6)和“**wider**”(频率为 3)中, 总共出现了 9 次³。然后, 我们合并这些符号, 将 **er** 视为一个符号, 然后再次计数:

corpus	vocabulary
5 low_	_ , d, e, i, l, n, o, r, s, t, w, er
2 lowest_	
6 newer_	
3 wider_	
2 new_	

最常见的一对是 **er_**, 我们要归并它, 系统已经学习到, 单词末的 **er** 应该有一个符记, 表示为 **er_**:

corpus	vocabulary
5 low_	_ , d, e, i, l, n, o, r, s, t, w, er_
2 lowest_	
6 newer_	
3 wider_	
2 new_	

接下来的 **ne**(总数为 8)合并为 **ne**:

corpus	vocabulary
5 low_	_ , d, e, i, l, n, o, r, s, t, w, er_ , ne
2 lowest_	
6 newer_	
3 wider_	
2 new_	

如果我们继续, 下一个合并是:

³ 注意, 可以有联系; 我们可以选择先归并 **r**, 因为它的频率也是 9。

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_

学习完词汇后，符记解析器将用于符记化测试语句。符记解析器仅在测试数据上运行，然后以我们学习它们的顺序贪心地从训练数据中学到了合并。(因此，测试数据中的频率不起作用，只是训练数据中的频率起作用)。因此，首先我们将每个测试句子单词分割成字符。然后我们应用第一个规则：将测试语料库中 `er` 的每个实例替换为 `r`；然后是第二条规则：将测试语料库中 `er_` 的每个实例替换为 `er_`，依此类推。最后，如果测试语料库包含单词“`newer_`”，它将被符记化为一个完整的单词。但是，一个新的(未知的)单词，如“`lower_`”，将被合并到 `low` 和 `er_` 两个符记中。

当然，在实际算法中，BPE 在非常大的输入语料库上运行成千上万次合并。结果是大多数单词将被表示为完整的符号，而只有非常罕见的单词(和未知单词)才必须由它们的部分(parts)来表示。

2.4.4. 单词规范化，词形还原和词干化

单词规范化(normalization)的任务是将单词/符记置于标准格式中，为具有多种形式的单词(如 `USA` 和 `US`、或 `uh-huh` 和 `uhhuh`)选择单一标准形式。尽管在标准化过程中丢失了拼写信息，但这种标准化可能是有价值的。对于关于 `US` 的信息检索或信息提取，我们可能希望看到来自文档的信息，无论它们是否提到了 `US` 或 `USA`。

大小写折叠(case folding)是另一种规范化。将所有内容都映射为小写意味着 `Woodchuck` 和 `woodchuck` 被相同地表示，这对于许多任务(例如信息检索或语音识别)的概括非常有帮助。相比之下，对于情感分析和其他文本分类任务，信息提取和机器翻译，大小写可能会很有帮助，并且通常不进行大小写折叠。这是因为，保持例如“美国是国家(`US the country`)”和“我们是代词(`us the pronoun`)”之间的差异，概括地说，可能会超过为其他词提供大小写折叠的优势。

对于许多自然语言处理情况，我们还希望单词的两种形态上不同的形式表现相似。例如，在 `Web` 搜索中，某些人可能会键入字符串 `woodchucks`，但有用的系统可能还希望返回提及不带 `s` 的 `woodchuck` 的页面。这在形态复杂的语言(例如俄语)中尤其常见，例如，`Moscow`(莫斯科)一词在 `Moscow`, `of Moscow`, `to Moscow` 等词组中都有不同的结尾。

词形还原(lemmatization)是确定两个词尽管表面存在差异但具有相同词根的任务。`am`, `are` 和 `is` 这三个词具有共同的词元 `be`。`dinner` 和 `dinners` 这两个词都具有词元 `dinner`。将这些形式中的每一个词化为同一个词元，将使我们能够找到所有提及的俄语单词，例如莫斯科。像 `He is reading detective stories` 这样的句子的词元化形式就是 `He be read detective story`。

词形还原是如何完成？最复杂的词形还原方法包括单词的完整形态解析。形态学是这样一种方式的研究，在这种方式中，词是由更小的承载意义的单位——**语素(morpheme)**构成的。可以将词素分为两大类：**词干(stem)**(词的中心词素，提供主要含义)和**词缀(affix)**(添加各种“附加”含义)。因此，例如，单词 `fox` 由一个词素(词素 `fox`)组成，而单词 `cats` 由两个词素组成：词素 `cat` 和词素 `-s`。形态学解析器将像 `cats` 这样的单词解析成两个语素 `cat` 和 `s`，或者将西班牙语单词 `amaren`(即‘if in the future they would love’)解析成语素 `amar`(即‘to love’)，其形态学特征为 3PL 和未来虚拟语气。

Porter(波特)词干分析器

词形还原算法可能很复杂。出于这个原因，我们有时会使用一种更简单但更粗糙的方法，主要包括去掉词尾词缀。这种朴素的形态分析版本称为**词干分析(stemming)**。最广泛使用的词干提取算法之一是 Porter(1980)。Porter **词干分析器(stemmer)**应用于如下段落：

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.

产生以下词干输出:

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, 以**级联(cascade)**的形式串行运行, 其中每次传递的输出作为下一个传递的输入; 以下是一些规则的示例:

ATIONAL	→ ATE (e.g., relational → relate)
ING	→ ε if stem contains vowel (e.g., motoring → motor)
SSES	→ SS (e.g., grasses → grass)

有关 Porter 词干分析器的详细规则列表以及代码(使用 Java, Python 等), 可以在 Martin Porter 的主页上找到; 另请参阅原始论文(Porter, 1980)。当我们需要折叠同一词元的不同变体时, 简单的词干分析器可能会很有用。然而, 如下表所示(Krovetz, 1993), 他们的确会犯概括过度 and 概括不足的错误。

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

2.4.5. 句子分割

句子分割(sentence segmentation)是文本处理的另一个重要步骤。把一篇文章切分成句子最有用的线索就是标点符号, 比如句号、问号和感叹号。问号和感叹号是比较明确的句子边界标记。另一方面, 句号则更为模糊。句点字符“.”在句子边界标记和缩写标记(例如 Mr.或 Inc.)之间是有歧义的。你刚刚读到的上一个句子展示了这种歧义的一个更复杂的情况, 在这个句子中, Inc.的最后一个句号同时标记了一个缩写和句子边界标记。因此, 句子符记化和单词符记化可以共同解决。

通常, 句子符记化方法的工作方式是首先(基于规则或机器学习)确定句点是单词的一部分还是句子边界标记。缩写字典可以帮助确定句点是否是常用缩写的一部分; 字典可以是手工构建的或机器学习的(Kiss 和 Strunk, 2006 年), 最后的句子拆分器也可以。例如, 在 Stanford CoreNLP 工具箱中(Manning 等人, 2014), 句子拆分是基于规则的, 这是符记化的确定性结果。当句子结尾的标点符号(., !, or ?)尚未与其他字符组合成符记(例如缩写或数字)时, 该句子结束(可选地, 后面加上其他最终引号或括号)。

2.5. 最小编辑距离

许多自然语言处理都与测量两个字符串的相似程度有关。例如, 在拼写更正中, 用户输入了一些错误的字符串(例如, **graffe**), 我们想知道用户的意思。用户可能想用 一个与 **graffe** 类似的词。在候选的相似单词中, 单词 **giraffe**(与 **graffe** 只有一个字母不同)直觉上似乎比 **grail** 或 **graf**(它们有更多的字母不同)更相似。另一个示例来自于**共指(coreference)**, 即确定两个字符串(例如以下字符串)是否引用同一实体的任务:

Stanford President Marc Tessier-Lavigne

Stanford University President Marc Tessier-Lavigne

同样, 这两个字符串非常相似(只有一个单词不同), 似乎是有用的证据, 可以决定它们可能是相关的。

编辑距离给了我们一种方法来量化这两种关于字符串相似性的直觉。更正式地说, 两个字符串之间的**最小编辑距离(minimum edit distance)**定义为将一个字符串转换为另一个字符串所需的最小编辑操作数量(如插入、删除、替换等操作)。

例如, “intention”与“execution”之间的差距为 5(删除 i, 将 e 替换为 n, 将 x 替换为 t, 插入 c, 将 u 替换为 n)。通过查看最重要的字符串距离可视化, 可以更容易地看到两个字符串之间的**对齐(alignment)**, 如图 2.14 所示。给定两个序列, 对齐是两个序列的子串之间的对应关系。因此, 我们说我与空字符串对

齐，N 与 E 对齐，依此类推。在对齐的字符串下面是另一种表示形式：一系列表示操作列表的符号，这些操作列表用于将顶部字符串转换为底部字符串：d 表示删除，s 表示替换，i 表示插入。

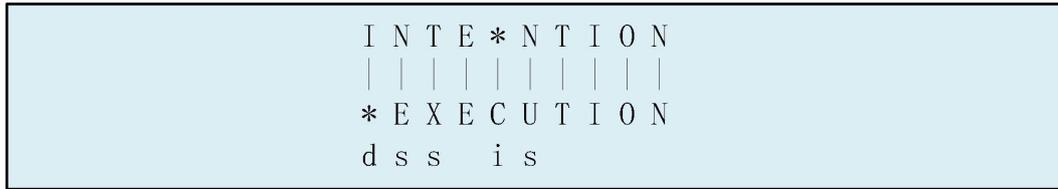


图 2-14：表示两个字符串之间的最小编辑距离

图注：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。两个序列之间的 Levenshtein 距离是最简单的加权因子，其中三个操作的每一个的成本为 1 (Levenshtein, 1966)--我们假设以字母代替其自身(例如，用 t 代替 t)的成本为零。Levenshtein 的 intention 和 execution 之间的距离为 5。Levenshtein 还提出了他的指标的替代版本，其中每个插入或删除的成本均为 1，并且不允许替换。(这等效于允许替换，但是给每个替换以 2 为代价，因为任何替换都可以用一个插入和一个删除来表示)。使用此版本，intention 和 execution 之间的 Levenshtein 距离为 8。

2.5.1. 最小编辑距离算法

我们如何找到最小的编辑距离?我们可以把这看作是一个搜索任务，在这个任务中，我们搜索从一个字符串到另一个字符串的最短路径(编辑序列)。

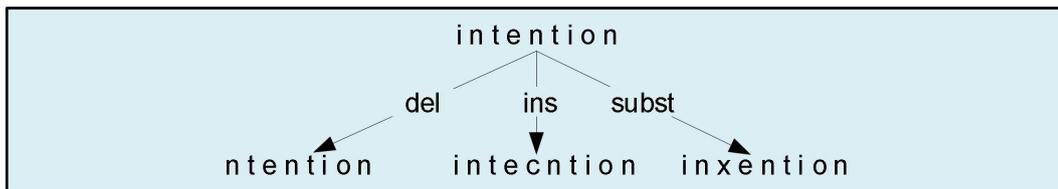


图 2-15：寻找编辑距离被视为一个搜索问题

所有可能的编辑空间都很大，因此我们不能天真地进行搜索。然而，许多不同的编辑路径最终将以相同的状态(字符串)结束，因此不必重新计算所有这些路径，我们可以在每次看到一个状态时只记住它的最短路径。我们可以通过使用[动态规划\(dynamic programming\)](#)来做到这一点。动态规划是一类算法的名称，该算法首先由 Bellman(1957)提出，该算法采用表驱动方法通过将解决方案与子问题相结合来解决问题。自然语言处理中一些最常用的算法利用了动态规划，例如 Viterbi 算法(第 8 章)和 CKY 算法进行解析(第 13 章)。

动态规划问题的直觉是，一个大问题可以通过适当地结合不同子问题的解来解决。考虑转换后的单词的最短路径，该路径代表图 2.16 中所示的字符串 intention 和 execution 之间的最小编辑距离。想象一些字符串(可能是 exention)在这个最优路径上(无论它是什么)的情景。动态规划的直觉是，如果在最佳操作列表中有 exention，则最佳序列还必须包括从 intention 到 exention 的最佳路径。为什么?如果从 intention 到 exention 有一条较短的路径，那么我们可以使用它来代替，从而导致更短的整体路径，最优序列不会是最优的，从而导致矛盾。

最小编辑距离算法被 Wagner 和 Fischer(1974)命名，但被许多人独立发现(见第 8 章的历史说明部分)。首先定义两个字符串之间的最小编辑距离。给定两个字符串，长度为 n 的源字符串 X 和长度为 m 的目标字符串 Y ，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。因此 X 和 Y 之间的编辑距离为 $D[n, m]$ 。

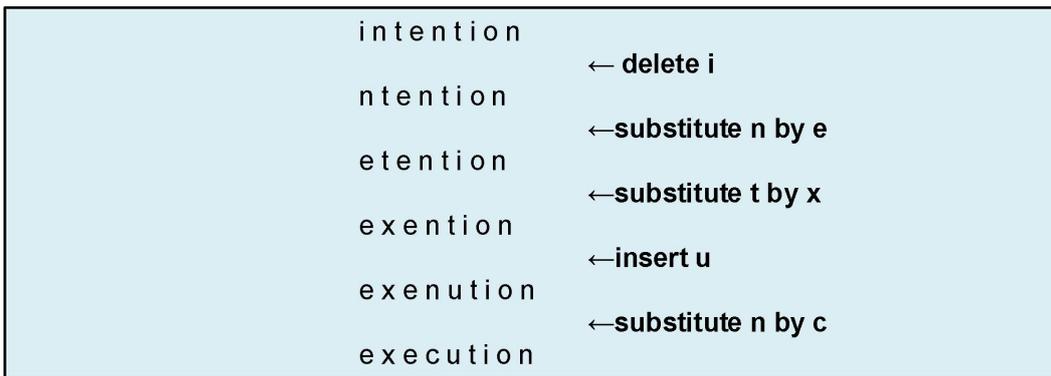


图 2-16: 从 intention 到 execution 的路径

我们将使用动态规划来自底向上计算 $D[n,m]$, 并结合子问题的解决方案。在基本用例中, 源子字符串长度为 i , 但目标字符串为空, 则从 i 个字符到 0 个字符需要 i 个删除。如果目标子串的长度为 j , 则从 0 个字符到 j 个字符需要插入 j 个字符。计算出较小的 i, j 的 $D[i, j]$ 之后, 我们将根据先前计算出的较小值来计算较大的 $D[i, j]$ 。 $D[i, j]$ 的值通过采用到达矩阵的三个可能路径中的最小值来计算:

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

假设 Levenshtein 距离的版本, 插入和删除的成本分别为 1 ($\text{ins-cost}(\cdot) = \text{del-cost}(\cdot) = 1$), 替换的成本为 2 (除了替换相同字母的成本为 0), 则 $D[i, j]$ 的计算为:

$$D[i, j] = \min \begin{cases} D[i-1, j] + 1 \\ D[i, j-1] + 1 \\ D[i-1, j-1] + \begin{cases} 2; & \text{if } source[i] \neq target[j] \\ 0; & \text{if } source[i] = target[j] \end{cases} \end{cases} \quad (2.8)$$

算法总结如图 2.17 所示; 图 2.18 是公式 2.8 中 Levenshtein 版本应用该算法计算从 intention 到 execution 距离的结果。

```

function MIN-EDIT-DISTANCE(source, target) returns min-distance
  n ← LENGTH(source)
  m ← LENGTH(target)
  Create a distance matrix distance[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

图注: 这是动态规划算法类别的示例。各种费用可以是固定的 (例如 $\forall x, \text{ins-cost}(x) = 1$), 也可以特定于字母 (以模拟某些字母比其他字母更容易插入的事实)。我们假设以字母代替自身没有成本 ($\text{sub-cost}(x, x) = 0$)。

知道最小编辑距离对于查找潜在的拼写错误更正等算法很有用。但是编辑距离算法在另一方面也很重要；通过一个小的改变，它还可以在两个字符串之间提供最小的成本对齐。对齐两个字符串在整个语音和语言处理中很有用。在语音识别中，最小编辑距离对齐用于计算单词错误率(第 26 章)。对齐在机器翻译中扮演重要角色，其中平行语料库(带有两种语言的文本的语料库)中的句子需要彼此匹配。

要扩展编辑距离算法以产生对齐，我们可以首先将对齐可视化通过编辑距离矩阵的路径。图 2.19 用加粗的单元格显示了该路径。每个加粗的单元格表示两个字符串中一对字母的对齐方式。如果两个加粗的单元格出现在同一行中，则从源到目标的过程中会有插入；同一列中两个加粗的单元格表示删除。

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: 计算从 intention 到 execution 之间的最小编辑距离

图注：用图 2.17 的算法，计算 Levenshtein 距离，插入或删除的成本为 1，替换的成本为 2。

图 2.19 还显示了如何计算此对齐路径的直觉。计算分两步进行。第一步，我们扩大最小编辑距离算法，以在每个单元格中存储反向指针。单元格的反向指针指向当前单元格的前一个(或多个)单元格。我们已经在图 2.19 中展示了这些反向指针的原理图。有些单元格有多个向后指针，因为最小扩展可能来自以前的多个单元格。第二步，我们执行一个回溯(back trace)。在回溯中，我们从最后一个单元格(最后一行和最后一列)开始，沿着指针返回动态规划矩阵。在最终单元格和初始单元格之间的每个完整路径都是最小距离对齐。练习 2.7 要求您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。

	#	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: 用反向指针计算对齐路径的原理图

图注：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三个相邻的单元格中的哪一个。在表满了之后，我们通过回溯来计算对齐(最小编辑路径)，从右下角的 8 开始，沿着箭头往回走。粗体单元格序列表示两个字符串之间的一种可能的最小成本对齐方式。根据 Gusfield(1997)之后的图表设计。

当我们用简单的 Levenshtein 距离来处理我们的例子时，图 2.17 中的算法允许操作上的任意权重。例如，为了纠正拼写错误，替换更可能发生在键盘上相邻的字母之间。维特比(Viterbi)算法是最小编辑距离的概率扩展。维特比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 8 章对此进行更多的讨论。

2.6. 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**分词和规范化**、**句子分割**和**词干提取**。我们还介绍了用于比较字符串的重要的**最小编辑距离**算法。以下是我们关于这些想法的要点总结:

- **正则表达式**语言是模式匹配的强大工具。
- 正则表达式的基本操作包括**连接符号**、**析取符号**(`[]`, `|`, 和 `.`), **计数器**(`*`, `+`, 和 `{n,m}`), **定位符**(`^`, `$`)和**优先级操作符**(`(,)`)。
- **单词符记化和规范化**通常是由级联的简单正则表达式替换或有限自动机。
- **Porter(波特)算法**是一个简单而有效的方法, 用来做词干提取、剥离词缀。它的精度不高, 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字符串所需的最少操作数。最小编辑距离可以通过**动态规划**来计算, 这也会导致两个字符串对齐。

2.7. 文献和历史说明

Kleene(1951, 1956)基于麦卡洛克-皮茨神经元, 首先定义了正则表达式和有限自动机。Ken Thompson是最早将正则表达式编译器构建为文本搜索编辑器的人之一(Thompson, 1968)。他的编辑器 `ed` 包含了一个命令 `g/正则表达式/p`, 即**全局正则表达式打印(Global Regular Expression Print, GREP)**, 它后来成为 `Unix grep` 实用程序。

文本规范化算法从一开始就被应用字段。最早被广泛使用的词干分析器之一是 Lovins(1968)。Packard(1973)也很早就将词干提取应用于数字人文学科, 他建立了一个古希腊语词缀剥离形态学解析器。目前, 可以使用多种用于符记化和规范化的代码, 例如 `Stanford` 符记生成器(<http://nlp.stanford.edu/software/tokenizer.shtml>)或 `Twitter` 专用的符记生成器(O'Connor 等人 2010), 或用于表达情感(<http://sentiment.christopherpotts.net/tokenizing.html>)。有关文本预处理的概述, 请参见 Palmer(2012)。NLTK 是必不可少的工具, 可提供有用的 `Python` 库(<http://www.nltk.org>)和教科书说明(Bird 等人 2009), 其中包括文本规范化和语料库接口在内的许多算法。

有关赫尔丹(Herdan)定律和希普斯(Heaps)定律的更多信息, 请参见 Herdan(1960, p.28), Heaps(1978), Egghe (2007) 和 Baayen(2001); Yasseri 等人(2012)讨论了与其他语言复杂性度量之间的关系。有关编辑距离的更多信息, 请参见优秀的 Gusfield(1997)。我们测量从“intention”到“execution”的编辑距离的示例改编自 Kruskal(1983)。有各种公开可用的软件包可以计算编辑距离, 包括 `Unix diff` 和 `NIST sclite` 程序(NIST, 2005)。

Bellman(1984)在他的自传中解释了他最初是如何提出动态规划这个术语的:

“...20 世纪 50 年代对数学研究来说不是好年头。国防部长...对研究这个词有一种病态的恐惧和憎恨...因此, 我决定用“规划”这个词。我想让大家明白这是动态的, 是多阶段的.....我想, 让我们.....拿一个意义绝对精确的词来说, 那就是动态.....用“动态”这个词来贬义是不可能的。试着想出一些可能带有贬义的组合。这是不可能的。因此, 我认为动态规划是个好名字。这是连国会议员都不会反对的事情。”

2.8. 练习

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 immediately 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 11. You might want to choose a different domain than a Rogerian psychologist, 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 distance 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.

