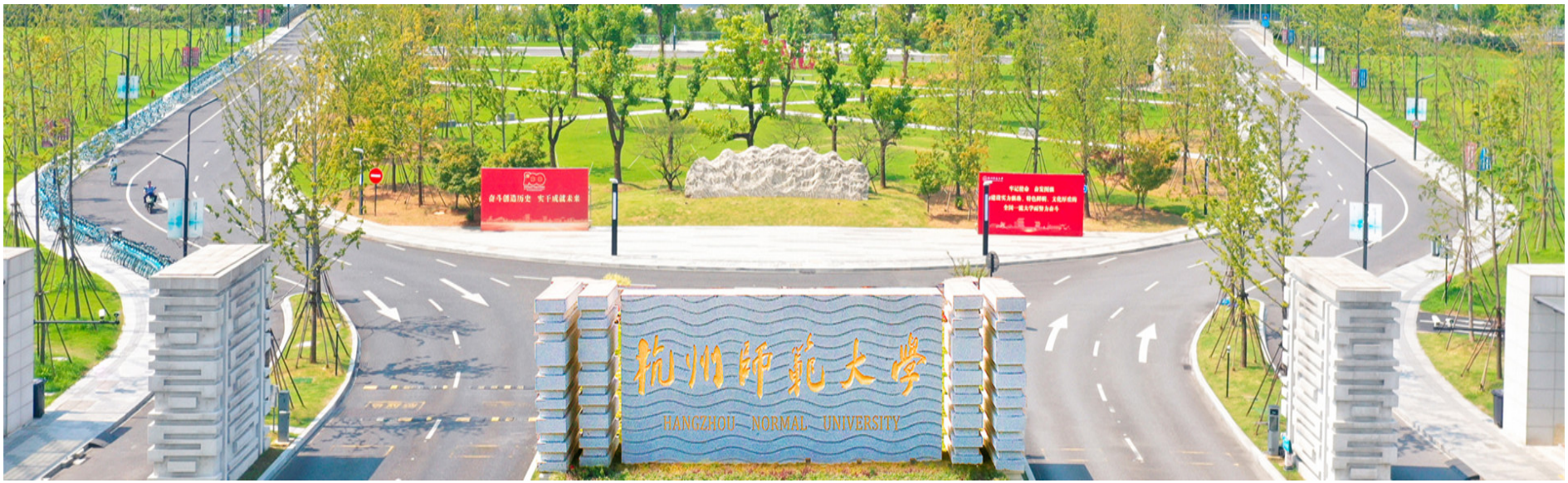
第二讲 - 文本处理基础：正则表达式与文本标准化

张建章

阿里巴巴商学院
杭州师范大学

2025-02-01



目录

正则表达式的定义与作用

正则表达式（Regular Expression，简称 regex）是一种用于匹配和操作字符串的语言。在文本处理中，正则表达式广泛应用于模式匹配，可以帮助我们查找、替换、拆分字符串。其核心任务是定义一组规则来描述我们需要的文本模式，通过这些模式，我们能够有效地在文本中进行搜索、替换、提取等操作。

正则表达式基本操作

正则表达式通过一系列的**字符**和**符号**组成，执行不同的匹配操作。其最基础的操作为字面字符匹配。

字面字符匹配：将字符按照顺序组合 (Concatenation) 来进行匹配。可以简单地使用一个正则表达式来查找一段固定的字符序列。

例如，查找文本中是否有“市场营销”这一具体词组：

```
import re

text = " 我们公司最近进行了市场营销活动，市场营销效果非常好。"
# 定义正则表达式
pattern = r" 市场营销"
# 使用 re.search 来查找匹配
result = re.search(pattern, text)
if result:
  print(f" 匹配成功: {result}")
else:
  print(" 没有匹配到")
```

例如，查找文本中是否有“销售”和“广告”这两个字。

```
text = " 我们的销售和广告团队合作密切，取得了显著成绩。"

# 定义正则表达式
pattern = r" 销售 | 广告"

# 使用 re.findall 来查找所有匹配
matches = re.findall(pattern, text)

print(f" 匹配到的词语: {matches}")
```

正则表达式 r" 销售 | 广告" 使用 | 表示“或”操作，匹配文本中的“销售”或“广告”字样。`re.findall()` 会返回所有匹配的结果。

练习：自学`re.match`、`re.search`、`re.findall`的区别；

字符集与字符范围

正则表达式提供了字符集和字符范围的功能，使得用户能够灵活地匹配多种字符，字符集和字符范围允许匹配一组字符中的任意一个字符：

- **字符集**：使用方括号 [] 定义字符集，匹配集合中的任意一个字符。例如，/[wW]/ 会匹配 w 或 W；
- **字符范围**：使用连字符 - 定义字符的范围。例如，/[a-z]/ 匹配任意一个小写字母，/[0-9]/ 匹配任意一个数字；
- **否定字符集**：使用 ^ 作为字符集的第一个字符，表示匹配集合中不包含的字符。例如，/[^a]/ 匹配任何非 a 字符。

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 1: 常用字符集别名

例如，查找文本中所有以“广”或“销”开头的词语。

```
text = " 我们正在进行广泛的市场推广和销售活动。"

# 定义正则表达式
pattern = r"[广销]\w+?"
# 使用 re.findall 来查找所有匹配
matches = re.findall(pattern, text)
print(f" 匹配到的词语: {matches}")
# 匹配到的词语: ['广泛', '广和', '销售']
```

① ? 表示非贪婪模式，匹配满足条件的最短字符串；② Python 3 的 re 模块默认启用了 Unicode 支持，\w 被定义为匹配 Unicode “单词字符”，包括中文字符，不包括标点符号；

注意：基本汉字字符集的正则表达式为 `r'[\u4e00-\u9fff]'`。

练习：从一段中英文混合的文本中提取出所有的汉字内容。

量词

量词指定了一个字符或子表达式出现的次数。常见的量词有 *、+、? 和 {n,m}：

- *：匹配前一个字符零次或多次（例如，/a*/ 会匹配零个或多个 a）；
- +：匹配前一个字符一次或多次（例如，/a+/ 会匹配一个或多个 a）；
- ?：匹配前一个字符零次或一次（例如，/colou?r/ 会匹配 color 或 colour）；
- {n}：匹配前一个字符恰好出现 n 次（例如，/[0-9]{3}/ 匹配恰好三个数字）。

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: 常用量词

例如，查找文本中所有以“广”或“销”开头的词语。

```
text = " 这个产品的市场推广的效果非常好，值得关注的点很多。"

# 定义正则表达式
pattern = r"\w{,2} 的\w{,2}"

# 使用 re.findall 来查找所有匹配
matches = re.findall(pattern, text)

print(f" 匹配到的词语: {matches}")
# ['产品的市场', '推广的效果', '关注的点很']
```

正则表达式 r"\w,2 的\w,2" 查找文本中所有符合“最多两个字符 + 的 + 最多两个字符”模式的子字符串。

特殊字符

正则表达式中有一系列特殊字符，这些字符具有特定的功能：

- .（点号）：匹配任意单个字符（除了换行符）；
- ^（脱字符）：匹配字符串的开始位置；
- $（美元符号）：匹配字符串的结束位置；
- \b：匹配一个单词边界（即空格、标点等地方）；
- \B：匹配非单词边界。

```python
import re
text = " 这家公司最近扩展了其业务，成效显著。"
pattern = r" 这. 公司"
result = re.search(pattern, text)
print(result.group())  # 输出: " 这家公司"
```

```python
pattern = r"^ 这家公司"
result = re.match(pattern, text)
print(result.group())  # 输出: " 这家公司"
```

```python
pattern = r" 显著。$"
result = re.search(pattern, text)
print(result.group())  # 输出: " 显著。"
```

\b 匹配一个单词边界，即匹配单词和非单词字符之间的位置（如空格、标点符号等）。

```
text = " 这家公司最近扩展了其业务，成效显著。"
pattern = r" 业务\b"
result = re.findall(pattern, text)
print(result)  # 输出: ['业务']
```

\B 匹配非单词边界，即匹配在单词与单词之间的位置。

```
pattern = r"\B 成效"
result = re.findall(pattern, text)
print(result)  # 输出: ['公司']
```

分组与捕获组

正则表达式可以使用圆括号 () 来将表达式进行分组。分组不仅可以改变运算符的优先级，还可以为分组的内容创建捕获组，以便后续引用。

- 分组：通过圆括号创建分组。例如，/(abc)/ 匹配字符串 abc；
- 捕获组：使用圆括号对部分正则表达式进行分组，并将匹配到的内容保存为一个组。例如，/(abc)(def)/ 会匹配 abcdef，并将 abc 和 def 分别保存到两个捕获组中。

```
import re
text = "公司A的季度财报显示利润增长30%，表现出色。
↪  公司B的销售额增长了15%，也表现良好。"
pattern = r"公司([A-Z])的([\u4e00-\u9fff]+)"  #
↪  匹配公司名和其业绩描述
result = re.findall(pattern, text)
print(result)
# 输出: [('A', '季度财报显示利润增长'), ('B', '销售额增长了')]
```

应用场景与实现

正则表达式在各种文本处理任务中都有广泛应用，常见场景包括：

- 文本搜索：查找文件或字符串中是否含有特定模式，或者提取匹配的部分。
- 替换操作：使用正则表达式进行文本替换，例如将文件中的某些特定格式的日期替换为统一格式。
- 数据验证：用于验证文本是否符合特定格式，例如电子邮件、电话号码等。

正则表达式在不同的编程语言和工具中有不同的实现方式。在 Unix 系统中，grep 命令就是基于正则表达式进行文本搜索的工具。在 Python 中，re 模块提供了正则表达式的实现，允许用户通过编程实现匹配、搜索和替换等操作。

练习：利用正则表达式来验证参考文献条目是否符合 GB/T 7714 标准中的中文期刊文献格式：

作者 1, 作者 2. 文章标题 [J]. 期刊名称, 年份, 卷号 (期号): 起止页码.

目录

语料库的特点

语料库（corpus）是进行自然语言处理（NLP）的基础，它指的是一组有代表性的、经过精心收集和标注的文本或语音数据集。在 NLP 研究中，语料库不仅是数据分析的基础，更能反映出语言使用的特定背景、语境和社会特征。因此，语料库的构建与选择对 NLP 任务的有效性至关重要。

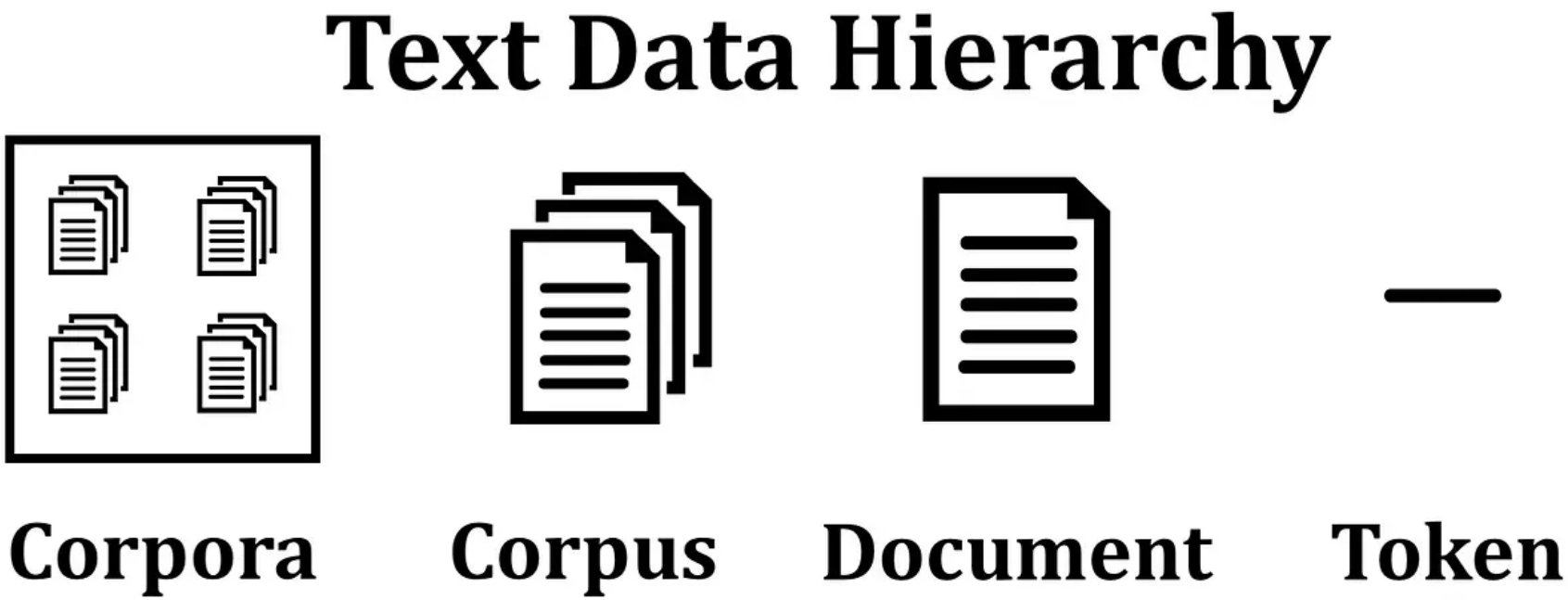


图 3: 语料库的层次结构

- **语言多样性**：世界上共有约 7097 种语言，而 NLP 算法大多是在英语语料库上开发和测试的。这种情况可能导致算法在处理其他语言时遇到困难，理想的 NLP 算法应能够处理多种语言。因此，在构建语料库时，研究者需要考虑语言的多样性，确保语料库能够代表不同语言的特点，并通过跨语言的研究拓展算法的适用范围。

- **语料库中的体裁差异**：不同体裁的文本在语言使用上有很大不同。例如，新闻报道、小说、学术论文、社交媒体和法律文件等文本形式都有其特定的表达方式和语言特点。因此，在进行文本分析时，研究者必须选择适合的语料库，并考虑体裁差异的影响。

- **语料库中的社会人口特征**：语料库的语言使用还受到说话者或写作者的社会人口特征的影响。这些特征包括年龄、性别、种族、社会阶层等，它们都会在一定程度上影响语言的表达方式。理解这些社会人口特征对语言的影响，有助于提高语料库分析的准确性和应用的适应性。

- **语料库的历史维度**：语言是动态变化的，随着时间的推移，语言的结构和词汇会发生变化。特别是在处理历史文本或长时间跨度的数据时，理解语言随时间变化的特点非常重要。

- **语料库的应用与数据声明**：为了确保语料库的质量和可信度，语料库创建者应提供详细的数据说明，明确语料库的收集动机、语料来源、收集过程、数据预处理、注释过程以及数据分发方式等信息。

目录

文本挖掘最新进展

分词（Tokenization）是文本处理中的基础任务之一，指的是将连续的文本划分为一个个有意义的单位（即 token），这些单位通常为单词、子词或字符。文本分词的两种主要方法：自顶向下分词和子词分词。

自顶向下分词依赖规则或预定义的词典，适合处理结构清晰的文本；

子词分词将词语进一步拆解为更小的单元（子词、字符或字母）来处理文本，能够有效处理未登录词和复杂语言结构。子词分词方法，如 BPE 和单语语言模型，能够为现代 NLP 模型提供更加灵活且强大的文本处理能力，尤其是在处理大规模语料和多语言任务时具有显著优势。

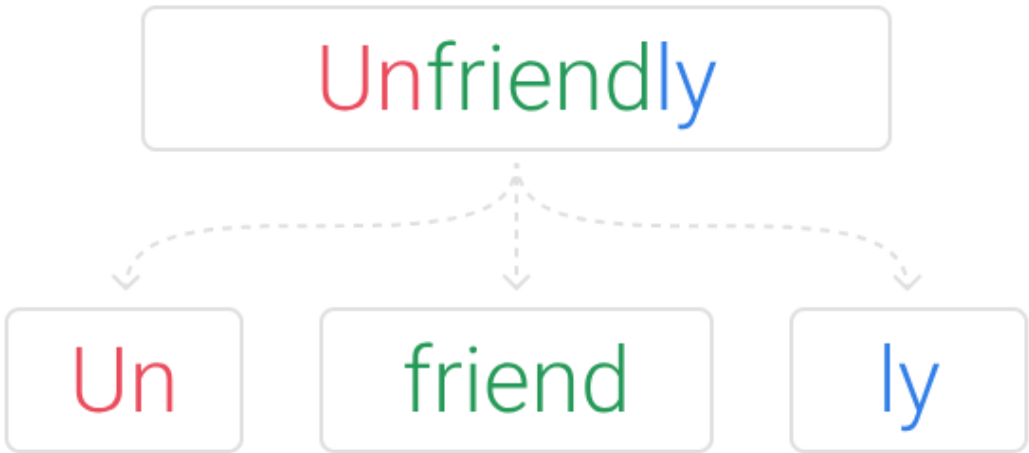


图 4: 子词 (subword) 分词

Byte-Pair Encoding (BPE) 方法

BPE 是一种基于统计的自底向上的分词方法，广泛应用于现代大规模语言模型的训练中。BPE 的基本思想是通过合并文本中最常见的字符对，逐步构建出新的子词单元。具体过程如下：

- 首先，BPE 从单字符开始，将所有文本中的字符视为独立的 token (token learner)；
- 然后，BPE 统计文本中最常见的字符对，并将它们合并成新的 token，直到合并达到预定次数为止 (token learner)；
- 合并后,BPE 使用这些新的子词单元进行文本的分词 (token segmenter)。

分词时，先把文本切分为单子符，然后按照学习到的合并顺序 (即，token 被学习到的顺序) 贪婪地执行合并操作。

BPE 的优势在于能够处理未登录词（OOV，Out-of-Vocabulary），即在训练数据中没有出现过的词。通过将未登录词拆分为更小的已知子词单元，BPE 能够有效地处理这些词汇，并且减少词汇表的大小。

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

$V \leftarrow$ all unique characters in C # initial set of tokens is characters
for $i = 1$ **to** k **do** # merge tokens k times
  $t_L, t_R \leftarrow$ Most frequent pair of adjacent tokens in C
  $t_{NEW} \leftarrow t_L + t_R$ # make new token by concatenating
  $V \leftarrow V + t_{NEW}$ # 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).

图 5: BPE 子词分词算法 (token learner part)

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_

图 6: BPE 子词分词算法运行示例 (token learner part)

词与子词 (Word V.S. Subword)

在 NLP 任务中，词汇单元（token）不仅仅指完整的单词，也可能是一个词的部分（即子词）。对于大多数任务，单词 (word) 作为 token 通常足够，但对于一些复杂任务，如处理未登录词、词形变化丰富的语言（如阿拉伯语、德语等）时，子词 (subword) 分词显得尤为重要。子词分词通过拆解词汇中的语素（如前缀、词根、后缀）来实现对词汇的更细致分析，从而提高 NLP 模型的鲁棒性。

在实际应用中，许多现代的 NLP 模型，如 BERT、GPT 等，使用基于 BPE 或其他子词分词方法的 tokenization。这些模型通过将词分解为更小的单位，使得在处理长词、稀有词或复杂语言时能够更高效且精确。

分词在不同语言中存在差异。例如，中文、日语和泰语等语言并不使用空格分隔单词，因此需要特殊的分词算法来识别词语边界。对于这些语言，通常需要使用基于字符或子词的 tokenization 技术，以确保语言模型能够处理这些没有明显分隔符的语言。

目录

词汇预处理：词汇标准化、词形还原和词干提取是文本预处理中的关键步骤，它们通过不同的方式简化和统一文本中的单词形式。词汇标准化通常涉及大小写折叠和同义词归一化，词形还原通过分析单词的词缀和词根来还原单词的基本形式，而词干提取则是通过简单的规则去掉词缀来提取词干。

- 大小写折叠 (统一)："Wood" 和"wood" 被视为相同的单词；
- 同义词归一化：将“USA”和“US”视为同义词；
- 口语化缩写的标准化：将“uh-huh”和“uhhuh”视为同一表达。

句子预处理：句子分割是自然语言处理中不可忽视的一步，其准确性直接影响到后续文本处理任务的效果。由于标点符号具有多重含义，句子分割不仅依赖标点符号，还需要结合上下文、词性标注以及其他语言特征来判断句子的边界。随着机器学习方法的引入，句子分割的精度有了显著提升，但仍需要根据特定任务和语言特点进行调整和优化。

目录

最小编辑距离是用于衡量两个字符串之间相似度的一个度量标准，它表示通过插入、删除或替换操作，将一个字符串转换为另一个字符串所需的最小操作次数。这个概念对于拼写校正、语音识别、文本相似性度量等多个 NLP 任务都有重要应用。

- **语音识别**：在语音识别中，最小编辑距离用于比较识别出的单词与真实单词之间的相似度，帮助修正识别错误；
- **文本相似性度量**：最小编辑距离可以用于文本之间的相似性评估，特别是在没有预先定义词汇表的情况下，例如在 DNA 序列比对中，或在比较未登录词时；
- **核心指代消解**：在文本中确定不同字符串是否指代同一对象时，最小编辑距离提供了一种衡量两个字符串相似度的方法。。

例如，在拼写校正中，假设用户输入了“coffie”，而系统需要找到一个与之最相似的正确单词（比如“coffee”）。最小编辑距离提供了一种方法来评估哪些单词更接近用户输入，操作次数越少的单词越可能是用户的真实意图。

最小编辑距离的计算

最小编辑距离计算的基本操作包括：

- 插入（Insertion）：在一个字符串中插入一个字符；
- 删除（Deletion）：删除一个字符；
- 替换（Substitution）：将一个字符替换为另一个字符。

每个操作都可能具有不同的代价，通常情况下，插入、删除和替换的代价都是 1 (Levenshtein 距离)。最小编辑距离即为将一个字符串通过这些操作转换为另一个字符串所需的最小代价。

Levenshtein 距离的一个变体是只允许插入和删除操作，替换操作等价于一次删除加一次插入，因此替换操作的代价为 2。

动态规划求解最小编辑距离 I

1. 定义状态与初始化

定义一个二维数组 $D[i][j]$，表示将字符串 X 的前 i 个字符转换为 Y 的前 j 个字符所需要的最小编辑距离。i 表示 X 中的字符索引，j 表示 Y 中的字符索引。初始化时：

- $D[0][j]$：表示将空字符串 X 转换为 Y 的前 j 个字符，显然，所需操作数是 j 次插入操作（插入 $Y[j]$ 中的每一个字符）。

- $D[i][0]$：表示将 X 的前 i 个字符转换为空字符串，显然，所需操作数是 i 次删除操作（删除 $X[i]$ 中的每一个字符）。

2. 递推公式

根据递推公式计算 $D[i][j]$ 的值。递推公式如下：

动态规划求解最小编辑距离 II

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

- $D[i-1,j]+1$ 表示将 X 的第 i 个字符删除；

- $D[i,j-1]+1$ 表示在 X 的末尾插入 Y 的第 j 个字符；

- $D[i-1,j-1] + \text{cost}(X[i], Y[j])$ 表示如果 X 的第 i 个字符与 Y 的第 j 个字符不同，则需要替换。

3. 逐步计算

根据递推公式逐个计算 $D[i][j]$ 的值，填充二维数组。

以 Levenshtein 距离变体计算为例，以上述算法过程如下；

以 Levenshtein 距离变体计算为例，以上述算法过程如下；

```
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., $\forall x, \text{ins-cost}(x) = 1$) or can be specific to the letter (to model the fact that some letters are more likely to be inserted than others). We assume that there is no cost for substituting a letter for itself (i.e., $\text{sub-cost}(x,x) = 0$).

计算单词 intention 和 execution 之间的 Levenshtein 距离 (变体)：

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 deletions, 2 for substitutions.

使用 Levenshtein 距离 (变体)，找出单词 intention 和 execution 之间的最小编辑路径 (有多条)，黑色格子序列为其中一条路径；

	#	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, again using Levenshtein distance with cost of 1 for insertions or deletions, 2 for substitutions. Diagram design after Gusfield (1997).

目录

1. 应用 Python 正则表达式判断参考文献条目是否符合 GB/T 7714 标准中的英文期刊/英文会议文献格式。

> Author1, Author2. Title of the article[J]. Journal Name, Year, Volume(Issue): Pages.
> Author1, Author2. Title of the paper[C]. In: Proceedings of the Conference Name, Year, pages.

2. 编写 Python 代码实现利用动态规划计算 Levenshtein 距离。

THE END