Lecture 5 超基础复习笔记 – 正则表达式进阶 & 文本预处理

零、先理解问题

Lecture 5分为两部分

Part A：正则表达式进阶 – 处理HTML标签 – 提取URL链接 – 使用反向引用（backreferences）

Part B：文本预处理 – 分词（Tokenization） – 小写转换（Lowercasing） – Twitter特殊元素处理（@mentions, #hashtags, emojis） – 类型–标记比率（Type–Token Ratio, TTR）

Part A：正则表达式进阶（HTML解析）

一、问题背景

任务：从HTML文件中提取信息

```
<html>
<head>
<title>COM6115: Text Processing</title>
</head>
<body>
<p>This is a <b>bold</b> word.</p>
<a href="http://example.com">Link</a>
</body>
</html>
```

需要提取： 1. HTML标签（如 `<p>`，`<b>`，`</p>`） 2. 区分开标签和闭标签 3. 标签参数（如 `href="..."`） 4. URL链接 5. 配对的标签之间的文本

二、核心概念

1. 贪婪匹配 vs 非贪婪匹配

贪婪匹配（Greedy）：尽可能多地匹配

```
text = "<p><b>text</b></p>"

# 贪婪匹配
pattern = '<.*>'
# 匹配结果：<p><b>text</b></p>  ← 整个字符串！
```

为什么？

```
'.*' 表示：匹配任意字符，任意多次
从第一个 '<' 开始，一直到最后一个 '>'
```

问题：我们想要分别匹配每个标签，而不是一次全匹配！

非贪婪匹配（Non-greedy）：尽可能少地匹配

```
text = "<p><b>text</b></p>"

# 非贪婪匹配
pattern = '<.*?>'
# 匹配结果：<p>, <b>, </b>, </p>  ← 4个单独的标签！
```

`?` 的作用：

```
'.*?' 表示：匹配任意字符，但尽可能少
遇到第一个 '>' 就停止
```

2. 字符类取反 `[^...]`

含义：匹配不在括号内的任意字符

```
# 例子1：匹配非数字
pattern = '[^0-9]'
text = "a1b2c3"
# 匹配：a, b, c（不匹配数字）

# 例子2：匹配HTML标签（不包含>）
pattern = '<[^>]+>'
text = "<p><b>text</b></p>"
# 匹配：<p>, <b>, </b>, </p>
```

为什么有效?

```
'<[^>]+>' 的意思：
< ← 以 < 开始
[^>]+ ← 匹配1个或多个"不是>"的字符
> ← 以 > 结束

这样就不会跨越多个标签！
```

---

3. 反向引用（Backreferences）

问题：如何匹配配对的标签?

```
<b>bold text</b>  ← 正确配对
<b>bold text</i>  ← 错误配对！
```

反向引用语法： `\1`，`\2`，`\3`...

```
# 匹配配对的开闭标签
pattern = r'<(\w+)>(.*?)</\1>'
```

详细解释：

```
r'<(\w+)>(.*?)</\1>'
   ↑      ↑     ↑
   |      |     └─ 反向引用第1组（必须匹配相同的标签名）
   |      └─ 第2组：标签之间的文本（非贪婪）
   └─ 第1组：标签名（用括号捕获）

例子：
<b>text</b>
第1组匹配：b
第2组匹配：text
\1 要求：也必须是 b（所以 </b> 才匹配）
```

执行示例：

```
import re

text = "<b>bold</b> and <i>italic</i>"
pattern = r'<(\w+)>(.*?)</\1>'

matches = re.finditer(pattern, text)
for m in matches:
    print(f"标签：{m.group(1)}，内容：{m.group(2)}")

# 输出：
# 标签：b，内容：bold
# 标签：i，内容：italic
```

为什么要用 `r'...'`（raw string）？

```
# 错误：不用 r
pattern = '<(\w+)>(.*?)</\1>'
# Python会把 \1 理解为转义字符（报错！）

# 正确：用 r
pattern = r'<(\w+)>(.*?)</\1>'
# Python保留 \1 作为正则表达式的反向引用
```

三、解决方案详解

任务1：匹配HTML标签

方案1：非贪婪匹配

```
tag = re.compile('</?(.*?)>')
```

解释：

```
</?(.*?)>
↑  ↑   ↑
|  |   └─ 右尖括号
|  └─ 第1组：标签名（非贪婪）
└─ 可选的斜杠（闭标签用）

例子：
<p>     → 匹配，group(1)='p'
</p>    → 匹配，group(1)='p'
<title> → 匹配，group(1)='title'
```

问题：这个方案会把参数也包括进去！

```
<a href="..."> → group(1) = 'a href="..."'  ← 不好！
```

方案2：字符类取反

```
tag = re.compile('</?([^>]+)>')
```

解释：

```
</?([^>]+)>
   ↑    ↑
   |    └─ 1个或多个"非>"字符
   └─ 第1组
```

```
例子：
<p>            → group(1) = 'p'
<a href="..."> → group(1) = 'a href="..."'
```

还是有问题：参数还在里面！

方案3：分离标签名和参数（最佳）

```
tag = re.compile(r'</?(\w+\b)([^>]+)?>')
```

详细解释：

```
r'</?(\w+\b)([^>]+)?>'
    ↑    ↑  ↑     ↑
    |    |  |     └─ 第2组：参数部分（可选）
    |    |  └─ 1个或多个"非>"字符
    |    └─ 单词边界（确保标签名完整）
    └─ 第1组：标签名

\w+ ← 1个或多个单词字符（标签名）
\b  ← 单词边界（Word Boundary）
```

`\b` 的作用（重要！）：

```
\b = 单词边界（Word Boundary）

位置标记，出现在：
- 单词字符(\w)和非单词字符之间
- 字符串开头/结尾

例子：
"<table border=1>"
  ↑    ↑
  table 后面是空格
  空格不是\w，所以这里是\b

\w+\b 确保只匹配 "table"，不包括后面的空格和参数
```

执行示例：

```
tag = re.compile(r'</?(\w+\b)([^>]+)?>')

html = '<table border=1 cellspacing=0>'
m = tag.search(html)

print(m.group(1))  # 'table'  ← 标签名
print(m.group(2))  # ' border=1 cellspacing=0'  ← 参数
```

任务2：区分开标签和闭标签

开标签：

```
openTag = re.compile(r'<(\w+\b)([^>]+)?>')
```

闭标签：

```
closeTag = re.compile(r'</(\w+\b)\s*>')
```

区别：

```
开标签：<(\w+\b)([^>]+)?>
        ↑
        没有斜杠要求，可选的参数

闭标签：</(\w+\b)\s*>
         ↑       ↑
         必须有斜杠   可选的空格（有些HTML会写成 </p  >）
```

执行示例：

```
openTag = re.compile(r'<(\w+\b)([^>]+)?>')
closeTag = re.compile(r'</(\w+\b)\s*>')

html = '<p>This is <b>bold</b> text.</p>'
```

```
# 查找开标签
for m in openTag.finditer(html):
    print('OPENTAG:', m.group(1))
    if m.group(2):  # 如果有参数
        print('  PARAMS:', m.group(2))

# 查找闭标签
for m in closeTag.finditer(html):
    print('CLOSETAG:', m.group(1))

# 输出:
# OPENTAG: p
# OPENTAG: b
# CLOSETAG: b
# CLOSETAG: p
```

任务3：提取标签参数

参数格式：

```
<table border=1 cellspacing=0 cellpadding=8>
       ↑        ↑            ↑
       参数1    参数2        参数3
```

解决方案：

```
openTag = re.compile(r'<(\w+\b)([^>]+)?>')

html = '<table border=1 cellspacing=0 cellpadding=8>'
m = openTag.search(html)

if m.group(2):  # 如果有参数
    params = m.group(2).split()  # 按空格分割
    for param in params:
        print('PARAM:', param)

# 输出:
# PARAM: border=1
```

```
# PARAM: cellspacing=0
# PARAM: cellpadding=8
```

为什么用 `.split()`？

```
# m.group(2) = ' border=1 cellspacing=0 cellpadding=8'
#               ↑ 开头有空格

params = m.group(2).split()
# .split() 默认按空格分割，并自动去除首尾空格
# 结果: ['border=1', 'cellspacing=0', 'cellpadding=8']
```

任务4：匹配配对的开闭标签

使用反向引用：

```
openCloseTagPair = re.compile(r'<(\w+\b)([^>]+)?>(.*?)</\1\s*>')
```

详细解释：

```
r'<(\w+\b)([^>]+)?>(.*?)</\1\s*>'
   ↑      ↑         ↑     ↑
   |      |         |     └─ 反向引用第1组（闭标签必须同名）
   |      |         └─ 第3组：标签之间的内容（非贪婪）
   |      └─ 第2组：开标签的参数（可选）
   └─ 第1组：开标签名

例子：
<b>bold text</b>
第1组：b
第2组：None（没有参数）
第3组：bold text
\1：必须是 b
```

执行示例：

```
pattern = r'<(\w+\b)([^>]+)?>(.*?)</\1\s*>'
html = '<p>This is <b>bold</b> and <i>italic</i> text.</p>'
```

```
for m in re.finditer(pattern, html):
    print(f"PAIR [{m.group(1)}]: \"{m.group(3)}\"")

# 输出:
# PAIR [b]: "bold"
# PAIR [i]: "italic"
# PAIR [p]: "This is <b>bold</b> and <i>italic</i> text."
```

为什么要非贪婪 `.*?`？

```
html = '<b>first</b> text <b>second</b>'

# 贪婪匹配 .*
pattern = r'<(\w+)>(.*)</\1>'
# 匹配: <b>first</b> text <b>second</b>  ← 错误！跨越了两个标签

# 非贪婪匹配 .*?
pattern = r'<(\w+)>(.*?)</\1>'
# 匹配: <b>first</b> 和 <b>second</b>  ← 正确!
```

任务5：提取URL

HTML中的链接格式：

```
<a href="http://example.com">链接文本</a>
<a href=http://example.com>链接文本</a>  ← 也可以不用引号
```

方案1：带引号的URL

```
url = re.compile('href=("[^">]+")', re.I)
```

解释：

```
href=("[^">]+")
      ↑    ↑
      |    └ 右引号
      └ 左引号 + 1个或多个"非引号且非>"的字符
```

```
[^">]+ ← 既不是引号"，也不是尖括号>

re.I ← 忽略大小写（HREF, Href, href都能匹配）
```

方案2：不带引号的URL

```
url = re.compile('href=([^">\s]+)', re.I)
```

解释：

```
href=([^">\s]+)
      ↑
      1个或多个"非引号、非>、非空格"的字符

[^">\s]+ ← 不是引号，不是>，不是空格
```

方案3：两种情况都支持（最佳）

```
url = re.compile(r'href=("[^">]+"|[^">\s]+)', re.I)
```

详细解释：

```
r'href=("[^">]+"|[^">\s]+)'
        ↑         ↑
        |         └─ 或：无引号的URL
        └─ 带引号的URL

竖线 | 表示"或"
左边：带引号的格式  "[^">]+"
右边：无引号的格式  [^">\s]+
```

执行示例：

```
url = re.compile(r'href=("[^">]+"|[^">\s]+)', re.I)

html1 = '<a href="http://example.com">Link</a>'
html2 = '<a HREF=http://example.com>Link</a>'

print(url.search(html1).group(1))  # "http://example.com"
print(url.search(html2).group(1))  # http://example.com
```

任务6：删除HTML标签（HTML Stripping）

目标：把HTML标签全部删除，只保留文本

```
输入：<p>This is <b>bold</b> text.</p>
输出：This is bold text.
```

使用 `.sub()` 方法：

```
tag = re.compile(r'</?(\w+\b)([^>]+)?>')

html = '<p>This is <b>bold</b> text.</p>'
stripped = tag.sub('', html)
print(stripped)  # 'This is bold text.'
```

`.sub()` 详解：

```
pattern.sub(replacement, string)
            ↑            ↑
            |            └─ 要处理的字符串
            └─ 替换成什么（空字符串=删除）

tag.sub('', html)
        ↑
        空字符串 ← 把所有匹配的标签替换成空
```

执行过程：

```
html = '<p>This is <b>bold</b> text.</p>'

# 找到的标签:
# 1. <p>
# 2. <b>
# 3. </b>
# 4. </p>

# 每个都替换成 ''

# 结果: 'This is bold text.'
```

四、完整代码示例

```
import re

# 定义所有正则表达式
tag = re.compile(r'</?(\w+\b)([^>]+)?>')
openTag = re.compile(r'<(\w+\b)([^>]+)?>')
closeTag = re.compile(r'</(\w+\b)\s*>')
openCloseTagPair = re.compile(r'<(\w+\b)([^>]+)?>(.*?)</\1\s*>')
url = re.compile(r'href=("[^">]+"|[^">\s]+)', re.I)

# 读取HTML文件
with open('RGX_DATA.html') as infs:
    linenum = 0
    for line in infs:
        linenum += 1
        if line.strip() == '':
            continue

        print('-' * 100, '[%d]' % linenum)
        print('TEXT:', line, end='')

        # 1. 查找所有标签
        for m in tag.finditer(line):
            print('** TAG:', m.group(1))

        # 2. 查找开标签（含参数）
        for m in openTag.finditer(line):
            print('** OPENTAG:', m.group(1))
```

```
            if m.group(2):
                for param in m.group(2).split():
                    print('    PARAM:', param)

        # 3. 查找闭标签
        for m in closeTag.finditer(line):
            print('** CLOSETAG:', m.group(1))

        # 4. 查找配对的标签
        for m in openCloseTagPair.finditer(line):
            print("** PAIR [%s]: \"%s\"" % (m.group(1), m.group(3)))

        # 5. 查找URL
        for m in url.finditer(line):
            print('** URL:', m.group(1))

        # 6. 删除标签
        stripped = tag.sub('', line)
        print('** STRIPPED:', stripped, end='')
```

Part B：文本预处理（Twitter数据）

一、文本预处理的必要性

问题：

```
"LOVe" 和 "love" 是同一个词吗?
"food" 和 "food." 是同一个词吗?
"I'm" 应该算1个词还是2个词?
```

解决方案：文本预处理

```
1. 分词（Tokenization）：把句子切分成单词
2. 小写转换（Lowercasing）：统一大小写
3. 标准化（Normalization）：统一格式
```

二、分词（Tokenization）

方法1：用正则表达式自己写

简单版本：

```
import re

def tokenise_regex(text):
    pattern = r"\w+(?:'\w+)?|[^\w\s]"
    return re.findall(pattern, text)
```

详细解释：

```
r"\w+(?:'\w+)?|[^\w\s]"
  ↑           ↑
  |           └─ 或：标点符号
  └─ 单词（可能包含撇号）

\w+(?:'\w+)? ← 匹配单词，如 "don't", "it's"
|            ← 或
[^\w\s]      ← 匹配标点符号（既不是单词字符也不是空格）
```

`(?:...)` 是什么？

```
(?:...) ← 非捕获组（Non-capturing group）

和 (...) 的区别：
(...)   ← 捕获组，会被 group(1) 提取
(?:...) ← 非捕获组，不会被提取，只用于分组

为什么用非捕获组？
因为我们只想匹配模式，不需要单独提取这部分
```

执行示例：

```
text = "He says I'm depressed most of the time."
tokens = tokenise_regex(text)
print(tokens)
```

```
# 输出:
# ['He', 'says', 'I', "'", 'm', 'depressed', 'most', 'of', 'the', 'tim
```

处理撇号:

```
pattern = r"\w+(?:'\w+)?|[^\w\s]"

# "I'm"
\w+      ← 匹配 'I'
(?:'\w+)?  ← 尝试匹配 "'m"
结果: 'I' 和 "'m" 分开

# "don't"
\w+      ← 匹配 'don'
(?:'\w+)?  ← 匹配 "'t"
结果: 'don' 和 "'t" 分开
```

问题: 撇号还是被分开了!

改进版本:

```
pattern = r"\w+(?:'\w+)?|[^\w\s]"

# 更好的版本:
pattern = r"\w+(?:'\w+)*|[^\w\s]"
#              ↑ 改成 *  (0次或多次)
```

但实际上原版就能工作:

```
text = "I'm"

# 匹配过程:
\w+       ← 匹配 'I'
(?:'\w+)? ← 尝试匹配 "'m"

但问题是: \w 也能匹配 m
所以 \w+ 会贪婪地匹配 'I', 'm' 都匹配了
(?:'\w+)? 就没机会匹配撇号了
```

```
正确的pattern应该更精确：
r"[a-zA-Z]+(?:'[a-zA-Z]+)?|[^\w\s]"
```

方法2：用NLTK的正则表达式分词器

NLTK提供的复杂pattern：

```
import nltk

pattern = r'''(?x)          # 允许verbose模式（可以写注释）
        (?:[A-Z]\.)+        # 缩写词，如 U.S.A.
      | \w+(?:-\w+)*        # 带连字符的词，如 San Francisco-based
      | \$?\d+(?:\.\d+)?%?  # 货币和百分比，如 $12.40, 82%
      | \.\.\.              # 省略号
      | [][.,;"'?():_`-]    # 标点符号
    '''

tokens = nltk.regexp_tokenize(text, pattern)
```

详细解释：

`(?x)` 模式：

```
(?x) ← verbose模式，允许：
  - 正则表达式中的空格被忽略
  - 可以写注释（# ...）
  - 让复杂的pattern更易读
```

各部分解释：

```
(?:[A-Z]\.)+  ← 缩写词
# U.S.A. → 匹配
# [A-Z]  ← 大写字母
# \.     ← 点号
# +      ← 1次或多次

\w+(?:-\w+)*  ← 带连字符的词
# San Francisco-based → 匹配整个
# \w+       ← 第一部分单词
```

```
# (?:-\w+)* ← 0次或多次的 "-单词"

\$?\d+(?:\.\d+)?%?  ← 数字、货币、百分比
# $12.40 → 匹配
# 82%    → 匹配
# \$?          ← 可选的美元符号
# \d+          ← 1个或多个数字
# (?:\.\d+)?   ← 可选的小数部分
# %?           ← 可选的百分号

\.\.\.  ← 省略号
# ...  → 匹配

[][.,;"'?():_`-]  ← 标点符号
# 各种标点，包括方括号
```

执行示例：

```
text = "The U.S.A. doesn't charge $10.5"
tokens = nltk.regexp_tokenize(text, pattern)
print(tokens)

# 输出:
# ['The', 'U.S.A.', 'does', "n't", 'charge', '$10.5']
```

方法3：用NLTK的内置分词器（最简单）

```
import nltk

text = "He says I'm depressed most of the time."
tokens = nltk.word_tokenize(text)
print(tokens)

# 输出:
# ['He', 'says', 'I', "'m", 'depressed', 'most', 'of', 'the', 'time',
```

优点： – 简单易用 – 处理了很多边缘情况 – 不需要自己写正则表达式

三、小写转换（Lowercasing）

方法1：用正则表达式替换

```
def lowercase_regex(text):
    return re.sub(r"[A-Z]", lambda x: x.group(0).lower(), text)
```

详细解释：

```
re.sub(r"[A-Z]", lambda x: x.group(0).lower(), text)
       ↑         ↑                                ↑
       |         |                                └─ 原文本
       |         └─ 替换函数：把匹配的字符转小写
       └─ pattern：匹配大写字母

lambda x: x.group(0).lower()
       ↑  ↑         ↑
       |  |         └─ 转小写
       |  └─ 获取匹配的字符
       └─ 匹配对象
```

执行过程：

```
text = "He Says HELLO"

# 匹配到：H, S, H, E, L, L, O
# 每个都调用 lambda 函数

# H → lambda(match_H) → match_H.group(0).lower() → 'h'
# S → lambda(match_S) → match_S.group(0).lower() → 's'
# ...

# 结果："he says hello"
```

方法2：用Python内置方法（更简单）

```
text = "He Says HELLO"
lowercased = text.lower()
print(lowercased)  # "he says hello"
```

四、Twitter特殊元素处理

Twitter文本的特点：

```
@username  ← 提到用户
#hashtag   ← 话题标签
:)         ← 表情符号（文本）
😊         ← 表情符号（emoji图像）
```

为什么要处理?

```
原文本:
"I love @Apple! #iPhone :)"
"I love @Google! #Pixel :)"

分词后:
['I', 'love', '@Apple', '!', '#iPhone', ':)']
['I', 'love', '@Google', '!', '#Pixel', ':)']

问题: @Apple 和 @Google 被认为是不同的词
     #iPhone 和 #Pixel 也是不同的词

但实际上，我们可能只关心:
- 有没有提到用户（不管是谁）
- 有没有用话题标签（不管什么话题）
- 情感如何（表情符号）
```

解决方案：标准化

```
@Apple  → <MENTIONS>
@Google → <MENTIONS>
#iPhone → <HASHTAGS>
```

```
#Pixel  → <HASHTAGS>
:)      → <EMOTICONS>
```

实现代码

```
def preprocess_tweet(text, include_emojis=False):
    # Pattern 1: @mentions
    mention_pattern = r"@[a-zA-Z0-9_]{0,15}"

    # Pattern 2: #hashtags
    hashtag_pattern = r"#(\w+)"

    # Pattern 3: 文本表情符号
    emoticon_pattern = r"(\:\w+\:|\<[\/\\]?3|[\(\)\\\D|\*\$][\-\^]?[\:

    # 替换（注意顺序很重要！）
    processed = re.sub(emoticon_pattern, "<EMOTICONS>", text)
    processed = re.sub(hashtag_pattern, "<HASHTAGS>", processed)
    processed = re.sub(mention_pattern, "<MENTIONS>", processed)

    return processed
```

详细解释：

Pattern 1: @mentions

```
mention_pattern = r"@[a-zA-Z0-9_]{0,15}"

@               ← @ 符号
[a-zA-Z0-9_]    ← 字母、数字、下划线
{0,15}          ← 0到15个字符（Twitter用户名限制）

例子:
@username123 → 匹配
@a_b_c       → 匹配
```

Pattern 2: #hashtags

```
hashtag_pattern = r"#(\w+)"

#     ← # 符号
(\w+) ← 1个或多个单词字符（捕获组）

例子:
#iPhone  → 匹配, group(1)='iPhone'
#AI2023  → 匹配, group(1)='AI2023'
```

为什么用捕获组 `(\w+)`？

```
# 不用捕获组:
re.sub(r"#\w+", "<HASHTAGS>", text)
# 结果: 完全替换 #iPhone → <HASHTAGS>

# 用捕获组:
re.sub(r"#(\w+)", "<HASHTAGS>", text)
# 结果: 也是完全替换

# 实际上这里捕获组不必要，但保留下来是为了:
# 1. 可能以后需要访问hashtag内容
# 2. 代码更清晰
```

Pattern 3: 文本表情符号（非常复杂！）

```
emoticon_pattern = r"(\:\w+\:|\<[\/\\]?3|[\(\)\\\D|\*\$][\-\^]?[\:\;\=
```

这个太复杂，我们分解:

```
# 简化版:
emoticon_pattern = r"(\:\w+\:)"  # 只匹配 :smile:
                 + r"|"          # 或
                 + r"[\:\;\=][\)\(]"  # :) :( =) =(
```

分解详细版本:

```
第1部分: \:\w+\:
  匹配: :smile: :joy: :cry:

第2部分: \<[\/\\]?3
  匹配: <3 </3 <\3 (心形)

第3部分: [\(\)\\\D|\*\$][\-\^]?[\:\;\=]
  匹配各种复杂表情, 如 :-)  ;)  =(

第4部分: [\:\;\=B8][\-\^]?[3DOPp\@\$\*\\\)\(\/\|]
  匹配更多变体

最后: (?=\s|[\!\.\?]|$)
  ← 前瞻断言 (Lookahead assertion)
  确保表情符号后面是空格、标点或结尾
```

前瞻断言 `(?=...)` 解释:

```
(?=...) ← 正向前瞻 (Positive lookahead)

意思: 后面必须匹配..., 但不消耗字符

例子:
text = "happy:)sad"

pattern = r":\)(?=\s)"
# 不匹配! 因为 :) 后面不是空格

text = "happy:) sad"
pattern = r":\)(?=\s)"
# 匹配! 因为 :) 后面是空格
# 但匹配的只是 :), 不包括空格
```

替换顺序为什么重要?

```
text = "Love @user #tag :)"

# 错误顺序1: 先替换mentions
processed = re.sub(mention_pattern, "<MENTIONS>", text)
# 结果: "Love <MENTIONS> #tag :)"
```

```
processed = re.sub(hashtag_pattern, "<HASHTAGS>", processed)
# 结果："Love <MENTIONS> <HASHTAGS> :)"

# 问题：如果hashtag pattern写错，可能匹配到 <MENTIONS> 的一部分

# 推荐顺序：从最特殊到最一般
# 1. emoticons（最特殊，符号容易和其他冲突）
# 2. hashtags
# 3. mentions
```

Emoji处理（可选挑战）

Emoji是什么？

```
😊 ← 这是emoji（Unicode字符）
:) ← 这是emoticon（文本符号）
```

匹配Emoji的pattern：

```
emoji_pattern = re.compile(pattern = "["
    u"\U0001F600-\U0001F64F"  # Emoticons range
    u"\U0001F300-\U0001F5FF"  # Symbols & pictographs
    u"\U0001FAC0-\U0001FaFF"  # Extended symbols
    u"\U00012600-\U000126FF"  # Miscellaneous
    u"\U00002700-\U000027BF"  # Dingbats
    u"\U0001F100-\U0001F1FF"  # Alphanumeric supplement
    u"\U0001F680-\U0001F6FF"  # Transport & map
    u"\U0001F1E0-\U0001F1FF"  # Flags
    u"\U00002190-\U000021FF"  # Arrows
    "]+", flags = re.UNICODE)
```

Unicode范围解释：

```
\U0001F600-\U0001F64F
  ↑        ↑
  |        └─ 结束码点
  └─ 开始码点

U+1F600 = 😀 (grinning face)
```

```
U+1F64F = 🙏 (folded hands)

这个范围包含所有"脸部表情"emoji
```

使用：

```
text = "I love Python! 😊🎉"
processed = emoji_pattern.sub("<EMOJIS>", text)
print(processed)
# "I love Python! <EMOJIS><EMOJIS>"
```

五、类型–标记比率（Type–Token Ratio, TTR）

概念

Types（类型）：不同单词的数量（词汇量）

```
text = "the cat and the dog"
tokens = ['the', 'cat', 'and', 'the', 'dog']

types = set(tokens)  # {'the', 'cat', 'and', 'dog'}
num_types = len(types)  # 4
```

Tokens（标记）：所有单词的总数

```
num_tokens = len(tokens)  # 5
```

TTR（类型–标记比率）：

```
TTR = Types / Tokens = 4 / 5 = 0.8
```

意义：

```
高TTR（接近1）：词汇丰富，很少重复
  例如：学术论文
```

```
低TTR（接近0）：词汇简单，大量重复
  例如：儿童读物
```

实现代码

```
def count_types_and_tokens(tweets_list):
    types = set()
    tokens = []

    for tweet in tweets_list:
        for token in tweet:
            # 只统计包含字母或数字的token（排除标点）
            if re.search("[a-zA-Z0-9]+", token):
                types.add(token)
                tokens.append(token)

    return len(types), len(tokens)
```

详细解释：

```
types = set()  # 集合，自动去重
tokens = []    # 列表，保留重复

for tweet in tweets_list:
    # tweets_list是列表的列表
    # 例如：[['I', 'love', 'Python'], ['Python', 'is', 'great']]

    for token in tweet:
        # 检查token是否包含字母或数字
        if re.search("[a-zA-Z0-9]+", token):
            types.add(token)       # 集合自动去重
            tokens.append(token)   # 列表保留所有
```

为什么要检查 `[a-zA-Z0-9]+`？

```
tokens = ['I', 'love', 'Python', '.', '!']

# 不检查：
types = {'I', 'love', 'Python', '.', '!'}  # 5个types
```

```
tokens_count = 5
TTR = 5/5 = 1.0  ← 但标点不算"词汇"!

# 检查后:
types = {'I', 'love', 'Python'}  # 3个types
tokens_count = 3
TTR = 3/3 = 1.0  ← 更准确
```

计算TTR

```
def print_config_ttr(config_string, processed_tweets):
    num_types, num_tokens = count_types_and_tokens(processed_tweets)
    ttr = round(num_types / num_tokens, 3)
    print(f"{config_string}: {ttr} ({num_types} types, {num_tokens} to
```

执行示例:

```
tweets = [
    "I love Python",
    "I love coding",
    "Python is great"
]

# 配置1: 无预处理
tweets_split = [tweet.split(" ") for tweet in tweets]
print_config_ttr("No preprocessing", tweets_split)

# 配置2: 分词
tweets_tokenized = [nltk.word_tokenize(tweet) for tweet in tweets]
print_config_ttr("Tokenised", tweets_tokenized)

# 配置3: 分词 + 小写
tweets_lower = [nltk.word_tokenize(tweet.lower()) for tweet in tweets]
print_config_ttr("Tokenised + lowercased", tweets_lower)

# 输出可能是:
# No preprocessing: 0.778 (7 types, 9 tokens)
# Tokenised: 0.778 (7 types, 9 tokens)
# Tokenised + lowercased: 0.714 (5 types, 7 tokens)
```

为什么TTR会下降?

```
# 配置1: 无预处理
tokens = ['I', 'love', 'Python', 'I', 'love', 'coding', 'Python', 'is'
types = {'I', 'love', 'Python', 'coding', 'is', 'great'}  # 6个
# （去掉标点后）
TTR = 6 / 9 = 0.667

# 配置3: 小写
tokens = ['i', 'love', 'python', 'i', 'love', 'coding', 'python', 'is'
types = {'i', 'love', 'python', 'coding', 'is', 'great'}  # 6个
# 'I' 和 'i' 合并了! 'Python' 和 'python' 合并了!
# 但在这个例子中，原本都是小写，所以types数量不变

# 但如果有:
# "I love Python" 和 "i love python"
# 不小写: types = {'I', 'love', 'Python', 'i', 'python'}  # 5个
# 小写后: types = {'i', 'love', 'python'}  # 3个
# types减少 → TTR下降
```

对下游任务的影响:

```
TTR下降 = 词汇量减少 = 每个词出现频率增加

优点:
1. 更容易找到模式
2. 稀疏性降低
3. 模型训练更稳定

缺点:
1. 可能丢失信息（如 'Apple' 公司 vs 'apple' 水果）
2. 情感分析可能受影响（'LOVE' vs 'love' 强度不同）
```

六、完整执行流程示例

输入数据:

```
tweets = [
    "He says I'm depressed most of the time. #sad",
```

```
    "For the first time I get to see @username actually being hateful!
    '"The San Francisco-based restaurant" they said, "doesn\'t charge
]
```

步骤1：原始分词

```
for tweet in tweets:
    tokens = tweet.split(" ")
    print(tokens)

# 输出:
# ['He', 'says', "I'm", 'depressed', 'most', 'of', 'the', 'time.', '#s
# ['For', 'the', 'first', 'time', 'I', 'get', 'to', 'see', '@username'
```

问题： – `"I'm"` 没有分开 – `"time."` 包含标点 – `"#sad"` 包含#号

步骤2：用NLTK分词

```
for tweet in tweets:
    tokens = nltk.word_tokenize(tweet)
    print(tokens)

# 输出:
# ['He', 'says', 'I', "'m", 'depressed', 'most', 'of', 'the', 'time',
# ...
```

改进： – `"I'm"` 分成了 `'I'` 和 `"'m"` – 标点独立了

步骤3：小写 + 分词

```
for tweet in tweets:
    lowercased = tweet.lower()
    tokens = nltk.word_tokenize(lowercased)
    print(tokens)
```

```
# 输出:
# ['he', 'says', 'i', "'m", 'depressed', 'most', 'of', 'the', 'time',
```

步骤4：预处理 + 小写 + 分词

```
for tweet in tweets:
    lowercased = tweet.lower()
    preprocessed = preprocess_tweet(lowercased)
    tokens = nltk.word_tokenize(preprocessed)
    print(tokens)

# 输出:
# ['he', 'says', 'i', "'m", 'depressed', 'most', 'of', 'the', 'time',
# ['for', 'the', 'first', 'time', 'i', 'get', 'to', 'see', '<MENTIONS>
# ['"', 'the', 'san', 'francisco-based', 'restaurant', '"', 'they', 's
```

改进： – `#sad` → `<HASHTAGS>` – `@username` → `<MENTIONS>` – `:)` → `<EMOTICONS>`

步骤5：计算TTR

```
# 配置1: 无预处理
tweets_split = [tweet.split(" ") for tweet in tweets]
num_types, num_tokens = count_types_and_tokens(tweets_split)
print(f"TTR: {num_types / num_tokens:.3f}")

# 配置4: 完整预处理
processed_tweets = [
    nltk.word_tokenize(preprocess_tweet(tweet.lower()))
    for tweet in tweets
]
num_types, num_tokens = count_types_and_tokens(processed_tweets)
print(f"TTR: {num_types / num_tokens:.3f}")
```

结果分析：

```
无预处理:
  types = 很多（每个@username, #hashtag都算不同的词）
  TTR = 高

完整预处理:
  types = 较少（所有@username都变成<MENTIONS>）
  TTR = 低

TTR降低是好事!
→ 词汇量减少
→ 每个词出现频率增加
→ 机器学习模型更容易学习模式
```

七、纸笔考试重点

必须掌握

1. ✅ 贪婪 vs 非贪婪匹配

```
贪婪: .*    → 尽可能多匹配
非贪婪: .*? → 尽可能少匹配

<p><b>text</b>
  .*  匹配: p><b>text</b
  .*? 匹配: p
```

2. ✅ 字符类取反 `[^...]`

```
[^>]+ → 匹配1个或多个"非>"字符
[^">\s]+ → 匹配1个或多个"非引号、非>、非空格"字符
```

3. ✅ 反向引用 `\1`, `\2`

```
r'<(\w+)>(.*?)</\1>'
   ↑          ↑
  第1组      必须匹配第1组的内容
```

```
<b>text</b> → 匹配（开闭标签都是b）
<b>text</i> → 不匹配（开闭标签不同）
```

4. ✅ 前瞻断言 `(?=...)`

```
(?=...) → 后面必须是...，但不消耗字符
(?!...) → 后面不能是...

pattern = r":\)(?=\s)"
"happy:) " → 匹配（后面是空格）
"happy:)." → 不匹配（后面是点号）
```

5. ✅ 非捕获组 `(?:...)`

```
(...)   → 捕获组，可用group(1)提取
(?:...) → 非捕获组，只用于分组，不提取

r"\w+(?:'\w+)?"
      ↑
      非捕获，不会被group(1)提取
```

6. ✅ TTR计算

```
TTR = Types / Tokens

Types = 不同单词数量
Tokens = 总单词数量

例子:
"the cat and the dog"
Types = 4 (the, cat, and, dog)
Tokens = 5
TTR = 4/5 = 0.8
```

7. ✅ 预处理对TTR的影响

```
预处理步骤越多 → Types越少 → TTR越低

原因：
1. 小写转换：'The' 和 'the' 合并
2. 标准化：@user1, @user2 都变成 <MENTIONS>
```

可能的考题

题型1：写正则表达式

题目：写一个正则表达式匹配HTML的 `<img>` 标签，提取src属性

```
<img src="photo.jpg" width="100">
```

▶ 答案

题型2：贪婪匹配问题

题目：为什么下面的正则表达式不能正确匹配每个单独的标签？

```
text = "<p><b>text</b></p>"
pattern = '<.*>'
# 结果：匹配整个字符串
```

如何修复？

▶ 答案

题型3：反向引用

题目：用反向引用写一个正则表达式，匹配重复的单词

```
输入："the the cat"
应该匹配："the the"
```

▶ 答案

题型4：计算TTR

题目：

给定tweets：

```
tweets = [
    "I love Python",
    "I LOVE coding",
    "Python is great"
]
```

计算： 1. 无预处理的TTR（用`.split()`分词） 2. 小写后的TTR

▶ 答案

题型5：Twitter预处理

题目：写正则表达式把所有@mentions替换成`<MENTIONS>`

```
text = "Hey @user1 and @user2, check this out!"
```

▶ 答案

八、记忆口诀

贪婪与非贪婪

```
星号贪婪吃到底，
问号节制刚刚好，
HTML标签分不清，
加个问号解烦恼。
```

反向引用

```
括号捕获第一组，
反斜数字再引用，
开闭标签要配对，
\1帮你来把关。
```

文本预处理

```
大写小写要统一，
标点符号需分离，
Twitter元素标准化，
TTR越低越好记。
```

九、常见错误

❌ 错误1：忘记用非贪婪匹配

```
# 错误
pattern = '<.*>'  # 匹配整个字符串
text = "<p><b>text</b></p>"
# 结果：<p><b>text</b></p>  ← 不是我们想要的！

# 正确
pattern = '<.*?>'  # 非贪婪
# 结果：<p>, <b>, </b>, </p>  ← 每个标签单独匹配
```

❌ 错误2：反向引用不用raw string

```
# 错误
pattern = '<(\w+)>(.*?)</\1>'
# Python会尝试解释 \1 作为转义字符（报错）

# 正确
pattern = r'<(\w+)>(.*?)</\1>'
# r'...' 保留 \1 作为正则表达式的反向引用
```

❌ 错误3：捕获组和非捕获组混淆

```
text = "don't"
pattern = r"\w+(?:'\w+)?"

# 执行
m = re.search(pattern, text)
print(m.group(0))  # "don't"  ← 整个匹配
print(m.group(1))  # 错误！没有group(1)

# (?:...) 是非捕获组，不会被编号
```

❌ 错误4：TTR计算忘记排除标点

```
# 错误
tokens = ['I', 'love', 'Python', '.', '!']
types = set(tokens)  # 5个（包括标点）
TTR = 5/5 = 1.0  ← 不准确！

# 正确
tokens = [t for t in tokens if re.search("[a-zA-Z0-9]+", t)]
# ['I', 'love', 'Python']
types = set(tokens)  # 3个
TTR = 3/3 = 1.0  ← 准确！
```

十、扩展知识

1. Unicode属性（高级）

```
# 匹配所有汉字
pattern = r'[\u4e00-\u9fff]+'

# 匹配所有emoji
pattern = r'[\U0001F600-\U0001F64F]+'
```

---

2. Verbose模式（让正则表达式更易读）

```
# 不用verbose（难读）
pattern = r'(?:[A-Z]\.)+|\w+(?:-\w+)*|\$?\d+(?:\.\d+)?%?|\.\.\.|\[.,;"'

# 用verbose（易读）
pattern = r'''(?x)
    (?:[A-Z]\.)+         # 缩写，如 U.S.A.
  | \w+(?:-\w+)*         # 带连字符的词
  | \$?\d+(?:\.\d+)?%?   # 货币和百分比
  | \.\.\.               # 省略号
  | [].,;"'?():_`-]      # 标点
'''
```

---

3. 实际应用场景

场景1：提取所有邮箱

```
pattern = r'\b[A-Za-z0-9._%+-]+@[A-Za-z0-9.-]+\.[A-Z|a-z]{2,}\b'
```

场景2：验证电话号码

```
pattern = r'^\+?1?\d{9,15}$'
```

场景3：清理HTML

```
clean_text = re.sub(r'<[^>]+>', '', html_text)
```

祝你考试顺利！ 🎉

Lecture 5核心要点： – 贪婪vs非贪婪：`.*` vs `.*?` – 字符类取反：`[^>]+` – 反向引用：`\1`，`\2`（必须用`r'...'`） – 前瞻断言：`(?=...)` – 非捕获组：`(?:...)` – TTR = Types / Tokens – 预处理降低TTR，有利于机器学习

记住：理解每个符号的含义，能手工模拟匹配过程！