Lecture 3 超基础复习笔记 - 词性标注（POS Tagging）

零、先理解问题（什么是词性标注？）

什么是词性（Part of Speech）？

词性 = 单词在句子中的语法角色

中文例子：

> 我（代词） 爱（动词） 吃（动词） 苹果（名词）

英文例子：

```
I     love   eating  apples
代词   动词   动词     名词
PRP   VBP    VBG     NNS
```

为什么需要词性标注?

应用场景：

1. **机器翻译** ``` "book" 可以是：
2. 名词：I read a book（书）
3. 动词：I will book a ticket（预订）

需要知道词性才能正确翻译！ ```

1. **语音合成** ``` "read" 的发音取决于词性：

2. 现在时动词：read [ri:d]（读）

3. 过去时：read [red]（已读） ```

4. **信息提取** 找出句子中的所有人名： 通常人名是专有名词（NNP标签）

一、数据格式（Brill Format）

原始格式

训练数据示例：

```
The/DT dog/NN is/VBZ running/VBG ./.
I/PRP love/VBP Python/NNP ./.
```

格式说明：

```
单词/词性标签 单词/词性标签 ...

The/DT
 ↑   ↑
单词 标签
```

常见词性标签（必须记住！）

标签	全称	中文	例子
NN	Noun (singular)	名词（单数）	dog, cat, book
NNS	Noun (plural)	名词（复数）	dogs, cats, books
NNP	Proper noun (singular)	专有名词	John, London, Python
VB	Verb (base form)	动词（原形）	run, eat, go
VBG	Verb (gerund/present participle)	动词（进行时）	running, eating, going
VBD	Verb (past tense)	动词（过去时）	ran, ate, went
VBZ	Verb (3rd person singular)	动词（第三人称）	runs, eats, goes
JJ	Adjective	形容词	good, big, happy
RB	Adverb	副词	quickly, very, well
PRP	Personal pronoun	人称代词	I, you, he, she
DT	Determiner	限定词	the, a, an, this
IN	Preposition	介词	in, on, at, with
CD	Cardinal number	基数词	1, 2, three, 100
.	Punctuation	标点符号	. ! ? ,

二、两层字典（核心数据结构！）

为什么需要两层字典?

问题：一个单词可能有多个词性!

```
"book" 在训练数据中:
- 作为名词 (NN) 出现了 50 次
- 作为动词 (VB) 出现了 10 次
```

需要存储:

```
book → NN: 50次
       VB: 10次
```

两层字典结构

```
word_tag_counts = {
    单词1: {标签A: 次数, 标签B: 次数},
    单词2: {标签C: 次数},
    ...
}
```

具体例子:

```
word_tag_counts = {
    'book': {'NN': 50, 'VB': 10},
    'running': {'VBG': 30, 'NN': 5},
    'the': {'DT': 1000},
    'Python': {'NNP': 20}
}
```

图解:

```
word_tag_counts
    |
    ├─ "book" ──── {'NN': 50, 'VB': 10}
    |                  ↑          ↑
    |              第2层字典    第2层字典
    |
    ├─ "running" ─ {'VBG': 30, 'NN': 5}
    |
    └─ "the" ────── {'DT': 1000}
```

三、构建两层字典（逐步详解）

整体流程

1. 读取文件的每一行
2. 分割每行为 "单词/标签" 对
3. 分离单词和标签
4. 更新两层字典的计数

代码详细讲解

第1步：解析一行数据

```
def parse_line(line):
    wdtags = line.split()
    wdtagpairs = []
    for wdtag in wdtags:
        parts = wdtag.split('/')
        wdtagpairs.append((parts[0], parts[1]))
    return wdtagpairs
```

执行示例：

输入：

```
line = "The/DT dog/NN runs/VBZ ./."
```

步骤1：按空格分割

```
wdtags = line.split()
# wdtags = ['The/DT', 'dog/NN', 'runs/VBZ', './.']
```

步骤2：遍历每个词标签对

```
# 第1次循环
wdtag = 'The/DT'
parts = wdtag.split('/')  # ['The', 'DT']
wdtagpairs.append(('The', 'DT'))

# 第2次循环
wdtag = 'dog/NN'
parts = wdtag.split('/')  # ['dog', 'NN']
wdtagpairs.append(('dog', 'NN'))

# ... 继续

# 最终结果
wdtagpairs = [('The', 'DT'), ('dog', 'NN'), ('runs', 'VBZ'), ('.', '.')]
```

返回值类型：

```
# 列表，里面是元组（tuple）
[('单词1', '标签1'), ('单词2', '标签2'), ...]
```

第2步：更新两层字典

```
word_tag_counts = {}

for line in data_in:
    for (wd, tag) in parse_line(line):
        if wd not in word_tag_counts:
            word_tag_counts[wd] = {}
        if tag in word_tag_counts[wd]:
            word_tag_counts[wd][tag] += 1
        else:
            word_tag_counts[wd][tag] = 1
```

超详细执行过程：

假设处理两行数据：

```
I/PRP love/VBP Python/NNP ./.
I/PRP love/VBP coding/VBG ./.
```

初始状态：

```
word_tag_counts = {}
```

处理第1个词：('I', 'PRP')

```
wd = 'I'
tag = 'PRP'

# 检查：'I' 在字典里吗？
if wd not in word_tag_counts:  # True（还是空字典）
    word_tag_counts[wd] = {}  # 创建第2层字典
    # 现在：word_tag_counts = {'I': {}}

# 检查：'PRP' 在 word_tag_counts['I'] 里吗？
if tag in word_tag_counts[wd]:  # False（第2层字典是空的）
    # 不执行
else:
    word_tag_counts[wd][tag] = 1
    # 现在：word_tag_counts = {'I': {'PRP': 1}}
```

处理第2个词：('love', 'VBP')

```
wd = 'love'
tag = 'VBP'

# 'love' 在字典里吗？
if wd not in word_tag_counts:  # True
    word_tag_counts[wd] = {}
    # 现在：word_tag_counts = {'I': {'PRP': 1}, 'love': {}}

# 'VBP' 在 word_tag_counts['love'] 里吗？
if tag in word_tag_counts[wd]:  # False
else:
    word_tag_counts[wd][tag] = 1
    # 现在：word_tag_counts = {'I': {'PRP': 1}, 'love': {'VBP': 1}}
```

处理第3个词：('Python', 'NNP')

```
word_tag_counts = {
    'I': {'PRP': 1},
    'love': {'VBP': 1},
    'Python': {'NNP': 1}
}
```

处理第4个词：('.', '.')

```
word_tag_counts = {
    'I': {'PRP': 1},
    'love': {'VBP': 1},
    'Python': {'NNP': 1},
    '.': {'.': 1}
}
```

继续处理第2行数据...

再次遇到 ('I', 'PRP')：

```
wd = 'I'
tag = 'PRP'

# 'I' 在字典里吗?
if wd not in word_tag_counts:  # False（已经存在了！）
    # 不执行

# 'PRP' 在 word_tag_counts['I'] 里吗?
if tag in word_tag_counts[wd]:  # True（已经有了！）
    word_tag_counts[wd][tag] += 1  # 1变成2
    # 现在：word_tag_counts['I']['PRP'] = 2
else:
    # 不执行
```

再次遇到 ('love', 'VBP')：

```
# 同样的逻辑
word_tag_counts['love']['VBP'] += 1  # 1变成2
```

遇到新词 ('coding', 'VBG')：

```
wd = 'coding'
tag = 'VBG'

# 'coding' 不在字典里
word_tag_counts['coding'] = {}
word_tag_counts['coding']['VBG'] = 1
```

最终结果：

```
word_tag_counts = {
    'I': {'PRP': 2},           # I出现2次，都是PRP
    'love': {'VBP': 2},        # love出现2次，都是VBP
    'Python': {'NNP': 1},      # Python出现1次，是NNP
    'coding': {'VBG': 1},      # coding出现1次，是VBG
    '.': {'.': 2}              # 句号出现2次
}
```

为什么要这样设计?

对比：只用一层字典

错误做法：

```
# 一层字典
counts = {'book': 60}  # 怎么区分NN和VB? ? ?
```

问题： - 无法记录同一个词的不同词性 - 丢失了重要信息!

正确做法：两层字典

```
word_tag_counts = {
    'book': {'NN': 50, 'VB': 10}  # 清楚记录!
}
```

优点： - 保留所有信息 - 可以计算每个词的最常见标签 - 可以统计歧义词的比例

四、朴素词性标注算法

核心思想（超级简单！）

算法：给每个词分配它最常见的词性

```
"book" → 在训练数据中:
  - NN: 50次  ← 最多!
  - VB: 10次

决定：遇到"book"就标注为 NN
```

算法步骤

1. 从训练数据学习：统计每个词的各种词性出现次数
2. 对每个词，找出出现最多的词性
3. 给新数据标注时，查表分配最常见词性
4. 如果遇到未知词（训练数据没见过），用特殊策略

第1步：找出每个词的最常见标签

```
maxtag = {}
for wd in word_tag_counts:
    tags = sorted(word_tag_counts[wd],
                  key=lambda x:word_tag_counts[wd][x],
                  reverse=True)
    maxtag[wd] = tags[0]
```

详细解释：

假设：

```
word_tag_counts = {
    'book': {'NN': 50, 'VB': 10},
    'running': {'VBG': 30, 'NN': 5},
    'the': {'DT': 1000}
}
```

处理 'book'：

```
wd = 'book'
word_tag_counts[wd] = {'NN': 50, 'VB': 10}

# 排序标签，按出现次数降序
tags = sorted(['NN', 'VB'],
              key=lambda x: word_tag_counts['book'][x],
              reverse=True)
```

`sorted()` 的工作过程：

```
# 比较 'NN' 和 'VB'
# lambda x: word_tag_counts['book'][x]
# x='NN' → word_tag_counts['book']['NN'] = 50
# x='VB' → word_tag_counts['book']['VB'] = 10

# 按次数排序（降序）
tags = ['NN', 'VB']  # NN的次数(50) > VB的次数(10)

# 取第一个（最多的）
maxtag['book'] = tags[0]  # 'NN'
```

处理 'running'：

```
wd = 'running'
word_tag_counts['running'] = {'VBG': 30, 'NN': 5}

# VBG出现30次，NN出现5次
tags = ['VBG', 'NN']
maxtag['running'] = 'VBG'
```

最终结果：

```
maxtag = {
    'book': 'NN',      # book最常见的是名词
    'running': 'VBG',  # running最常见的是动名词
    'the': 'DT'        # the总是限定词
}
```

第2步：标注测试数据

```
for line in test_file:
    for wd, truetag in parse_line(line):
        if wd in maxtag:
            newtag = maxtag[wd]  # 查表
        else:
            newtag = tag_unknown(wd)  # 未知词

        # 比较正确率
        if newtag == truetag:
            correct += 1
        alltest += 1
```

执行示例：

测试数据：

```
The/DT book/NN is/VBZ interesting/JJ ./.
```

标注过程：

```
# 词1: "The"
wd = 'The'
truetag = 'DT'  # 正确答案
newtag = maxtag['The'] = 'DT'  # 我们的预测
newtag == truetag  # True ✅ 正确!
correct += 1

# 词2: "book"
wd = 'book'
truetag = 'NN'
newtag = maxtag['book'] = 'NN'
newtag == truetag  # True ✅ 正确!
correct += 1

# 词3: "is"
wd = 'is'
truetag = 'VBZ'
newtag = maxtag['is'] = 'VBZ'
newtag == truetag  # True ✅ 正确!
correct += 1

# 词4: "interesting" (假设训练数据没见过)
wd = 'interesting'
truetag = 'JJ'
wd in maxtag  # False! 未知词
newtag = tag_unknown('interesting')  # 用猜测策略
#  (取决于具体实现)
```

五、处理未知词（重要！）

问题

> 训练数据：只见过 10,000 个不同的词
> 测试数据：出现了新词 "coronavirus"
>
> 怎么办?

策略1：全部标记为UNK（最差）

```
def tag_unknown(wd):
    return 'UNK'   # 未知标签（肯定错误）
```

结果：所有未知词都错误

策略2：使用最常见标签

```
def tag_unknown(wd):
    return 'NN'   # 名词最常见
```

原理： - 统计发现：英语中名词最常见（约30%） - 猜名词有30%的正确率

策略3：基于规则猜测（最好！）

```
def tag_unknown(wd):
    # 规则1：首字母大写 → 专有名词
    if wd[0].isupper():
        return 'NNP'

    # 规则2：包含连字符 → 形容词
    if '-' in wd:
        return 'JJ'

    # 规则3：包含数字 → 基数词
    if any(char.isdigit() for char in wd):
        return 'CD'

    # 规则4：以-ing结尾 → 动名词
    if wd.endswith('ing'):
        return 'VBG'

    # 规则5：以-ly结尾 → 副词
    if wd.endswith('ly'):
        return 'RB'

    # 规则6：以-s结尾 → 复数名词
    if wd.endswith('s'):
        return 'NNS'

    # 规则7：以-ed结尾 → 过去式动词
    if wd.endswith('ed'):
        return 'VBD'

    # 默认：名词
    return 'NN'
```

规则详解：

规则1：首字母大写 → NNP（专有名词）

```
if wd[0].isupper():
    return 'NNP'
```

例子：

```
Beijing → 首字母B大写 → NNP（北京）
Python → 首字母P大写 → NNP（Python语言）
John → 首字母J大写 → NNP（人名）
```

为什么有效？ - 英语中，专有名词（地名、人名、品牌名）首字母大写 - 准确率很高！

规则2：包含连字符 → JJ（形容词）

```
if '-' in wd:
    return 'JJ'
```

例子：

```
well-known → 包含'-' → JJ（著名的）
state-of-the-art → 包含'-' → JJ（最先进的）
```

规则3：包含数字 → CD（基数词）

```
digitRE = re.compile('\d')  # 匹配任意数字
if digitRE.search(wd):
    return 'CD'
```

例子：

```
123 → 包含数字 → CD
3.14 → 包含数字 → CD
year2023 → 包含数字 → CD
```

规则4：后缀-ing → VBG（动名词）

```
if wd.endswith('ing'):
    return 'VBG'
```

例子：

```
running → 以ing结尾 → VBG
swimming → 以ing结尾 → VBG
interesting → 以ing结尾 → VBG
```

注意：不是100%准确

```
thing → 以ing结尾，但是名词NN，不是VBG
king → 以ing结尾，但是名词NN

但大部分情况下VBG是对的!
```

规则5：后缀-ly → RB（副词）

```
if wd.endswith('ly'):
    return 'RB'
```

例子：

```
quickly → 以ly结尾 → RB（快速地）
happily → 以ly结尾 → RB（快乐地）
slowly → 以ly结尾 → RB（慢慢地）
```

规则的优先级

为什么顺序重要?

```
# 假设处理词："Running"（首字母大写）

# 如果先检查-ing：
if wd.endswith('ing'):  # True
    return 'VBG'  # 返回VBG（可能错！）

# 如果先检查大写：
if wd[0].isupper():  # True
    return 'NNP'  # 返回NNP（专有名词，可能对！）
```

建议顺序（代码中的顺序）：

```
1. 首字母大写（最特殊）
2. 包含连字符
3. 包含数字
4. 后缀规则（-ed, -ing, -ly等）
5. 默认值
```

六、计算准确率和统计

准确率（Accuracy）

公式：

```
准确率 = 正确标注的词数 / 总词数 × 100%
```

例子：

```
测试数据有100个词
正确标注了85个

准确率 = 85 / 100 = 0.85 = 85%
```

代码实现

```
correct = 0  # 正确数量
alltest = 0  # 总数量

for line in test_file:
    for wd, truetag in parse_line(line):
        newtag = predict_tag(wd)  # 预测标签
        alltest += 1
        if newtag == truetag:
            correct += 1

accuracy = (correct / alltest) * 100
print(f"准确率: {accuracy:.1f}%")
```

歧义词统计

什么是歧义词?

一个词有多个可能的词性

```
word_tag_counts = {
    'book': {'NN': 50, 'VB': 10},  # 歧义词（2个标签）
    'the': {'DT': 1000},           # 非歧义词（1个标签）
    'can': {'MD': 20, 'NN': 5}     # 歧义词（2个标签）
}
```

统计歧义词比例:

```
ambiguous_types = 0  # 歧义词种类数
all_types = len(word_tag_counts)  # 总词种类数

for wd in word_tag_counts:
    tags = word_tag_counts[wd]
    if len(tags) > 1:  # 有多个标签
        ambiguous_types += 1

proportion = (ambiguous_types / all_types) * 100
print(f"歧义词比例: {proportion:.1f}%")
```

执行示例：

```
word_tag_counts = {
    'book': {'NN': 50, 'VB': 10},   # len=2, 歧义
    'the': {'DT': 1000},             # len=1, 非歧义
    'can': {'MD': 20, 'NN': 5},      # len=2, 歧义
    'run': {'VB': 30, 'NN': 10}      # len=2, 歧义
}

all_types = 4
ambiguous_types = 3  # book, can, run

proportion = 3/4 = 0.75 = 75%
```

七、完整执行示例（手工模拟）

训练数据

```
I/PRP love/VBP Python/NNP ./.
I/PRP love/VBP coding/VBG ./.
The/DT book/NN is/VBZ good/JJ ./.
I/PRP book/VB tickets/NNS ./.
```

步骤1：构建word_tag_counts

逐行处理：

第1行：

```
word_tag_counts = {
    'I': {'PRP': 1},
    'love': {'VBP': 1},
    'Python': {'NNP': 1},
    '.': {'.': 1}
}
```

第2行：

```
word_tag_counts = {
    'I': {'PRP': 2},        # +1
    'love': {'VBP': 2},     # +1
    'Python': {'NNP': 1},
    'coding': {'VBG': 1},   # 新词
    '.': {'.': 2}           # +1
}
```

第3行：

```
word_tag_counts = {
    'I': {'PRP': 2},
    'love': {'VBP': 2},
    'Python': {'NNP': 1},
    'coding': {'VBG': 1},
    'The': {'DT': 1},       # 新词
    'book': {'NN': 1},      # 新词
    'is': {'VBZ': 1},       # 新词
    'good': {'JJ': 1},      # 新词
    '.': {'.': 3}           # +1
}
```

第4行：

```
word_tag_counts = {
    'I': {'PRP': 3},        # +1
    'love': {'VBP': 2},
    'Python': {'NNP': 1},
    'coding': {'VBG': 1},
    'The': {'DT': 1},
    'book': {'NN': 1, 'VB': 1},  # 'book'作为VB出现!
    'is': {'VBZ': 1},
    'good': {'JJ': 1},
    'tickets': {'NNS': 1},  # 新词
    '.': {'.': 4}           # +1
}
```

步骤2：计算最常见标签

```
maxtag = {
    'I': 'PRP',       # 只有1个标签
    'love': 'VBP',    # 只有1个标签
    'Python': 'NNP',  # 只有1个标签
    'coding': 'VBG',  # 只有1个标签
    'The': 'DT',      # 只有1个标签
    'book': 'NN',     # NN:1, VB:1 → 都是1, 取第一个
    'is': 'VBZ',      # 只有1个标签
    'good': 'JJ',     # 只有1个标签
    'tickets': 'NNS', # 只有1个标签
    '.': '.'          # 只有1个标签
}
```

步骤3：标注测试数据

测试数据：

```
I/PRP book/VB a/DT flight/NN ./.
```

标注过程：

```
# 词1: "I"
wd = 'I', truetag = 'PRP'
newtag = maxtag['I'] = 'PRP'  ✅ 正确
correct = 1

# 词2: "book"
wd = 'book', truetag = 'VB'
newtag = maxtag['book'] = 'NN'  ❌ 错误（我们猜的是名词）
correct = 1（不变）

# 词3: "a"（未知词，训练数据没有）
wd = 'a', truetag = 'DT'
wd not in maxtag  # True
newtag = tag_unknown('a')
# 'a'不满足任何规则，返回默认'NN'
newtag = 'NN'  ❌ 错误
correct = 1

# 词4: "flight"（未知词）
wd = 'flight', truetag = 'NN'
newtag = tag_unknown('flight')
# 不满足特殊规则，返回默认'NN'
newtag = 'NN'  ✅ 正确（运气好！）
correct = 2

# 词5: "."
wd = '.', truetag = '.'
newtag = maxtag['.'] = '.'  ✅ 正确
correct = 3

# 总结
alltest = 5
correct = 3
accuracy = 3/5 = 0.6 = 60%
```

八、算法局限性

问题1：无法利用上下文

```
句子1: "I will book a ticket"
句子2: "I read a book"

朴素标注器:
都标注 "book" 为 NN（最常见）

但句子1中应该是VB（动词）！
```

为什么？ - 只看单词本身，不看前后文 - 不知道 "will" 后面通常是动词

问题2：未知词处理不准确

```
未知词: "beautifully"

规则猜测:
wd.endswith('ly') → RB（副词）✅ 正确

但是:
未知词: "family"
wd.endswith('ly') → RB ❌ 错误（应该是NN）
```

为什么？ - 后缀规则不是100%准确 - 需要更复杂的机器学习方法

问题3：歧义词选择不一定对

```
"book" 在训练数据:
NN: 50次
VB: 10次

朴素标注器总是选NN

但在某些句子中，应该是VB:
"Please book a table"
```

为什么? - 只选最常见的，忽略语境 - 对少见用法错误率高

九、改进方向

1. 隐马尔可夫模型（HMM）

考虑前一个词的标签:

```
P(VB | will) > P(NN | will)
→ "will" 后面更可能是动词
```

2. 条件随机场（CRF）

考虑整个句子的上下文

3. 神经网络

使用深度学习（LSTM, BERT等）

十、纸笔考试重点

必须掌握

1. ✅ 两层字典结构

2. 格式：`{word: {tag: count}}`
3. 为什么需要两层
4. 如何更新计数
5. ✅ **朴素标注算法**
6. 原理：最常见标签
7. 如何找最常见标签
8. 代码实现
9. ✅ **未知词处理**
10. 为什么有未知词
11. 常见规则（-ing, -ly等）
12. 规则优先级
13. ✅ **准确率计算**
14. 公式：正确数/总数
15. 手工计算示例

可能的考题

题型1：构建两层字典

题目：给定数据，构建`word_tag_counts`

```
run/VB fast/RB ./.
run/NN is/VBZ fun/JJ ./.
```

▶ **答案**

题型2：未知词标注

题目：用规则标注这些未知词

1. `Beijing`
2. `quickly`
3. `swimming`
4. `123`

▶ 答案

题型3：计算准确率

题目：

训练数据：

```
love: {'VBP': 10, 'NN': 2}
book: {'NN': 5, 'VB': 1}
```

测试数据：

```
I/PRP love/VBP books/NNS ./.
```

计算准确率（未知词都标为NN）

▶ 答案

十一、记忆口诀

两层字典

> 外层存单词，
> 内层存标签，
> 值是出现次数，
> 字典套字典。

朴素标注

> 统计词标签，
> 选择最常见，
> 未知用规则，
> 简单但有效。

未知词规则

> 大写是专名，
> -ing是动名，
> -ly是副词，
> -ed是过去。

十二、常见错误

❌ 错误1：字典层级搞反

```
# 错误
word_tag_counts = {
    'NN': {'book': 50, 'dog': 30}  ❌
}

# 正确
word_tag_counts = {
    'book': {'NN': 50, 'VB': 10}  ✅
}
```

❌ 错误2：忘记检查键是否存在

```
# 错误
word_tag_counts[wd][tag] += 1  ❌
# 如果wd或tag不存在，会报错KeyError

# 正确
if wd not in word_tag_counts:
    word_tag_counts[wd] = {}
if tag in word_tag_counts[wd]:
    word_tag_counts[wd][tag] += 1
else:
    word_tag_counts[wd][tag] = 1  ✅
```

❌ 错误3：分割字符串错误

```
line = "The/DT dog/NN"

# 错误
parts = line.split('/')
# ['The', 'DT dog', 'NN']  不对!

# 正确
words = line.split()  # 先按空格分
for wdtag in words:
    parts = wdtag.split('/')  # 再按/分
```

祝你考试顺利！ 🎉

记住： - 两层字典是核心数据结构 - 朴素标注=选最常见标签 - 未知词用规则猜测 - 理解原理比记代码重要!