基于CYK+PCFG的短语结构句法分析

实验内容

1. 概率上下文无关文法

PGFG在GFG的基础上引入了P，加上了每个规则的概率。

PCFG中定义一棵句法树的概率为所有用到的规则概率的乘积，一般来说，概率值大的更可能是正确的句法树。

S → NP VP	0.9		
S → VP	0.1	N → *people*	0.5
VP → V NP	0.5	N → *fish*	0.2
VP → V	0.1	N → *tanks*	0.2
VP → V @VP_V	0.3	N → *rods*	0.1
VP → V PP	0.1	V → *people*	0.1
@VP_V → NP PP	1.0	V → *fish*	0.6
NP → NP NP	0.1	V → *tanks*	0.3
NP → NP PP	0.2	P → *with*	1.0
NP → N	0.7		
PP → P NP	1.0		

2. 阅读并理解CYK算法

CYK算法是一个基于“动态规划”算法设计思想，用于测试串w对于一个上下文无关文法L的成员性的一个算法。CYK算法可以在O(n^3)的时间内得出结果。CYK算法是由三个独立发现同样思想本质的人（J. Cocke、 D. Younger和T. Kasami）来命名的。

基于上述文法和CYK算法，编程求句子**fish people fish tanks**的最优分析树。

实验步骤

1. 读入文法并构建4*4矩阵

根据CYK算法，每格Cell[i, j]包含了跨越单词i+1, j+1的所有语法成分（实际计算中下标是从0开始的）。

以Cell[1, 3]为例，Cell[1, 3]格中的成分分别为：(1,1)和(2,3)组成，(1,2)和(3,3)组成，包含了 people fish tanks 所有语法成分。

Fish	people	fish	tanks
score[0][0]	score[0][1]	score[0][2]	score[0][3]
	score[1][1]	score[1][2]	score[1][3]
		score[2][2]	score[2][3]
			score[3][3]

https://blog.csdn.net/Chase1995

```
grams = read_grams()
sentence = "fish people fish tanks"
sentence = sentence.split(" ")
length = len(sentence)
dp = [[[] for _ in range(length)] for _ in range(length)]
```

2. 处理叶子节点

(1) 找出所有直接推导

(2) 根据叶子节点中的单词的词性线性递归地找一元匹配

以[0][0]中NP → N 0.14为例，0.14 = 0.7(规则集中NP→N) * 0.2([0][0]中的N→fish)。

```
def get_closure(gs):
    '''线性递归地找一元匹配闭包'''
    if len(gs) == 0:
        return []
    for g in gs:
        before = get_before(g)
        gs += before  # 动态循环
    '''选取每个表达式的最大概率（去重）'''
    gs.sort(key=lambda x: -x[2])
    res = []
    for g in gs:
        flag = True
        for r in res:
            if r[0] == g[0]:
                flag = False
        if flag:
            res.append(g)
    return res

def get_leaf():
    for dp_idx in range(length):
        for gram in grams:
            if len(gram[1]) == 1 and gram[1][0] == sentence[dp_idx]:
                dp[dp_idx][dp_idx].append(list(gram) + [(-2, -2), (-2, -2)])  #
（-2，-2）代表树叶词本身
    for idx in range(length):
        dp[idx][idx] = get_closure(dp[idx][idx])

'''处理dp[i][i]叶子结点'''
get_leaf()
```

Fish	people	fish	tanks
score[0][0] N → fish 0.2 V → fish 0.6 NP → N 0.14 VP → V 0.06 S → VP 0.006	**score[0][1]**	**score[0][2]**	**score[0][3]**
	score[1][1] N → people 0.5 V → people 0.1 NP → N 0.35 VP → V 0.01 S → VP 0.001	**score[1][2]**	**score[1][3]**
		score[2][2] N → fish 0.2 V → fish 0.6 NP → N 0.14 VP → V 0.06 S → VP 0.006	**score[2][3]**
			score[3][3] N → tanks 0.2 V → tanks 0.3 NP → N 0.14 VP → V 0.03 S → VP 0.003

3.处理非叶子节点（枝干节点）

根据PCYK算法π(i, j, X) =max (q(X→YZ) × π(i, k, Y) × π(k+1, j, Z))。

例如$score[0][1] = score[0][0] \times score[0+1][1]$，我们可以从规则集中找所有能够满足[0][0]和[1][1]的规则(NP → NP NP/ VP → V NP/ S → NP VP)，并再递归地找满足[0][1]的规则(S → VP)。

因为此时S→有两条规则，我们比较其大小，仅保留其对大概率的一条规则即可。

概率计算方法以[0][1]中的S → NP VP 0.00126为例，0.0126 = 0.9(规则集中的S → NP VP)* 0.14([0][0]中的NP → N 0.14) * 0.01([1][1]中的NP → N 0.14)。

[1][2]，[2][3]同理。

根据PCYK算法$score[0][2] = q(X \rightarrow YZ) \times max(score[0][0] \times score[0+1][2], score[0][1] \times score[1+1][2])$。

我们知道，无论是[0][0]+[1][2]还是[0][1]+[2][2]都覆盖了前三个单词的路径，因此我们分别从[0][0]和[1][2]，[0][1]和[2][2]找对应的匹配规则。再对结果找到对应[0][2]的一元规则。

当同一个非终端语符有多条规则时，我们仅保留其最大项。

[1][3]同理。

根据PCYK算法我们分别从[0][0]+[1][3]，[0][1]+[2][3]，[0][2]+[3][3]找对应的匹配规则，再对结果找对应[0][3]的一元规则，这样便可覆盖率所有单词。

当同一个非终端语符有多条规则时，我们仅保留其最大项。

```
def get_branch(X, Y):
    if X == [] or Y == []:
        return []
    res = []
    for x in X:
        for y in Y:
            after = x[0] + y[0]
            for gram in grams:
                if len(gram[1]) == 2 and gram[1] == after:
                    res.append([gram[0], gram[1], x[2] * y[2] * gram[2]])
    return res

def get_branches():
    for d in range(1, length):
        for i in range(length - d):
            j = i + d
            '''get dp[i][j]=dp[i][k]+dp[k+1][j]    '''
            for k in range(i, j):
                new_nodes = get_branch(dp[i][k], dp[k + 1][j])
                dp[i][j] += [new_node + [(i, k), (k + 1, j)] for new_node in
new_nodes]
            '''closure'''
            dp[i][j] = get_closure(dp[i][j])

'''处理非叶子节点(枝干)'''
get_branches()
```

Fish	people	fish	tanks
score[0][0] N → fish 0.2 V → fish 0.6 NP → N 0.14 VP → V 0.06 S → VP 0.006	**score[0][1]** NP → NP NP 0.0049 VP → V NP 0.105 ~~S → NP VP 0.00126~~ S → VP 0.0105	**score[0][2]** NP→NP NP 6.86e-5 ~~NP→NP NP 6.86e-5~~ VP → V NP 0.00147 S → NP VP 8.82e-4 ~~S → NP VP 3.087e-5~~ ~~S → VP 1.47e-4~~	**score[0][3]** NP→NP NP 9.604e-7 ~~NP→NP NP 9.604e-7~~ ~~NP→NP NP 9.604e-7~~ VP→V NP 2.058e-5 ~~S→NP VP 1.2348e-5~~ S→NP VP 1.8522e-4 ~~S→NP VP 1.8522e-6~~ ~~S → VP 2.058e-6~~
	score[1][1] N → people 0.5 V → people 0.1 NP → N 0.35 VP → V 0.01 S → VP 0.001	**score[1][2]** NP → NP NP 0.0049 VP → V NP 0.007 S → NP VP 0.0189 ~~S → VP 0.0007~~	**score[1][3]** NP→NP NP 6.86e-5 ~~NP→NP NP 6.86e-5~~ VP → V NP 9.8e-5 S → NP VP 0.01323 ~~S → NP VP 1.323e-4~~ ~~S → VP 9.8e-6~~
		score[2][2] N → fish 0.2 V → fish 0.6 NP → N 0.14 VP → V 0.06 S → VP 0.006	**score[2][3]** NP → NP NP 0.00196 VP → V NP 0.042 ~~S → NP VP 0.00378~~ S → VP 0.0042
			score[3][3] N → tanks 0.2 V → tanks 0.3 NP → N 0.14 VP → V 0.03 S → VP 0.003

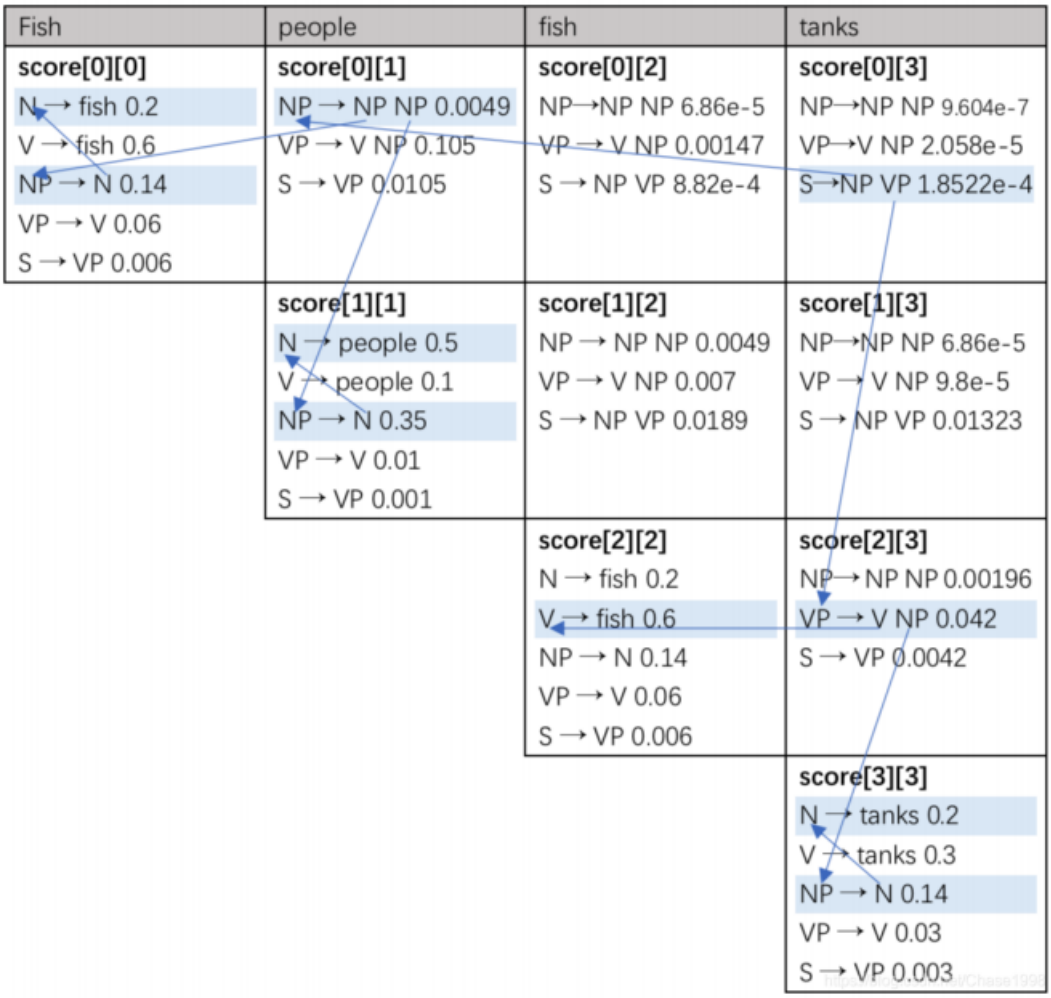
4. 回溯找出结果

编写**get_tree(begin_idx, end_idx, s)**函数，三个参数分别代表开始下标，结束下标和寻找的标记符，返回地结果是指定格式的语法树

从根节点的开始标志S出发，按照之前保留的路径找出概率最大句法树。

```
def get_tree(begin_idx, end_idx, s):
    max_p = 0
    max_gram = []
    res1 = "["
    res2 = "]"
    if s == " ":
        for gram in dp[begin_idx][end_idx]:
            if max_p < gram[2]:
                max_p = gram[2]
                max_gram = gram
    else:
        for gram in dp[begin_idx][end_idx]:
            if gram[0][0] == s:
                max_gram = gram
    res1 += max_gram[0][0] + "["
    res2 = "]" + res2

    while max_gram[3] == (-1, -1):
```

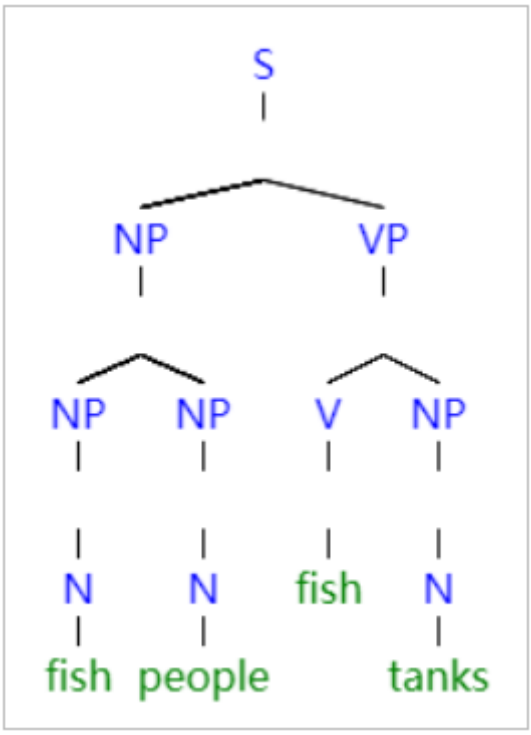
```
        for gram in dp[begin_idx][end_idx]:
            if gram[0][0] == max_gram[1][0]:
                max_gram = gram
                res1 = res1 + "[" + max_gram[0][0]
                res2 = "]" + res2
                break
    if max_gram[3] == (-2, -2):
        return res1 + "[" + max_gram[1][0] + "]" + res2
    return res1 + get_tree(max_gram[3][0], max_gram[3][1], max_gram[1][0]) +
get_tree(max_gram[4][0], max_gram[4][1],max_gram[1][1]) + res2

print(get_tree(0, len(sentence) - 1, " "))
```



实验结果

语法树：

```
[S[[NP[[NP[[N[fish]]]][NP[[N[people]]]]]][VP[[V[[fish]]][NP[[N[tanks]]]]]]]
```


S
NP
VP
NP
NP
V
NP
N
N
fish
N
fish
people
tanks