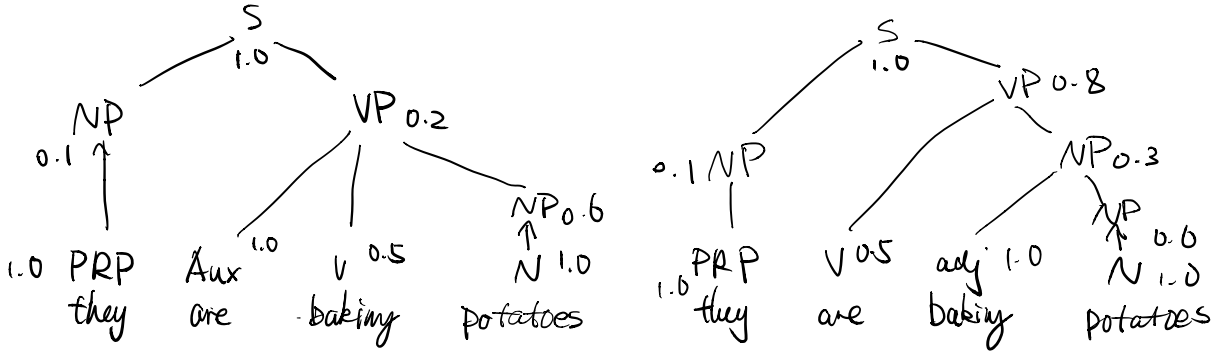
HW2 COMS W4705

Name: Ruoyu Li Uni: rl3161

Problem 1.

a) $P(tags, words) = \left(\prod_{i=1}^{n} P(t_i|t_{i-1}) P(w_i|t_i)\right) * P(END|t_n)$



~~PRP Aux V N~~

~~PRP Aux adj N~~

~~PRP V V N~~

PRP V adj N

P(PRP, Aux, V, N, they, are, baking, potatoes)

$= 1.0 \times 0.1 \times 1.0 \times 0.2 \times 1.0 \times 0.5 \times 1.0 \times 0.6 =$

$= 0.02 \times 0.3 = 0.006$

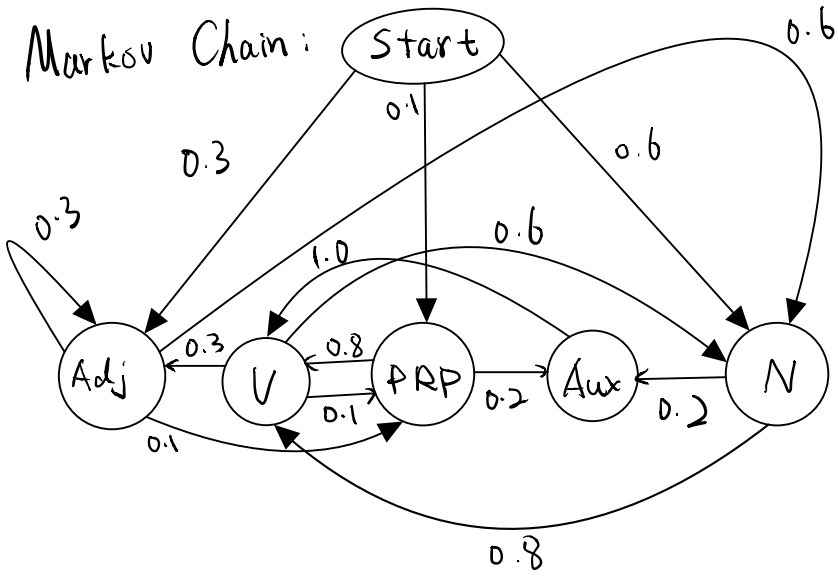
P(PRP, Aux, V, N, they, are, baking, potatoes)

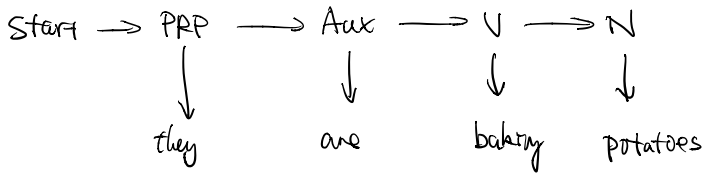
$= 1.0 \times 0.1 \times 1.0 \times 0.8 \times 0.5 \times 0.3 \times 1.0 \times 0.6 \times 1.0$

$= 0.04 \times 0.18$

$= 0.0072$

b) Markov Chain:

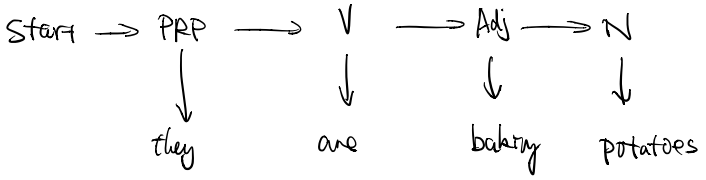




P(PRP | start) = 1.0 × 0.1 P(they | PRP) = 1.0
P(Aux | PRP) = 0.2 P(are | Aux) = 1.0
P(V | Aux) = 1.0 P(baking | V) = 0.5
P(N | V) = 0.6 P(potatoes | N) = 1.

1.0 × 0.1 × 1.0 × 1.0 × 0.2 × 0.5 × 0.6 × 1.0

= 0.02 × 0.3 = 0.006



P(PRP | start) = 1.0 × 0.1
P(V | PRP) = 0.8
P(adj | V) = 0.3
P(N | adj) = 0.6
1.0 × 0.1 × 1.0 × 0.8 × 1.0 × 0.3 × 0.3 × 0.6 × 1.0

= 0.08 × 0.09
= 0.0072

C) It is not possible. PCFG can produce recursion when generating tags. But, HMMs can not do recursion. So when PCFG generate a infite many tags, HMMs can not reproduce the process.

P2. a.

chart [0]
S → • NP VP [0,0] pred
NP → • Adj NP [0,0] pred
NP → • PRP [0,0] pred
NP → • N [0,0] pred
Adj → • baking [0,0] noth
PRP → • they [0,0] scan
N → • potatoes [0,0] noth

chart [1]
PRP → they • [0,1] comp
NP → PRP • [0,1] comp
S → NP • VP [1,1] pred
VP → • V NP [1,1] pred
VP → • Aux V NP [1,1] pred
V → • baking [1,1] noth
V → • are [1,1] scan
Aux → • are [1,1] scan

chart [2]
V → are • [1,2] comp
Aux → are • [1,2] comp
VP → V • NP [2,2] pred
VP → Aux • V NP [2,2] pred
NP → • Adj NP [2,2] pred
NP → • PRP [2,2] pred
NP → • N [2,2] pred
V → • baking [2,2] scan
V → • are [2,2] noth
Adj → • baking [2,2] scan
PRP → • they [2,2] noth
N → • potatoes [2,2] noth

chart [3]

V → baking • [2,3] comp

Adj → baking • [2,3] comp

VP → Aux V • NP [3,3] pred

NP → Adj • NP [3,3] pred

NP → • Adj NP [3,3] pred

NP → • PRP [3,3] pred

NP → • N [3,3] pred

Adj → • baking [3,3] auth

PRP → • they [3,3] auth

N → • potatoes [3,3] scan

chart [4]

N → potatoes • [3,4] comp

NP → N • [3,4]

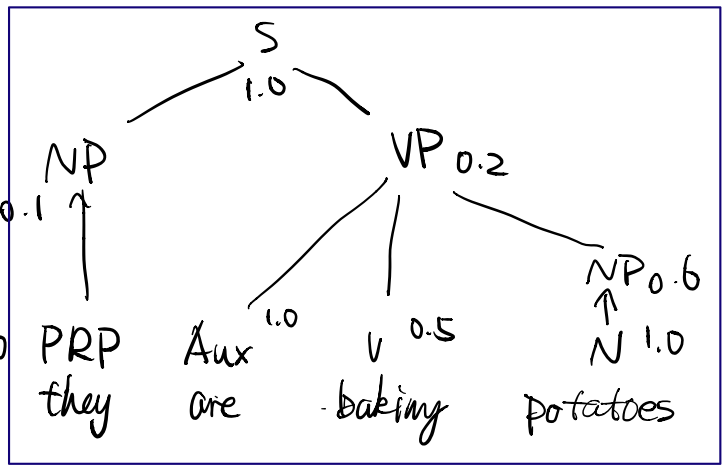
VP → Aux V NP • [1,4]

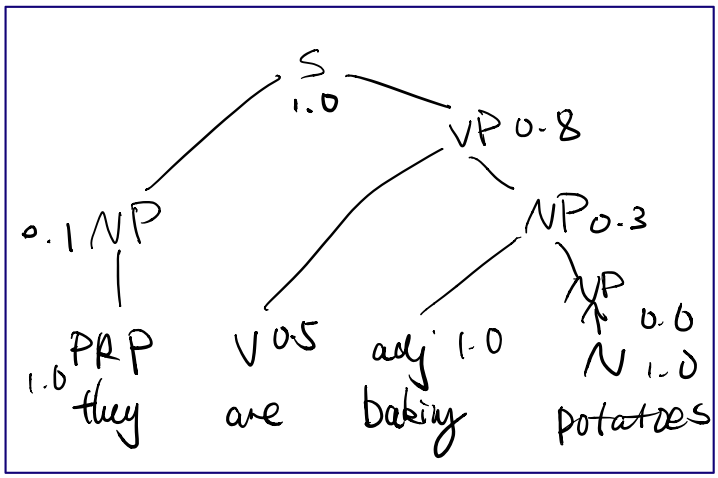
NP → Adj NP • [2,4]

VP → V NP • [1,4]

S → NP VP • [0,4]

b)





P(PRP, Aux, V, N, they, are, baking, potatoes)

= 1.0 x 0.1 x 1.0 x 0.2 x 1.0 x 0.5 x 1.0 x 0.6 =

= 0.02 x 0.3 = 0.006

P(PRP, Aux, V, N, they, are, baking, potatoes)

= 1.0 x 0.1 x 1.0 x 0.8 x 0.5 x 0.3 x 1.0 x 0.6 x 1.0

= 0.04 x 0.18

= 0.0072

P3. S → NP VP

NP → Adj NP

VP → V NP

b. 1) A → B. When B → b, then A → b.

2) A → B C D E. Creat X → BC and Y → DE. A → X Y.

VP → Aux B

B → V NP

Adj → baking

V → baking

V → are

Aux → are

NP → they

NP → potatoes

b)

		they	are	baking	potatoes 4
0		NP			S
1			Aux, V		VP, B
2				V, Adj	NP, VP, B
3					NP
4					

$NP_{[2,4]} \rightarrow Adj_{[2,3]}\ NP_{[3,4]}$

$VP_{[2,4]} \rightarrow V_{[2,3]}\ NP_{[3,4]}$

$B_{[2,4]} \rightarrow V_{[2,3]}\ NP_{[3,4]}$

$VP_{[1,4]} \rightarrow V_{[1,2]}\ NP_{[2,4]}$

$VP_{[1,4]} \rightarrow Aux_{[1,2]}\ VP_{[2,4]}$

$B_{[1,4]} \rightarrow V_{[1,2]}\ NP_{[2,4]}$

$S_{[0,4]} \rightarrow NP_{[0,1]}\ VP_{[1,4]}$

P4. I only write A down, when A gets changed

$(\sigma \mid root, he \mid \beta, \{\}_A) \xrightarrow{shift} (\sigma \mid he, sent \mid \beta, \{\}_A)$

$\xrightarrow{L\text{-}Arc\ nsubj} (\sigma \mid root, sent \mid \beta, \{(sent, nsubj, he)\}) \xrightarrow{shift} (\sigma \mid sent, her \mid \beta, A)$

$\xrightarrow{R\text{-}Arc\ jobj} (\sigma \mid root, sent \mid \beta, \{(sent, nsubj, he), (sent, jobj, her)\}) \xrightarrow{shift}$

$(\sigma \mid sent, a \mid \beta, A) \xrightarrow{shift} (\sigma \mid a, funny \mid \beta, A) \xrightarrow{shift}$

$(\sigma \mid funny, meme \mid \beta, A) \xrightarrow{L\text{-}Arc\ amod}$

$(\sigma, meme \mid \beta, \{(sent, nsubj, he), (sent, jobj, her), (meme, amod, funny)\}) \xrightarrow{L\text{-}Arc\ det}$

$(\sigma \mid sent, meme \mid \beta, \{(sent, nsubj, he), (sent, jobj, her), (meme, amod, funny), (meme, det, a)\}) \xrightarrow{R\text{-}Arc\ dobj}$

$(\sigma \mid root, sent \mid \beta, \{(sent, nsubj, he), (sent, jobj, her), (meme, amod, funny), (meme, det, a), (sent, dobj, meme)\}) \xrightarrow{shift} (\sigma \mid sent, today \mid \beta, A) \xrightarrow{R\text{-}Arc\ advmod}$

(σ|root, sent|β, {(sent, nsubj, he), (sent, jobj, her), (meme, amod, funny), (meme, det, a), (sent, dobj, meme), (sent, advmod, today)}) R-Arc pred

(α, root|β, {(sent, nsubj, he), (sent, jobj, her), (meme, amod, funny), (meme, det, a), (sent, dobj, meme), (sent, advmod, today), (root, pred, sent)})

shift → (root, [], A).

So the final A is

{(sent, nsubj, he), (sent, jobj, her), (meme, amod, funny), (meme, det, a), (sent, dobj, meme), (sent, advmod, today), (root, pred, sent)}

And transitions are:

(shift, left-Arc$_{nsubj}$, shift, Right-Arc$_{jobj}$, shift, shift, shift, left-Arc$_{amod}$, left-Arc$_{det}$, Right-Arc$_{dobj}$, shift, Right-arc$_{advmod}$, right-Arc$_{pred}$)

COMS W4705 – Fall B 2020 - Natural Language Processing - Homework 2

Name: Ruoyu Li
UNI: rl3161

Programming Component

Part 1

```
def verify_grammar(self):
    """
    Return True if the grammar is a valid PCFG in CNF.
    Otherwise return False.
    """
    # TODO, Part 1
    for lhs_key in self.lhs_to_rules.keys():
        rules = self.lhs_to_rules[lhs_key]
        lhs_probs = []
        for rule in rules:
            lhs, rhs, prob = rule
            lhs_probs.append(prob)
            if len(rhs) not in (1, 2):
                print('Error Message: ', rhs, 'is not in a format of "A -> BC" or "A -> b"')
                return False
            elif len(rhs) == 1:
                for c in rhs[0]:
                    if c.isupper():
                        print('Error Message: ', rhs, 'should all be lower case.')
                        return False
            elif len(rhs) == 2:
                for c in rhs[0]:
                    if c.islower():
                        print('Error Message: ', rhs, 'should all be UPPER CASE.')
                        return False
                for c in rhs[1]:
                    if c.islower():
                        print('Error Message: ', rhs, 'should all be UPPER CASE.')
                        return False
        if fsum(lhs_probs) < 0.999 or fsum(lhs_probs) > 1.001:
```

```
            print('Error Message: ', lhs, '\'s probability does not sum to 1.0')
            return False

    print("This is a valid PCFG in CNF.")
    return True
```

Part 2

```
def is_in_language(self,tokens):
    """
    Membership checking. Parse the input tokens and return True if
    the sentence is in the language described by the grammar. Otherwise
    return False
    """
    # TODO, part 2
    table = defaultdict(tuple)
    N = len(tokens)

    for i in range(0, N):
        if (tokens[i],) not in self.grammar.rhs_to_rules:
            print('Error Message: ', tokens[i], 'is not in terminal words.')
            return False
        rules = self.grammar.rhs_to_rules[(tokens[i],)]
        for rule in rules:
            table[(i, i+1)] += (rule[0],)

    for length in range(2, N+1):
        for i in range(0, N-length+1):
            j = i + length
            for k in range(i+1, j):
                for B in table[(i, k)]:
                    for C in table[(k,j)]:
                        if (B, C) in self.grammar.rhs_to_rules.keys():
                            rules = self.grammar.rhs_to_rules[(B, C)]
                            for rule in rules:
                                table[(i, j)] += (rule[0],)

    if self.grammar.startsymbol in table[(0, N)]:
```

```
        return True
    return False
```

Part 3

```
def parse_with_backpointers(self, tokens):
    """
    Parse the input tokens and return a parse table and a probability table.
    """
    # TODO, part 3
    table = defaultdict(defaultdict)
    probs = defaultdict(defaultdict)
    N = len(tokens)

    for i in range(N):
        if (tokens[i],) not in self.grammar.rhs_to_rules.keys():
            print('Error Message: ', tokens[i], 'is not in terminal words.')
            return table, probs
        rules = self.grammar.rhs_to_rules[(tokens[i],)]
        for rule in rules:
            table[(i, i+1)][rule[0]] = rule[1][0]
            probs[(i, i+1)][rule[0]] = math.log2(rule[2])

    for length in range(2, N + 1):
        for i in range(0, N - length + 1):
            j = i + length
            for k in range(i + 1, j):
                for B in table[(i, k)]:
                    for C in table[(k, j)]:
                        if (B, C) in self.grammar.rhs_to_rules.keys():
                            rules = self.grammar.rhs_to_rules[(B, C)]
                            for rule in rules:
                                prob = math.log2(rule[2]) + probs[(i, k)][B] + probs[(k, j)][C]
                                if rule[0] not in probs[(i, j)].keys() or prob > probs[(i, j)][rule[0]]:
                                    table[(i, j)][rule[0]] = ((B, i, k), (C, k, j))
                                    probs[(i, j)][rule[0]] = prob
```

```
    #print(table[(0, N)])
    return table, probs
```

Part 4

```
def get_tree(chart, i,j,nt):
    """
    Return the parse-tree rooted in non-terminal nt and covering span i,j.
    """
    # TODO: Part 4
    if isinstance(chart[(i, j)][nt], str):
        return (nt, chart[(i, j)][nt])
    else:
        left = chart[(i, j)][nt][0]
        right = chart[(i, j)][nt][1]
        return (nt, get_tree(chart, left[1], left[2], left[0]), get_tree(chart, right[1], right[2], right[0]))
```