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]))
```