Advanced CL- Final Project

(a)

Implementation Description:

Near-CNF implementation:

In the main function- "to_near_cnf", I followed the steps of the algorithm described in class, and called other functions:

Firstly I created a new instance of PCFG and added the rules of the grammar and a new start variable.

For the second step- eliminating epsilon rules, I created a list of every variable that its epsilon rule was eliminated, called black, and called the function "eliminate_epsilon_rules" until there were no epsilon rules (except S_0 → "").

- In "eliminate_epsilon_rules" I followed the algorithm- removed the current epsilon rule A and normalized the remaining rules for A (using "normalize").
- I found all rules where A appears in the right-hand side (using "right_hand_side"). For each of them I found the occurrences of A in the derivation (using "indices_of_vari") and found all rules that can be obtained by applying the removed epsilon rule partially or fully (using "remove_occurrences"). I updated the original rule's probability (using "new_prob_computation"). For each rule obtained by removing the epsilon rule, I checked if it existed in the grammar (using "exists") and added or updated the rule accordingly.

For the third step- converting long rules:

For rules with multiple symbols:

I created a list of all rules with RHS > 2 and for each rule I broke the RHS of it into a sequence of rules whose RHS ≤ 2 symbols using the function "multiple_symbols".

- In "multiple_symbols", for each rule, I generated new rules of the required form (RHS ≤ 2). The last rule created in each function call is composed of the last two variables in the derivation of the original rule.

For converting rules with terminals:
I created 3 lists:
a list of all rules of the form $A \rightarrow Bc[p]$ where $B \in V$ and $c \in \Sigma$
a list of all rules of the form $A \rightarrow bC[p]$ where $b \in \Sigma$ and $C \in V$
a list of all rules of the form $A \rightarrow bc[p]$ where $b,c \in \Sigma$

For each rule derivation I checked if it matches to one of the 3 lists using the function "is_variable" that decides if an element u in a derivation is a terminal (if it is not a key in the rules dictionary).
For each rule in each list, I converted it to the required form:
If A → Bc [p]:

A → BY [p] (changed)

Y → c [1] (new)

If A→ bC [p]:

A → c Y[p] (changed)

Y → B [1] (new)

If A→ bc [p], B, C ∈ V and b, c ∈ Σ:

A → XY [p] (changed)

X → b [1] (new)

Y → c [1] (new)

(using the function "one_terminal").

cky_parser Implementation:

I created a table and a backpointers table (using "table"), the tables are (n+1)X(n+1) as I found it more convenient to work with. In each cell in the table t I kept variables and their probabilities, in each cell in the backpointers table bp I kept variables with their derivation and probability. For each variable in the derivation, I kept its probability too. It was convenient for debugging.

I filled the tables inductively:

For each cell (i,j) , $j = i+1$, I found the unit rules (because in that cell appears a word which is a single terminal) that match to the current word using "vars_to_terminal" and added them to the tables. In "vars_to_terminal" I found the derivations in which the current word appears.

For each cell (i, j), $j > i+1$, I looked at all possible ways to divide the partial string w[i,j] into 2 non empty strings w[i,k] and w[k,j] and for each partition I found its corresponding binary rules using the function "binary_rules".

- In "binary_rules", for each possible partition I found the variables in the required cells and called the function "vars_to_vars" to check whether binary rules can be composed from the variables found.
- In "vars_to_vars", for each variables combination that composes a binary rule Z→XY [q] in the grammar, I checked if the product of the probability of X in its cell (i,k), with the probability of Y in its cell (k,j) with the probability q is larger than the probability of Z in its cell (i,j) (0 if Z is not in the cell). If so, I added it to a list of the variables to add to the tables/ update in the tables in "binary_rules". In order to check if a probability update was needed, I called the functions "check_if_rule_was_added_list", "check_if_rule_was_added_dic" from "binary_rules".

After adding binary rules, I called the function "unary_rules" to add unary rules that can be derived from the variables in the current cell (i,j).

- In "unary_rules" I iterated over the variables in the cell until no new possible unary rules in the cell are found. At each iteration I added new variables to the cell.

After both of the tables were filled, I created a new backpointers table without the probabilities, only with a variable and its derivation (using "bp_no_probs").

I checked if the given grammar can derive the given string. If so, I sent it to the function "my_tree" with the new backpointers table.

The function "my_tree" builds the tree recursively- for each sub-tree it finds its leftmost child and adds the child's right brother if exists.

Halting condition was getting to the bottom row or to the leftmost column.

is_valid_grammar implementation:

For each variable in the grammar, I summed the probabilities of the rules that it derives. If one of the sums was more then $1 + |\varepsilon|$ or less then $1 - |\varepsilon|$, $|\varepsilon| = 2^{-50}$, I declared the grammar to be not valid. If every sum was in the interval $(1 - |\varepsilon|, 1 + |\varepsilon|)$, $|\varepsilon| = 2^{-50}$ I declared the grammar to be valid.

main Implementation:

The function reads from the files, generates the grammar in "generate_grammar", creates a new instance of the grammar converted to near cnf (calls the function "to_near_cnf") and for each sentence calls "cky_parser". (I removed " " from terminals)

I used other functions that were not mentioned here. The code is documented and relevant comments appear in the functions.

(b)

Near-CNF Probabilities:

The probability updates I decided on:

- For the new start variable S_0 and the rule S_0 → S[p] I chose $p = 1$ because S is S_0's unique derivation at the first step of adding a new start variable. $\sum_{\alpha} P(\alpha|S_0) = P(S|S_0) = 1$ so, the probability of deriving S_0 in the new grammar will be the same because we multiply the probability of deriving S from a sequence of rules by 1.

```
def to_near_cnf(self):
    """
    Returns an equivalent near-CNF grammar.
    """
    near_cnf = PCFG(self.start, self.rules)  # generate a grammar with self.rules
    new_start = PRule("S_0", (self.start,), 1)  # 1. create new start variable
    near_cnf.add_rule(new_start)  # 1. add new start variable
    near_cnf.start = new_start.variable  # set as start variable
```

p =1

- For eliminating epsilon rules:
  - After removing an epsilon rule A → ""[p], I normalized the probabilities of its other rules (notated as q) so the sums of the probabilities will be 1 again. I used the formula that we saw in class: "each rule A → u[q] (where $u \in \Sigma \cup V$) $*$) will become: A → u[$\frac{q}{1-p}$]".

$$Before\ removing\ A \to \epsilon[p]: \sum P(\alpha|A) = p + \sum q_i = 1$$

$$After\ removing, before\ normalization: 1 - p = \sum q_i$$
$$After\ normalization: 1 = \sum \frac{q_i}{1-p}$$

```
def eliminate_epsilon_rules(self, rule, black):
    """
    eliminate all rules of the form A → ε[p] (except A = S0)
    from the grammar
    :param rule: a rule of the form A → ε[p]
    :return: None
    """
    my_variable = rule.variable
    p = rule.probability
    self.remove_rule(rule)  # a. remove A --> epsilon[p]
    black.append(rule.variable)
    self.normalize(rule)  # b. normalize all remaining rules for A
```

```
def normalize(self, curr_rule):
    """
    normalize all remaining rules for curr_rule
     each rule A → u[q] (where u ∈ (Σ ∪ V )*) will become A → u[q/1-p]
    """
    my_variable = curr_rule.variable
    p = curr_rule.probability
    for rule in self.rules[my_variable]:    # each rule A → u[q] (where u ∈ (Σ ∪ V )*)
        q = rule.probability
        rule.probability = self.norm_computation(q, p)  # will become A → u[q/1-p]
```

- For every rule β with variable B that A appears in its RHS and has never had the form B → "" before, I added to the grammar all the combinations described in class:
  B → $u_1u_2Au_3A...Au_n$

  .....

  B → $u_1...u_n$

In order to maintain the sum $\sum_\alpha P(\alpha|B) = 1$, for each rule that was not part of the rules before, I updated its probability to be: $q \cdot p^k \cdot (1-p)^{m-k}$ where p is the probability of A → "", q is the probability of the rule B that S appears on its RHS, k is the number of occurrences of A that were removed, m is the number of occurrences of A in the RHS of original rule.
For each rule that was already part of the rules and was not the original rule, I added $q \cdot p^k \cdot (1-p)^{m-k}$ to its probability.
The probability of the original rule became $q \cdot (1-p)^m$, $(k = 0)$.

$$Before\ adding\ and\ changing\ P(\beta|B) = q$$
$$\sum_\alpha P(\alpha|B) = 1 = \sum_{\alpha \neq \beta} P(\alpha|B) + P(\beta|B)$$

$$We\ would\ like\ to\ maintan\ the\ sum\ 1 - q = \sum_{\alpha \neq \beta} P(\alpha|B)$$

so the probability of β and the other rules added and the addition for every rule changed $= q$

The sum of rules added and changed is $\binom{m}{k}$, for each rule, we add $q \cdot p^k \cdot (1-p)^{m-k}$:

$$\sum_{k=0}^{m} \binom{m}{k} q \cdot p^k \cdot (1-p)^{m-k} =_{binom} q(1-p+p)^m = q$$

as required. (*) Every rule added or updated reflects the process of deriving a string in the former grammar using epsilon rules. (applying β (q), applying k times epsilon rule (p^k), not applying m-k times epsilon rules $(1-p)^{m-k}$).

In the process of eliminating epsilon rules, we find every combination of each rule derivation that can be obtained when applying an epsilon rule on one of the derivation variables. We add each combination to our grammar. In that way, after the process of eliminating epsilon rules, every possible rule derivation that can be obtained by applying a sequence of epsilon rules in the former grammar, is included in the current grammar as a rule. The probability of each new or updated rule is calculated per variable that had an epsilon rule.

At the beginning, the current grammar assigns the same total probability to each string as the former grammar (including the new start variable, explained above).

For the first epsilon rule that is eliminated, the probability calculations can be seen and they above create a valid grammar that assigns the same total probability to each string as the former grammar (explained in (*)).

Suppose we eliminated k-1 epsilon rules and the current grammar assigns the same total probability to each string as its former grammar (at each of the k-1 eliminations, hence as the original grammar as well).

For the k'th epsilon rule with variable V that is eliminated, we apply the process described above. After the process, the current grammar assigns the same total probability to each string as the former grammar because it reflects the process of deriving a string in the former grammar using epsilon rules as described above in (*). As supposed, the former grammar assigns the same total probability to each string as the original grammar, so the current grammar does so as well.

Hence, for each variable that its epsilon rule has been removed, the current grammar assigns the same total probability to each string as the former grammar.

```
m = len(indices)
for k in range(1,
               m + 1):  # all rules that can be obtained by applying the removed rule partially or fully
    new_rules += self.remove_occurrences(k, indices, r, p)
r.probability = self.new_prob_computation(r.probability, p, 0, m)  # update rule probability
for new_rule in new_rules:
    if self.exists(new_rule):  # If one of these rules already exists in the grammar we update its
        # probability

        existing = self.find_unknown_rule(new_rule)
        existing.probability = existing.probability + new_rule.probability
```

Probability addition to an existing rule

```
@staticmethod
def new_prob_computation(q, p, k, m):
    """
    compute: q * p_k * p_l
    :param q: probability of a rule
    :param p: probability of another rule
    :param k: k<m, number of removed occurrences of a variable
    :param m: total number of occurrences of a variable in a derivation
    :return: q * p_k * p_l
    """
    p_k = pow(p, k)
    p_l = pow(1 - p, m - k)
    return q * p_k * p_l
```

-

- For converting long rules:
  - For every rule of the form $A \rightarrow v_1 \ldots v_n[p]$ where $n > 2$ ($\forall i. v_i \in \Sigma \cup V$) I added n-2 new rules that have only 1 possible derivation, so the probability of each such rule is 1.
    The probability of the rule $A \rightarrow v_1 N_1$ stays the same probability of $A \rightarrow v_1 \ldots v_n[p]$. The probability stays the same because:
    Original rule : $A \rightarrow v_1 \ldots v_n[p]$
    New/ changed rules:
    $A \rightarrow v_1 N_1[p]$
    $N_1 \rightarrow v_2 N_2$ [1]
    $N_2 \rightarrow v_3 N_3$ [1]
    ....
    $N_i \rightarrow v_{i+1} N_{i+1}$ [1]
    ....
    $N_{n-2} \rightarrow v_{n-1} v_n$ [1]
    Getting to the sequence $v_1 \ldots v_n$ in the new sequence of rules will "cost" us $P(v_1 N_1 | A) \cdot \prod_{i=1}^{n-2} (\sum_{\alpha} P(\alpha | N_i)) = p \cdot 1 \cdot 1 \cdot \ldots \cdot 1 \cdot \ldots \cdot 1 = p$ as in the original rule.
  - For every rule of the forms: (1) $A \rightarrow Bc$ [p], (2) $A \rightarrow bC$ [p], (3) $A \rightarrow bc$ [p], $B, C \in V$ and $b, c \in \Sigma$:
    I changed and added the rules:
    (1) $A \rightarrow BY$ [p] (changed)
    $Y \rightarrow c$ [1] (new)
    (2) $A \rightarrow c Y$[p] (changed)

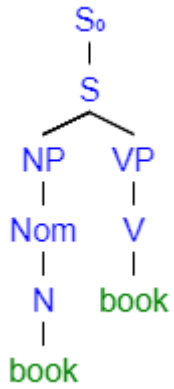
Y → B [1] (new)
(3) A → XY [p] (changed)
X → b [1] (new)
Y → c [1] (new)
The probability of the changed rules stays the same because:
(1) Getting to the sequence Bc will "cost" us : $P(BY|A) \cdot P(c|Y) = p \cdot 1 = p$ as in the original rule
(2) Getting to the sequence cB will "cost" us : $P(cY|A) \cdot P(B|Y) = p \cdot 1 = p$ as in the original rule
(3) Getting to the sequence bc will "cost" us : $P(XY|A) \cdot P(b|X) \cdot P(c|Y) = p \cdot 1 \cdot 1 = p$ as in the original rule

(c)

book book:

probability- 0.000495

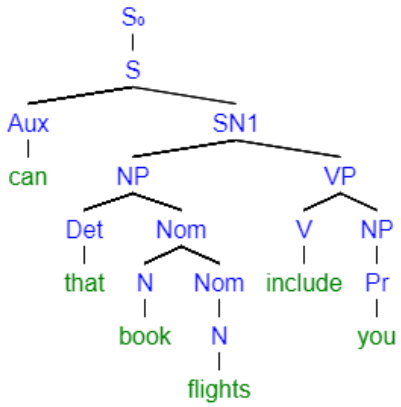
tree- [S_0 [S [NP [Nom [N book]]] [VP [V book]]]]



can that book flights include you:

probability- 8.64e-08

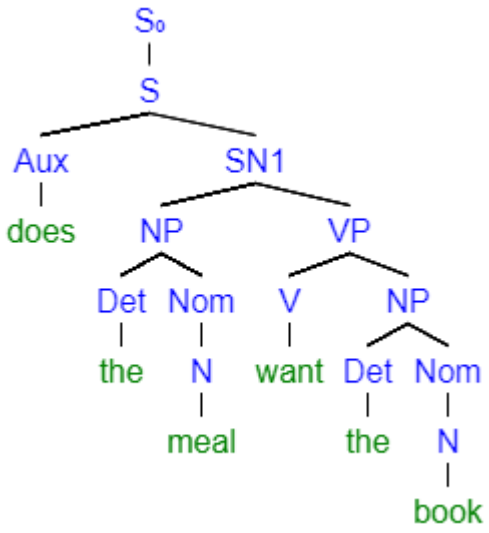
tree - [S_0 [S [Aux can] [S_N1 [NP [Det that] [Nom [N book] [Nom [N flights]]]] [VP [V include] [NP [Pr you]]]]]]



does the meal want the book:

probability- 4.1472e-06

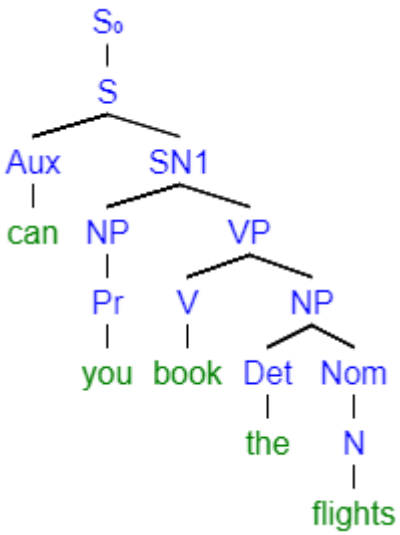
tree- [S_0 [S [Aux does] [S_N1 [NP [Det the] [Nom [N meal]]] [VP [V want] [NP [Det the] [Nom [N book]]]]]]]



<u>can you book the flights</u>:

probability- 6.912e-05

tree - [S_0 [S [Aux can] [S_N1 [NP [Pr you]] [VP [V book] [NP [Det the] [Nom [N flights]]]]]]]



<u>do that Denver flights include a book Denver book a meal</u>:

the given grammar cannot derive the given string (low probability)