Traitement automatique du langage
TP 4 — PCFGs and Parsing

Yves Scherrer

1 Implement a probabilistic CKY parser

```
#! /usr/bin/env python3
# -*- coding: utf-8 -*-

import json, sys, collections


# recursive procedure to convert a nested list into a nested
   parentheses representation
def list2str(l):
    if type(l) == str:
        return l
    else:
        return ”(” + ” ”.join([list2str(e) for e in l]) + ”)”


# auxiliary procedure to display the contents of the CKY matrix
def printTable(pi, n):
    row = [””] + list(range(1, n))
    print(” ”.join([”{:<10}”.format(x) for x in row]))
    for i in range(1, n):
        row = [i]
        for j in range(1, n):
            probs = {X: pi[(I, J, X)] for I, J, X in pi if I==i and
               J==j and pi[(I, J, X)] > 0}
            if len(probs) == 0:
                row.append(” ”)
            else:
                row.append(”,”.join([”{}:{:.3f}”.format(p, probs[p])
                    for p in sorted(probs)]))
        print(” ”.join([”{:<10}”.format(x) for x in row]))
    print()
```

```
# remove <..> nodes
def reconstructTree1(tree):
    for i, element in enumerate(tree):
        if (type(element) == list) and element[0].startswith(”<”)
           and element[0].endswith(”>”):
            del tree[i]
            tree[i:i] = element[1:]
    for element in tree:
        if type(element) == list:
            reconstructTree1(element)


# split + nodes
def reconstructTree2(tree):
    for i, element in enumerate(tree):
        if (type(element) == list) and (”+” in element[0]):
            nodes = element[0].split(”+”)
            if len(nodes) == 2:
                tree[i] = [nodes[0]] + [[nodes[1]] + element[1:]]
            elif len(nodes) == 3:
                tree[i] = [nodes[0]] + [[nodes[1]] + [[nodes[2]] +
                   element[1:]]]
    for element in tree:
        if type(element) == list:
            reconstructTree2(element)


# follow the backpointers and recursively build a nested list
def retrieveTree(bp, a, b, symbol):
    if (a, b, symbol) in bp:
        rule, s = bp[(a, b, symbol)]
        if s == 0:          # terminal rule
            return [rule[0], rule[1]]
        else:               # non-terminal rule
            r = retrieveTree(bp, a, s, rule[1])
            l = retrieveTree(bp, s+1, b, rule[2])
            return [symbol, r, l]
    else:
        return []


# load a grammar from a JSON file
def readGrammar(grammarfile):
    grammar = json.load(open(grammarfile, ’r’))
    d = {}
    for rule in grammar[”NTR”]:                         # convert [”S”,
        "NP", "VP", 1.0] into {"S": {("NP", "VP"): 1.0}}
        if rule[0] not in d:
            d[rule[0]] = {}
        d[rule[0]][(rule[1], rule[2])] = rule[3]
    grammar["NTR"] = d
    d = {}
    for rule in grammar["TR"]:
        if rule[0] not in d:
            d[rule[0]] = {}
        d[rule[0]][rule[1]] = rule[2]
    grammar["TR"] = d
    return grammar


# implementation of the CKY algorithm to parse a sentence using a
   grammar
def parseSentence(grammar, sentence):
    N = set(grammar["NTR"].keys()) | set(grammar["TR"].keys())
    S = grammar["I"]

    sentence = [""] + sentence.split(" ")   # to get 1-based indices
    n = len(sentence)
    pi = collections.defaultdict(float)     # no need to initialize
       cells to 0
    bp = {}

    # initialization
    for i in range(1, n):
        found = False
        for X in N:
            #print(X, i, sentence[i], grammar["TR"])
            if (X in grammar["TR"]) and (sentence[i] in grammar["TR"
               ][X]):
                pi[(i, i, X)] = grammar["TR"][X][sentence[i]]
                bp[(i, i, X)] = ((X, sentence[i]), 0)
                found = True
        if not found:
            print("Word {} not found in grammar!".format(sentence[i
               ]))
            # add-one smoothing if _UNKNOWN_ probabilities are set
               in the grammar
            for X in grammar["TR"]:
                if "_UNKNOWN_" in grammar["TR"][X]:
                    pi[(i, i, X)] = grammar["TR"][X]["_UNKNOWN_"]
                    bp[(i, i, X)] = ((X, sentence[i]), 0)

    # algorithm
    for l in range(1, n):       # l < n, so last l equals to n-1
        for i in range(1, n-l+1):       # i < n-l+1, so last i
           equals to n-l
            j = i + l
            for X in N:
                # start of the max calculation
                maxProb = 0.0
                maxRule = ””
                maxS = ””
                if X in grammar[”NTR”]:
                    for rule in grammar[”NTR”][X]:
                        Y, Z = rule
                        for s in range(i, j):
                            prob = grammar[”NTR”][X][rule] * pi[(i,
                               s, Y)] * pi[(s+1, j, Z)]
                            if prob > maxProb:
                                maxProb = prob
                                maxRule = (X, Y, Z)
                                maxS = s
                if maxProb > 0.0:
                    pi[(i, j, X)] = maxProb
                    bp[(i, j, X)] = (maxRule, maxS)

    #printTable(pi, n)
    tree = retrieveTree(bp, 1, n-1, grammar[”I”])
    reconstructTree1(tree)
    reconstructTree2(tree)
    return (tree, pi[(1, n-1, grammar[”I”])])


def testToyGrammar():
    g = readGrammar(”toygrammar.json”)
    sentences = [”the man the girl”, ”the girl sees the telescope”,
       ”the girl sees the telephone”, ”the telescope watches the man
        with the girl”, ”the man sees the man with the telescope”]
    for s in sentences:
        print(s)
        tree, p = parseSentence(g, s)
        print(list2str(tree))
        print(p)
        print(”--------------”)


def testTreebankGrammar():
    g = readGrammar(”treebankrules.json”)
    testfile = open(”test.txt”, ”r”)
    outfile = open(”test.parsed.txt”, ”w”)
    for line in testfile:
        tree, p = parseSentence(g, line.strip())
        print(list2str(tree))
        print(p)
        print(”--------------”)
        if len(tree) == 0:
            outfile.write(”\n”)
        else:
            outfile.write(list2str(tree) + ”\n”)
    testfile.close()
    outfile.close()

    from PYEVALB import scorer
    s = scorer.Scorer()
    s.evalb(”test.gold.txt”, ”test.parsed.txt”, ”result.txt”)


def testSentence(s):
    g = readGrammar(”treebankrules.json”)
    tree, p = parseSentence(g, s)
    print(tree)
    print(p)


if __name__ == ”__main__”:
    testToyGrammar()
    testTreebankGrammar()
```