Reference Solution for PA 2

Alexander Rush

1 Introduction

This document describes the reference solution in Python for Programming Assignment 2. It is meant as a teaching aid for students taking the class who have completed the assignment. **Please do not distribute this document to students taking the class in future sessions or post outside of the Coursera forums. Doing so will be considered a violation of the honor code.**

We begin with the imports necessary for the program.

```
from __future__ import division
import sys, json
```

2 Managing the PCFG

In the first section we provide scaffolding for the PCFG. First we read in the counts from a file handle and group them into dictionaries. Next we define the maximum-likelihood estimates based on these counts. Finally we specify RARE words based on the counts.

```
class PCFG:
  "Store the counts from a corpus."

  def __init__(self, nonterms, bin_rules, unary_rules, words):
    "Initialize the PCFG."
    self.nonterms = dict(nonterms)
    self.bin_rules = dict(bin_rules)
    self.unary_rules = dict(unary_rules)
    self.words = words

    # Set up binary rule table.
    self.bin_rule_table = {}
    for (a, b, c), count in bin_rules:
      self.bin_rule_table.setdefault(a, [])
      self.bin_rule_table[a].append((b, c))

  def has_unary_rule(self, nonterm, word):
    "Does the grammar have this unary rule?"
```

```python
    return (nonterm, word) in self.unary_rules

  def nonterminals(self):
    "Returns the list of nonterminals."
    return self.nonterms.keys()

  def rules(self, a):
    "Returns all the binary rules of the form a -> b c."
    return [(a, b, c) for b, c in self.bin_rule_table.get(a, [])]

  def binary_rule_prob(self, rule):
    "Probability of a binary rule."
    return self.bin_rules[rule] / self.nonterms[rule[0]]

  def unary_rule_prob(self, rule):
    "Probability of a unary rule."
    return self.unary_rules[rule] / self.nonterms[rule[0]]

  def is_rare_word(self, word):
    "Is a word rare in this PCFG."
    return self.words.get(word, 0) < 5

  @staticmethod
  def from_handle(handle):
    "Read the rules from a file handle."
    nonterms = []
    bin_rules = []
    unary_rules = []
    words = {}
    for l in handle:
      t = l.strip().split()
      count = float(t[0])
      if t[1] == "NONTERMINAL":
        nonterms.append((t[2], count))
      if t[1] == "BINARYRULE":
        bin_rules.append(((t[2], t[3], t[4]), count))
      if t[1] == "UNARYRULE":
        unary_rules.append(((t[2], t[3]), count))
        words.setdefault(t[3], 0)
        words[t[3]] += count
    return PCFG(nonterms, bin_rules, unary_rules, words)
```

3 Replacing RARE Words

The first question asks us to go through the trees and replace the rare words with _RARE_. The trick to this question is first converting into tree format and then traversing to the leaves of the tree before replacing words. We do this by writing a recursive function.

```python
def replace_rare_words(pcfg, tree):
  "Mutate tree to replace rare words."
  if len(tree) == 3:
    replace_rare_words(pcfg, tree[1])
    replace_rare_words(pcfg, tree[2])
  elif len(tree) == 2:
    if pcfg.is_rare_word(tree[1]): tree[1] = "_RARE_"
```

4 The CKY Algorithm

Sections 2 and 3 focus on the CKY algorithm. Before giving the algorithm we first reintroduce the argmax helper from PA1.

```python
def argmax(ls):
  "Compute the argmax of a list (item, score) pairs."
  if not ls: return None, 0.0
  return max(ls, key = lambda x: x[1])
```

The CKY algorithm should look very similar to the algorithm given in the class notes. We add two optimizations to speed things up.

1. We only consider rules X $\rightarrow$ Y Z that are seen in training.

2. We prune elements of π that have zero probability.

```python
def CKY(pcfg, sentence):
  "Run the CKY algorithm."

  # Define variables to have the same names as notes.
  n = len(sentence)
  N = pcfg.nonterminals()
  x = [""] + sentence
  def q1(X, Y): return pcfg.unary_rule_prob((X, Y))
  def q2(X, Y, Z): return pcfg.binary_rule_prob((X, Y, Z))
  pi = {}
  bp = {}

  # Initialize the chart.
  for i in range(1, n + 1):
    for X in N:
      if pcfg.has_unary_rule(X, x[i]):
        pi[i, i, X] = q1(X, x[i])
        bp[i, i, X] = (X, x[i], i, i)

  # Dynamic program.
  for l in range(1, n):
    for i in range(1, n - l + 1):
      j = i + l
      for X in N:
```

```
        # Note that we only check rules that exist in training
        # and have non-zero probability.
        back, score = \
            argmax([((X, Y, Z, i, s, j),
                     q2(X, Y, Z) * pi[i, s, Y] * pi[s + 1, j, Z])
                    for s in range(i, j)
                    for X, Y, Z in pcfg.rules(X)
                    if pi.get((i, s, Y), 0.0) > 0.0
                    if pi.get((s + 1, j, Z), 0.0) > 0.0
                    ])
        if score > 0.0: bp[i, j, X], pi[i, j, X] = back, score

  # Return the tree root'd in SBARQ.
  if (1, n, "SBARQ") in pi:
    tree = backtrace(bp[1, n, "SBARQ"], bp)
    return tree, score
```

To reconstruct the tree, we write a recursive function over the chart of back-pointers bp.

```
def backtrace(back, bp):
  "Extract the tree from the backpointers."
  if not back: return None
  if len(back) == 6:
    (X, Y, Z, i, s, j) = back
    return [X, backtrace(bp[i, s, Y], bp),
            backtrace(bp[s + 1, j, Z], bp)]
  else:
    (X, Y, i, i) = back
    return [X, Y]
```

5 Putting It Together

The last step is to implement a controller to run the program. For this assignment we only need to have two commands.

- REPLACE - Replace rare words with _RARE_.
- PARSE - Run the parsing algorithm.

We also add a helper function to replace the rare words in the input sentence.

```
def replace_rare_sent(pcfg, sent):
  "Replace rare words in a flat sentence."
  return [word if not pcfg.is_rare_word(word) else "_RARE_" for word in sen
t]


def main(mode, count_file, sentence_file):
```

```
  pcfg = PCFG.from_handle(open(count_file))
  for i,l in enumerate(open(sentence_file)):
    if mode == "PARSE":
      sentence = replace_rare_sent(pcfg, l.strip().split())
      parse, score = CKY(pcfg, sentence)
      print json.dumps(parse)
    elif mode == "REPLACE":
      parse = json.loads(l.strip())
      replace_rare_words(pcfg, parse)
      print json.dumps(parse)

if __name__ == "__main__":
  main(sys.argv[1], sys.argv[2], sys.argv[3])
```

That basic implementation should parse reasonably well and efficiently. There are many possible extensions we might consider adding to this code: smoothing the parameters, lexicalizing the grammar or switching to a dependency representation, or improving the speed of this parser. We encourage you to continue extending your parser based on what you take from this note.