CS 544 Assignment 1 Solutions

13 Feb 2007

Syntax

1. (10 points) Write a CFG to generate the sentences

(1) The man whom a man hired hired another man.
(2) The man (whom a man)^n hired^n hired another man.

Write the derivation (as a sequence of rewrites or as a tree) for n = 2. Explain briefly why we couldn’t do this with a finite-state machine.

\[
\begin{align*}
S &\rightarrow NP \ VP \\
S[\text{gap = whom}] &\rightarrow NP \ VP[\text{gap = whom}] \\
VP &\rightarrow V[\text{subcat = (NP)}] \ NP \\
VP[\text{gap = whom}] &\rightarrow V[\text{subcat = (NP)}] \\
NP &\rightarrow Det \ N \\
NP &\rightarrow NP \ S'[\text{relative}] \\
S'[\text{relative}] &\rightarrow WP[\text{whom}] \ S[\text{gap = whom}] \\
Det &\rightarrow a \\
Det &\rightarrow the \\
Det &\rightarrow another \\
WP[\text{whom}] &\rightarrow whom \\
N &\rightarrow man \\
V[\text{subcat = (NP)}] &\rightarrow hired
\end{align*}
\]
Note: one of these rules (6) is not in Chomsky normal form. For later exercises we will replace it with
\[ S[gap = whom] \rightarrow NP V[\text{subcat } = (NP)] \]

2. (10 points) Write a CFG to generate the sentences

(17) The man who John said bit the dog exists.
(18) The man who John said Mary said bit the dog exists.
(19) The man who (John said Mary said)\(^n\) bit the dog exists.

as well as

(20) The man whom (John said Mary said)\(^n\) the dog bit exists.

It’s okay to generate other sentences as long as they’re good English sentences. Write the derivation for (19) for \( n = 2 \).
S → NP VP

S[gap = who] → VP

S[gap = whom] → NP VP[gap = whom]

S[gap = x] → NP VP[gap = x] \quad x \in \{\text{who, whom}\}

S'[relative] → WP[x] S[gap = x] \quad x \in \{\text{who, whom}\}

VP → V[subcat = \langle NP\rangle] NP

VP → V[subcat = \langle \rangle]

VP[gap = x] → V[subcat = \langle S\rangle] S[gap = x] \quad x \in \{\text{who, whom}\}

VP[gap = whom] → V[subcat = \langle NP\rangle]

Det → the

WP[who] → who

WP[whom] → whom

NP → John

NP → Mary

N → man

N → dog

V[subcat = \langle NP\rangle] → bit

V[subcat = \langle S\rangle] → said

V[subcat = \langle \rangle] → exists
3. (10 points) Do exercise 2 using a tree-adjoining grammar. You shouldn’t need to use any features, just the basic categories $S$, $S'$, NP, VP, V.
Parsing

4. (20 points) Implement the CKY algorithm to solve the recognition task. It should take a grammar in the format

\[
\begin{align*}
S &\rightarrow NP \ VP \\
NP &\rightarrow Det \ N \\
NP &\rightarrow NP \ PP \\
Det &\rightarrow the \\
N &\rightarrow man \\
\end{align*}
\]

e tc., and a sequence of input sentences (one per line, words separated by whitespace), and output a yes or no for each sentence. Demonstrate your implementation using your grammar from exercise 1 and sentence (2) for \( n = 5 \). If your grammar isn’t in Chomsky normal form, you can either convert it into Chomsky normal form (manually or automatically), or you can hack your parser to allow some non-Chomsky-normal-form rules. But your parser must work correctly for your grammar (with any input sentence) and any Chomsky-normal-form grammar.

Here is a sample Python implementation for CKY including probabilities and pruning:

```python
#!/usr/bin/env python
import math

class Item(object):
    def __init__(self, x, p, ants):
        self.x = x
        self.p = p
        self.ants = ants

    def __eq__(self, other):
        return self.x == other.x

    def __hash__(self):
        return hash(self.x)

    def merge(self, other):
        if other.p > self.p:
            self.p = other.p
            self.ants = other.ants

    def reconstruct(self):
        children = []
        for ant in self.ants:
            if isinstance(ant, Item):
                children.append(ant.reconstruct())
            else:
```
children.append(ant)

return "(%s,)%s" % (self.x, "\s".join(children))

class Cell(object):
    def __init__(self):
        self.items = {}

    def add(self, item):
        if item in self.items:
            self.items[item].merge(item)
        else:
            self.items[item] = item

    def prune(self, threshold, priors):
        best = None
        for item in self.items:
            best = max(best, item.p+priors[item.x])
        dead = []
        for item in self.items:
            if item.p+priors[item.x] < best+threshold:
                dead.append(item)
        for item in dead:
            del self.items[item]

    def __iter__(self):
        return iter(self.items)

class Grammar(object):
    def __init__(self, start):
        self.start = start
        self.rules = {}

    def add(self, lhs, rhs, p):
        self.rules.setdefault(rhs, {})
        self.rules[rhs][lhs] = p

    def parse(self, w, priors):
        n = len(w)
        chart = [[[Cell() for j in range(n+1)] for i in range(n+1)]
        for i in range(n):
            for (x,p) in self.rules.get((w[i],),{}).iteritems():
                chart[i][i+1].add(Item(x, p, (w[i],)))
        for l in range(2,n+1):
            for i in range(n-l+1):
                j = i+l
                for k in range(i+1,j):
                    for item1 in chart[i][k]:
                        for item2 in chart[k][j]:
                            for (x,p) in self.rules.get((item1.x,item2.x),{}).iteritems():
                                chart[i][j].add(Item(x, p+item1.p+item2.p, (item1, item2)))
                                chart[i][j].prune(math.log(0.001), priors)
        for item in chart[0][n]:
            if item.x == self.start:
                return item
return None

if __name__ == "__main__":
    import sys, re
    import time

g = Grammar("S")
rulere = re.compile(r"(\S+)\s*->\s*(.*)\s*#\s*(.*)")
for line in file(sys.argv[1]):
    lhs, rhs, p = rulere.match(line).groups()
    g.add(lhs, tuple(rhs.split()), math.log(float(p)))

priors = {}
for line in file(sys.argv[2]):
    lhs, p = line.split()
    priors[lhs] = math.log(float(p))

for line in sys.stdin:
    start_time = time.clock()
    words = line.split()
    goal = g.parse(words, priors)
    sys.stderr.write("length␣%s	" % len(words))
    sys.stderr.write("time␣%s
" % (time.clock()-start_time))
    if goal is not None:
        print goal.reconstruct()
        sys.stderr.write("p=%s
" % goal.p)
    else:
        print ">(no␣parse)"
        sys.stderr.write("p=-inf\n")

5. (20 points) Extend your implementation to allow probabilities on rules and output trees. It should
take a grammar in the format

```
S -> NP VP # 1.0
NP -> Det N # 0.3
NP -> NP PP # 0.7
Det -> the # 1.0
N -> man # 0.33
```

etc., and a sequence of input sentences, and output a tree for each sentence, one per line, in the
following format:

```
(S (NP (Det The) (N man)) (VP (V saw) (NP (Det the) (N woman))))
```

Note: you will probably want to use log-probabilities instead of probabilities to avoid underflow.
Demonstrate your implementation again using the same grammar (with whatever probabilities you
want) and sentence from the previous exercise.

6. (10 points) Now let’s work with some real data. Implement an estimator that takes the output of
rules.py and learns a PCFG from it. Run your estimator on some training data:

```
cat sec02-21.mrg | preprocess.sh | binarize.py | rules.py | your-estimator
```

7. (10 points) Now we will test and improve the parser’s speed.
• Run your parser using the grammar learned from sec02-21.mrg (with start symbol S) on test.txt, or at least on the first ten sentences of it. Have your parser record the time spent on each sentence.

• Show a plot of time vs. sentence length. Use a log-log scale: an $O(n^k)$ function will appear as a straight line with slope $k$. What is the observed time complexity of your parser?

• Make your estimator also produce a table of probabilities $P(X)$. The format should be:

```
S 0.1
NP 0.2
etc.
```

• Then modify your parser to do threshold pruning (use a threshold of 0.001) using the probabilities $P(X)$ as your heuristic function. Record the parsing times again and show the plot of time vs. sentence length again. How did pruning affect the speed of your parser?

Here is a plot of the parsing time, with and without pruning, compared with ideal cubic-time curves:

![Plot of parsing time with and without pruning](image)

The actual parsing time looks like it might be slightly worse than cubic-time in this range. Note that pruning provides only a linear speedup.

8. (10 points) Run your parser on all 100 sentences of test.txt and evaluate the parse quality:

```
cat your-parser-output | unbinarize.py > your-parser-output.post
evalb -P COLLINS.prm test.mrg your-parser-output.post
```

Use the scores for “All”, not “len $\leq$ 40”. Calculate the F1-measure ($\frac{2 \cdot \text{precision} \cdot \text{recall}}{\text{precision} + \text{recall}}$). Your score must be at least 71% to get full credit for this exercise.