Homework 3
Due Tuesday, 02/19/08

Please turn in a print-out of your homework solutions at the beginning of class. If a program is required for a problem then you should provide sample input and output. (If the input is data from an on-line source then you may just indicate where the file can be found.) Programming solutions to problems should be placed in separate files whose names indicate the problem number (e.g. prob1.py). These separate files should be put into a folder whose name includes the assignment number and your name (e.g., Hmwk1-yourname). This folder should be dragged into the class dropoff folder at

`/common/cs/cs181/dropbox`

While I expect that I will be able to do most of the grading based on the papers that you hand in at the beginning of class, I would like to have access to the programs so that I can test them if necessary. I should be able to load your file into python and have it compile and run without error. All programming solutions should be fully documented and use good variable names.

Note that the files on our system can be accessed using ssh via the Mac server xserv.cs.pomona.edu or the linux server linus.cs.pomona.edu. Files and/or folders can be deposited into the dropbox remotely by using a program utilizing the sftp protocol.

Warning: There are several different versions of Bird et al floating around. From now on all assignments will be relative to the pdf edition of the entire book dated January 24, 2008, and available via the course web page.

1. Please do Problem 4.2 on page 39 of JM.
2. Please do Problem 4.3 on page 39 of JM.
3. Please do Problem 4.5 on page 39 of JM.

   Solutions to 1-3 from Erik:

```
#This program reads data from standard input and prints to standard output the
#10 most common n-grams in the input, where n is a command-line argument. Line
#breaks, capitalization, and leading and trailing punctuation are ignored.
#Words are broken at spaces and retain any interior punctuation.

import sys,re,math

num_to_print = 100  #Number of ngrams to print
threshold = 5       #Max value to use cstar for

def addNGram(lst):
    """ Increases the count of the given ngram as a list. """
    ngram = tuple(lst)
    #Update an old entry or create a new one
    if ngram in ngrams: ngrams[ngram] = ngrams[ngram] + 1
    else : ngrams[ngram] = 1

#Check for a valid argument
if len(sys.argv) != 2 or not sys.argv[1].isdigit() or int(sys.argv[1]) < 1:
    print "usage: python ngrams.py [n]"
```

```
    print "\twhere n is a positive integer giving the size of the n-grams"
    exit()

#Initialize the n-gram counts
ngrams = {}

#Get input and normalize it
n = int(sys.argv[1])
data = sys.stdin.read().replace('\n', ' ')  #Ignore linebreaks
words = data.split()
words = map(lambda word: word.lower(), words)

#Ensure that the input is long enough
if n > len(words):
    print 'No %d-grams in input' % n
    exit()

#Initialize a ring buffer to hold the n-grams
buffer = [words[i] for i in xrange(n)]
bufferLoc = 0
addNGram(buffer)

#Count the n-grams
numNGrams = 0
for i in xrange(n, len(words)):
    buffer[bufferLoc] = words[i]
    addNGram(buffer[bufferLoc+1:] + buffer[:bufferLoc+1])
    numNGrams += 1
    bufferLoc = (bufferLoc+1) % n

#Compute frequencies of frequencies
N = {}
for (k,c) in ngrams.items():
    if not c in N: N[c] = 0
    N[c] += 1

#Compute discounted counts
cstar = {}
for (c,v) in N.items():
    if (c+1) in N: Nc1 = float(N[c+1])
    else: Nc1 = 0.0
    cstar[c] = (c + 1) * (Nc1/N[c])
cstar[0] = float(N[1])/numNGrams

#Get a list of n-gram counts sorted by the count in descending order
vkList = [(v,k) for (k,v) in ngrams.items()]
vkList.sort()
vkList.reverse()
```

```
#Print the top n-grams
print "count\tcstar\t\tprob\t\tGT prob\t\t%d-gram" % n
for (count, ngram) in vkList[:num_to_print]:
    if count < threshold: cs = cstar[count]
    else: cs = count
    print "%5d\t%5f\t%5f\t%5f\t%s" % \
        (count,cs,
         float(count)/numNGrams,float(cs)/numNGrams,
         ngram)

#I ran this program on sections J (Learned) and P (Fiction: Romance and Love
#Stories) of the brown corpus. The first 10 lines of the bigram results from
#section J are pasted below:

#count  cstar           prob            GT prob         2-gram
# 2059  2059.000000     0.011320        0.011320        ('of', 'the')
# 1310  1310.000000     0.007202        0.007202        ('.', 'the')
# 1172  1172.000000     0.006444        0.006444        ('in', 'the')
#  988  988.000000      0.005432        0.005432        (',', 'and')
#  719  719.000000      0.003953        0.003953        (',', 'the')
#  560  560.000000      0.003079        0.003079        ('to', 'the')
#  500  500.000000      0.002749        0.002749        (';', ';')
#  430  430.000000      0.002364        0.002364        ('and', 'the')
#  417  417.000000      0.002293        0.002293        ('.', 'in')

#There wasn't a great deal of difference between P and J for the very most
#common unigrams and bigrams. Slightly farther down the list though, one can see
#the personal pronouns and quotation marks are much more common in section P,
#whereas parentheses were more common in section J, which makes sense.
```

4. Please do Problem 3 on page 111 of Bird.

   Solution from Alex:

```
import nltk
from nltk import defaultdict

pos_tags = defaultdict(set)
for (word, tag) in nltk.corpus.brown.tagged_words():
    pos_tags[word.lower()].add(tag)

ambiguous_words = set()
for (word, tag) in pos_tags.items():
    if len(tag) > 1:
        ambiguous_words.add(word)

ambiguous_occurence = 0.0
total_occurence = len(nltk.corpus.brown.words())
```

```
for word in nltk.corpus.brown.words():
    if word.lower() in ambiguous_words:
        ambiguous_occurence += 1
ambiguous_percent = ambiguous_occurence/len(nltk.corpus.brown.words())*100

print "Number unambiguous tags:", len(pos_tags) - len(ambiguous_words)
print "Number ambiguous words:", len(ambiguous_words)
print "Percentage of ambiguous occurences %.2f" % (ambiguous_percent)

"""
Output:

Number unambiguous tags: 40235
Number ambiguous words: 9580
Percentage of ambiguous occurences 84.15
"""
```

5. Section 4.4-4.5 of Bird discusses some taggers and using backoff to improve taggers. Use the techniques shown there (including backoff) to create the best tagger you can from the built-in taggers of nltk (many of which must be trained on already tagged data).

   Train your tagger on the (tagged) Brown corpus, group a. Then run your tagger on the untagged corpus, group a. Compare your results with the (officially) tagged data. Indicate what percent of the tags you got right. Discuss where most of your mistakes occurred and suggest methods that will improve its accuracy.

   Several of you got to nearly 92% accuracy on testing your tagger on the same data it was trained on – which of course is cheating, but it is what I told you to do. The following is Erik's solution, which barely eaked out the best score.

```
ba = nltk.corpus.brown.tagged_sents(categories='a')

patterns = [(r'^\'?-?([0-9]+,?)+(.[0-9]+)?$', 'CD'),
            (r'^[0-9]+(d|st|nd|rd|th)', 'OD'),
            (r'^[A-Z]', 'NP'),
            (r'^\$', 'NNS')]

tagger = nltk.DefaultTagger('NN')
tagger = nltk.AffixTagger(ba, affix_length=1, min_stem_length=3,backoff=tagger)
tagger = nltk.AffixTagger(ba, affix_length=-1, min_stem_length=3,backoff=tagger)
tagger = nltk.AffixTagger(ba, affix_length=2, min_stem_length=3,backoff=tagger)
tagger = nltk.AffixTagger(ba, affix_length=-2, min_stem_length=3,backoff=tagger)
tagger = nltk.AffixTagger(ba, affix_length=3, min_stem_length=3,backoff=tagger)
tagger = nltk.AffixTagger(ba, affix_length=-3, min_stem_length=3,backoff=tagger)
tagger = nltk.RegexpTagger(patterns,backoff=tagger)
tagger = nltk.UnigramTagger(ba, backoff=tagger)
tagger = nltk.BigramTagger(ba, backoff=tagger)
tagger = nltk.TrigramTagger(ba, backoff=tagger)

print "Accuracy:", nltk.tag.accuracy(tagger, ba)
```

```
#Output is as follows:

#Accuracy: 0.918720289596
```