Homework #4: CMPT-413

Due on Mar 22, 2011

Anoop Sarkar – http://www.cs.sfu.ca/~anoop/teaching/cmpt413

Only submit answers for questions marked with †. For the questions marked with a ††; choose one of them and submit its answer.

(1) Important! To solve this homework you must read the following chapters of the NLTK Book, available online at *http://www.nltk.org/book*

- Chapter 7. Extracting Information from Text
- Chapter 8. Analyzing Sentence Structure

(2) Once we have some text that has been tagged with part of speech labels, we can *chunk* words together into non-overlapping spans of text. Each chunk corresponds to some meaningful unit, e.g. we can find all chunks that are *noun phrases*. Here is an example sentence from the Wall Street Journal where the noun phrases are marked up with brackets:

> [The/DT market/NN] for/IN [system-management/NN software/NN] for/IN [Digital/NNP 's/POS hardware/NN] is/VBZ fragmented/JJ enough/RB that/IN [a/DT giant/NN] such/JJ as/IN [Computer/NNP Associates/NNPS] should/MD do/VB well/RB there/RB ./.

In this question, we will use regular expressions on the part of speech tags to identify chunks. First, we need to put the input sentence into the right format for NLTK:

```
from nltk import chunk
tagged_text = """
The/DT market/NN for/IN system-management/NN software/NN for/IN
Digital/NNP 's/POS hardware/NN is/VBZ fragmented/JJ enough/RB
that/IN a/DT giant/NN such/JJ as/IN Computer/NNP Associates/NNPS
should/MD do/VB well/RB there/RB ./.
"""
input = chunk.tagstr2tree(tagged_text)
print input

cp = chunk.RegexpParser("NP: {<DT><NN>}")
print cp.parse(input)
```

Examine the output after executing the above. You will notice that the function `tagstr2tree` automatically puts a S chunk around the entire sentence (it assumes that the input is a full sentence). We are interested only in noun phrase (NP) chunks here, so we can ignore the S chunk.

We can now use the NLTK regular expression chunker to identify some NP chunks in the sentence:

```
cp = chunk.RegexpParser("NP: {<DT><NN>}")
print cp.parse(input)
```

The above code finds the following two NP chunks in the sentence:

```
  (NP: ('The', 'DT') ('market', 'NN'))
  (NP: ('a', 'DT') ('giant', 'NN'))
```

You can provide multiple regexp patterns for identifying NPs (and also provide comment strings) using this syntax:

```
grammar = r"""
NP:
   {<DT>?<JJ>*<NN>}    # chunk determiners, adjectives and nouns
   {<NNP>+}            # chunk sequences of proper nouns
"""
cp = chunk.RegexpParser(grammar)
```

Note that the order of the patterns is important. You can debug your regexp patterns by using `cp.parse(input, trace=1)` which provides detailed information on the order of the pattern matching.

Provide a program that chunks the example sentence provided and identifies all five of the noun phrase chunks as shown in the marked up example above. Print out the chunked output for the sentence.

(3) †† Provide a regular expression based chunker (using the NLTK chunker) to identify noun phrase chunks for the CONLL-2000 chunk dataset which contains Wall Street Journal text that has been chunked by human experts. Your chunker should at least have 91% accuracy.

Since you will have to deal with a large variety of noun phrase chunks you will need to generalize your regexp patterns. A *tag pattern* is a sequence of part-of-speech tags delimited using angle brackets, e.g. `<DT><JJ><NN>`. Tag patterns are the same as the regular expression patterns we have already seen, except for two differences which make them easier to use for chunking. First, angle brackets group their contents into atomic units, so `<NN>+` matches one or more repetitions of the tag NN; and `<NN|JJ>` matches the NN or JJ. Second, the period wildcard operator is constrained not to cross tag delimiters, so that `<N.*>` matches any single tag starting with N.

To test the accuracy of your chunker use the built-in NLTK function:

```
from nltk import chunk
from nltk.corpus import conll2000

cp = chunk.RegexpParser("NP: {<DT><NN>}")
print chunk.accuracy(cp, conll2000.chunked_sents('test', chunk_types=('NP',)))
```

You can compare the output of your chunker with the gold standard to find out which chunks you are missing. For instance, the following code prints the gold standard and then prints the chunker output:

```
gold_tree = conll2000.chunked_sents('train', chunk_types=('NP',))[1]
print gold_tree
print cp.parse(gold_tree.flatten())
```

Use the following steps in your development process:

a. Write down a regexp chunker using tag patterns that can identify the following examples of noun phrases:

```
another/DT sharp/JJ dive/NN
trade/NN figures/NNS
any/DT new/JJ policy/NN measures/NNS
earlier/JJR stages/NNS
Panamanian/JJ dictator/NN Manuel/NNP Noriega/NNP
his/PRP$ Mansion/NNP House/NNP speech/NN
3/CD %/NN to/TO 4/CD %/NN
more/JJR than/IN 10/CD %/NN
the/DT fastest/JJS developing/VBG trends/NNS
```

b. Write a tag pattern to match noun phrases containing plural head nouns, e.g. `many/JJ types/NNS`, `two/CD weeks/NNS`, `both/DT new/JJ positions/NNS`. Try to do this by generalizing the tag pattern that handled singular noun phrases.

c. Write tag pattern to cover noun phrases that contain gerunds, e.g. `the/DT receiving/VBG end/NN`, `assistant/NN managing/VBG editor/NN`.

d. Write one or more tag patterns to handle coordinated noun phrases, e.g. `July/NNP and/CC August/NNP`, `all/DT your/PRP$ managers/NNS and/CC supervisors/NNS`, `company/NN courts/NNS and/CC adjudicators/NNS`.

e. Compare your output with the gold standard output on some randomly chosen examples from the training data of CoNLL-2000 dataset. See if there are any NP chunks missing in your output and find tag patterns that will include them. Generalize your tag patterns to avoid having one pattern per example. This should not take too long. If you are spending too long on this part, step back and see if you can take a simpler strategy to the solution.

(4) †† This question is is more advanced than Question 3.

The file `genia3.02-small-pos.txt` contains a small amount of text extracted out of bio-medical journals. In this question, we will test how well the chunker you have developed on the Wall Street Journal can deal with text in a completely different domain.

Note that you will now have to deal with the raw text and convert it into a format suitable for use with your chunker. In particular, you should do the following conversions:

`[` to `-LSB-`

`]` to `-RSB-`

`(` to `-LRB-`

`)` to `-RRB-`

whenever these characters occur as the word or tag in the genia corpus. This ensures that the NLTK tree format does not get confused by these parentheses which occur as actual words and tags.

As you can imagine, the text in this corpus can be very different. Here is a typical example sentence from our bio-medical corpus:

> These/DT findings/NNS should/MD be/VB useful/JJ for/IN therapeutic/JJ strategies/NNS and/CC the/DT development/NN of/IN immunosuppressants/NNS targeting/VBG the/DT CD28/NN costimulatory/NN pathway/NN ./.

But does dealing with a different domain affect your chunker? Run your chunker on this dataset. Depending on the regexp patterns you created for the WSJ text, you may have to tweak your regexp chunker with some additional rules.

Note that we cannot test accuracy on this domain since we do not have human labeled data. We will compare your output with our own chunker on this domain.

(5) † **Hidden Markov models**

Suppose you are given a large amount of text in a language you cannot read (and perhaps you are not even sure that it *is* a language). Faced with a total lack of knowledge about the language and the script used to transcribe it, you try to see if you can discover some very basic knowledge using unsupervised learning. Perhaps it is worth trying to see if the letters in the script can be clustered into meaningful groups.

Hidden Markov models (HMMs) are particularly suited for such a task. The hidden states correspond to the meaningful clusters we hope to find, and the observations are the characters of the text. The text provided to you is taken from the Brown corpus (which is in English, but we will try to "decipher" it anyway). Let us take an HMM with two hidden states, and the observations are to be taken from the list of 26 lowercase English characters plus one extra character for a single space character that separates the words in the text.

We will use the software `umdhmm` which implements the Baum-Welch algorithm for unsupervised training of HMMs. It has been installed in `~anoop/cmpt413/sw` on CSIL Linux machines. The web page for this software is:

http://www.kanungo.com/software/software.html

You can use the esthmm program to estimate the parameters of an HMM based on maximizing the likelihood of a training data sequence. The parameters of an HMM with n states and m observation symbols are: the probability of moving from one state to another (an n by n matrix A), the probability of producing a symbol from a state (an n by m matrix B), and the probability of starting a sequence from a state (a vector of size n).

In the notation used by umdhmm the parameters for $m = 2$ observation symbols and $n = 3$ states is:

```
M= 2
N= 3
A:
0.333 0.333 0.333
0.333 0.333 0.333
0.333 0.333 0.333
B:
0.5   0.5
0.75  0.25
0.25  0.75
pi:
0.333 0.333 0.333
```

While you do not need to implement the unsupervised training of an HMM for this question, you do need to interpret the HMM that is learned.

You are provided the training data from the Brown corpus and it is already in the correct format for the esthmm program. The file name for the training data is brownchars_sf.txt and it contains the first 50000 characters from the science fiction section of the Brown corpus. In this file, the numbers 1 to 26 are an index that specify the ASCII characters 'a' to 'z'. The number 27 is mapped to the space character, and in this data a single space character is used to separate words.

a. Use the training corpus brownchars_sf.txt to train an HMM using esthmm. Provide the HMM that is learned. Try different initializations and see if the learned HMM is different.

b. With careful initialization of the HMM, it is possible to train the HMM to automatically distinguish vowels (let us coarsely define these to be the ASCII characters *a, e, i, o, u*) and consonants (everything else) with the space character being in neither set. Provide a graph that clearly shows (visually) that the trained HMM has learned to distinguish one group (vowels) from the other (consonants).

(6) † In NLTK you can easily represent trees. For instance:

```
from nltk.tree import bracket_parse
sent = '(S (S (NP Kim) (V arrived)) (conj or) (S (NP Dana) (V left)))'
tree = bracket_parse(sent)
print tree[0]
left_tree = tree[0]
print left_tree[0]
```

The above code will print out two constituents of the tree:

```
(S (NP Kim) (V arrived))
(NP Kim)
```

Write a program that prints out *all* the constituents of a tree, one per line, using the NLTK tree handling functions shown above. For the above input it should produce:

```
(S:
  (S: (NP: 'Kim') (V: 'arrived'))
  (conj: 'or')
  (S: (NP: 'Dana') (V: 'left')))
(S: (NP: 'Kim') (V: 'arrived'))
(NP: 'Kim')
(V: 'arrived')
(conj: 'or')
(S: (NP: 'Dana') (V: 'left'))
(NP: 'Dana')
(V: 'left')
```

(7) Write down five trees, one for each reading of the phrase *natural language processing course*.

(8) Run the recursive descent parser demo:

```
from nltk.app import rdparser
rdparser()
```

(9) Chapter 8 of the NLTK book provides a good overview of the grammar development process that can be used to describe the *syntax* of natural language sentences. The notion of a context-free grammar allows us to describe nested constituents unlike a chunking grammar. Based on the ideas provided in Chapter 8 of the NLTK book and the lecture notes, write a context-free grammar that can recognize the following sentences:

```
(26a) Jodie won the 100m freestyle
(26b) 'The Age' reported that Jodie won the 100m freestyle
(26c) Sandy said 'The Age' reported that Jodie won the 100m freestyle
(26d) I think Sandy said 'The Age' reported that Jodie won the 100m freestyle
```

Write down your context-free grammar using the following format:

```
productions = """
S -> NP VP
NP -> 'John' | 'Mary' | 'Bob' | Det N | Det N PP
VP -> V NP | V NP PP
V -> 'saw' | 'ate'
Det -> 'a' | 'an' | 'the' | 'my'
N -> 'dog' | 'cat' | 'cookie' | 'park'
PP -> P NP
P -> 'in' | 'on' | 'by' | 'with'
"""
```

You can then use your grammar to parse an input sentence. For example, the following code prints out a parse for the sentence *Mary saw Bob* when analyzed using the above grammar.

```
import nltk

grammar = nltk.parse_cfg(productions)
tokens = 'John saw Mary in the park'.split()
cp = nltk.ChartParser(grammar)
for (c,tree) in enumerate(cp.nbest_parse(tokens)):
    print tree
print "number of parse trees=", c+1
```

Print out the parses for the example sentences above using the context-free grammar you developed to analyze them. Also experiment with the chart parser to parse the same sentence.

```
import nltk

grammar = nltk.parse_cfg(productions)
cp = nltk.ChartParser(grammar)
sent = 'Mary saw Bob'.split()
for p in cp.nbest_parse(sent):
    print p
```

(10) † A Treebank is a corpus of sentences such that each sentence is provided with it's most plausible syntax tree as determined by a human expert. You are provided with a Treebank for sentences from the Air Travel Information Service (ATIS) domain in the file `atis3.treebank`.

From this Treebank, extract a context-free grammar. Save the context-free grammar as a text file. Trim down the number of rules in your context-free grammar, either based on the frequency of the rule or manual inspection or both (this step is necessary to reduce the time taken by the parser). Note that the recursive rules and rules with very long right hand sides are the main culprit when it comes to parsing time, especially if they are infrequent.

Use your reduced context-free grammar with the NLTK recursive descent parser or the NLTK chart parser and parse all the sentences in the file `atis.test`. You may need to add lexical rules (e.g. $NNP \rightarrow Miami$) in order to parse some of these sentences.

Submit your Python code and a text file containing a list of parse trees, one for each sentence in `atis.test`. When running from `exec.py` only print the total number of parses reported by the parser. Your program must finish parsing `atis.test` in $\leq$10 minutes on the CSIL Linux machines (a smaller grammar will give you a faster parser).

(11) † Provide a Python file `exec.py` that will execute each of your programs. Run your programs with different test cases (especially the ones provided in the homework as examples). Please provide verbose descriptions to explain how your programs work (when necessary).