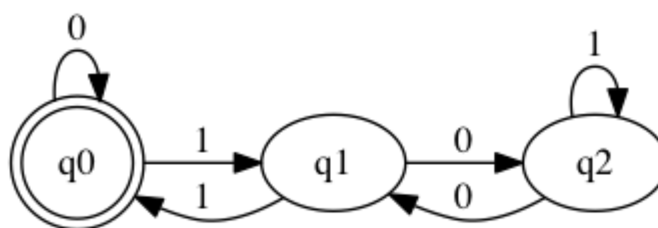
NLP Final Paper

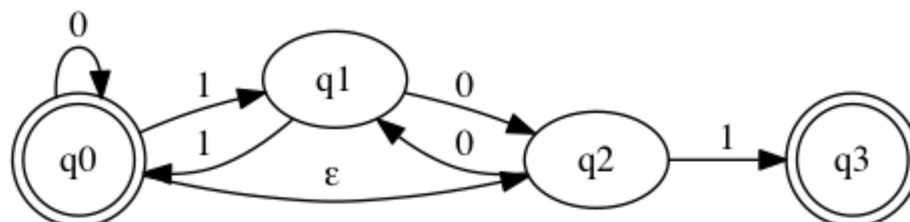
This semester's Tutorial on Natural Language Processing (NLP) was an introduction to the sub-field of Computational Linguistics (CL) which deals with the interaction of humans with computers through the parsing of language which is generally used in larger programs for Computer Assisted Translation, Computer Aided Language Learning, games, and automated chat AI's. In this Tutorial we covered automata, *n*-grams and word counts, Bayesian (conditional) probabilities, Markov models, and the "Natural Language Toolkit".

The first step on this journey of NLP was the finite (state) automaton, and its description and explanation in *Speech and Language Processing*, the first of two textbooks to be used in the course. Both Deterministic and Non-Deterministic Finite Automata (DFAs and NFAs, respectively) are defined by a set of states $Q$, a set of inputs $\Sigma$, a transition function $\delta$ a start state $q_0 \in Q$, and a set of accept states $F \subseteq Q$. DFAs have a finite number of states, and take as input finite strings of symbols. The "deterministic" in the name means that, for each input, there is only one possible output. For example, a DFA with binary input $\Sigma$={0, 1} could look like this:




Input comes from the left (starting at q0), and depending on the input, transitions until an accept state is reached. This DFA would accept strings such as "0" (and strings of "0"s), "011", and "010101". NFAs are finite state machines that have transitions between states when the input symbol satisfies the transition condition. These transitions are dependent on prior transitions as well

as the current state. For example, a simple NFA that accepts the input $\Sigma=\{0, 1, \varepsilon^{1}\}$ could look like this:




Another type of problems that can be drawn as NFAs are grammatical rules such as e-insertion and k-insertion, both of which happen between morphemes in a single word.

The next major topic that we looked at was the *n*-gram, a series of (usually) $n \geq 2$words. The probability of an *n*-gram sequence can be expressed as $P(w_n|w_{n-N+1}^{n-1}) = \frac{C(w_{n-N+!}^{n-1} \cdot w_n)}{C(w_{n-N+!}^{n-1})}$ – the count of the desired *n*-gram divided by the count of the given/prior elements – where $w_n$ is the $n^{th}$ word, $w_b^a$ is the sequence of words from the $a^{th}$ to $b^{th}$ word, and *N* is the total number of elements in the *n*-gram. As a simple example, if the string "aabacdbcdaacd" is in a corpus, and we want to know the probability of the trigram "cda" appearing as the next character, we are looking for $P(\text{"}a\text{"}|\text{"}cd\text{"}) = \frac{C(\text{"}cda\text{"})}{C(\text{"}cd\text{"})}$, which is ½ ("aaba*cd*b***cd*a**acd").

After *n*-grams, we looked at Markov chains, which are mathematical systems that undergo transitions from one state to another, between a finite or countable number of possible states.[2] Transitions to subsequent states in Markov chains are only dependent on the current state, not anything that happens before that. Markov models are models that use Markov chains as is;

[1] ε = empty string

[2] Wikipedia contributors, "Markov chain," *Wikipedia, The Free Encyclopedia,* http://en.wikipedia.org/w/index.php?title=Markov_chain&oldid=461950942 (accessed November 29, 2011).

whereas HMMs have the original Markov chain plus a set of observable states that have certain probabilities of appearing given certain hidden states. In Natural Language Processing, HMMs are used in speech recognition and part-of-speech tagging. For probabilistic sequences, like determining the next state in a Markov chain, or the next word in an *n*-gram, we can use Bayes' theorem, which can be simplified to $P(B|A) = \frac{P(A|B) \cdot P(B)}{P(A)}$ (the probability of *b* given *a* is seen).

This simplification is derived from the union of conditional probability calculations, as follows:

*Derivation of Bayes' Theorem*

$$P(A|B) = \frac{P(A \cap B)}{P(B)}$$

$$P(B|A) = \frac{P(A \cap B)}{P(A)}$$

$$\Rightarrow P(A \cap B) = P(A|B) \cdot P(B) = P(B|A) \cdot P(A)$$

$$\Rightarrow P(B|A) = \frac{P(A|B) \cdot P(B)}{P(A)}$$

After Bayes' theorem, we started looking at the Natural Language Toolkit (NLTK).[3] The NLTK text[4] has chapters on coding basics, string manipulation, tagging, syntax, and grammars. The first few chapters were background knowledge, answering questions such as "what is a function?" and current style conventions in writing Python programs. This chapter also discussed algorithm design, discussing recursion, space and time requirements, and dynamic programming – breaking problems into smaller problems (polynomial-time reduction).

The next topic we looked at was part-of-speech tagging. Part-of-speech tagging aims to assign the correct parts-of-speech to words in a sentence. This process, after some training, uses the sentence structures (S-V-O...) of the language being used, as well as Markov models and Bayes'

---

[3] http://www.nltk.org/

[4] http://www.nltk.org/book

theorem to assign the tags. The NLTK code comes with different corpora bundled with it, among them wordbanks (Brown, Penn) and texts from Project Gutenberg (such as Carroll's *Alice's Adventures in Wonderland*). In this course, part-of-speech tagging was only used on these corpora. The Toolkit has a `pos_tag()` method that assigns the tags to words, and the API describes different types of taggers, using tags of surrounding *n*-grams, and other rules. The following chapter is on text classification. As this was covered in some depth in the Artificial Intelligence course, I didn't spend too much time on it.

The next chapter of new material was on information extraction. In the case of NLTK, this deals with syntax and grammars, and drawing syntax trees. Background information (such as what all the NP, PP, VP, etc. tags mean) to this chapter was learned in the Grammar as Science course taught at Marlboro in Fall 2010.

Chapter eight consisted of more syntax as well as the introduction of Context Free Grammars (CFGs). CFGs (or phrase-structure grammars) are formal grammars such that the grammar can be represented as A -> B where A is a non-terminal symbol and B can consist of terminal and/or non-terminal symbols or can be empty. An example of this is the common sentence definition "S -> NP VP", which defines a sentence as being constructed from a Noun Phrase and a Verb Phrase. There are functions in NLTK that parse a grammar with rules written in the A -> B form and print the phrase structure of a given sentence as nested parentheses. For example:

```
grammar1 = nltk.parse_cfg("""
S -> NP VP
VP -> V NP | V NP PP
PP -> P NP
V -> "saw" | "ate" | "walked"
NP -> "John" | "Mary" | "Bob" | Det N | Det N PP
Det -> "a" | "an" | "the" | "my"
N -> "man" | "dog" | "cat" | "telescope" | "park"
P -> "in" | "on" | "by" | "with"
""")
sent = "the man saw Bob with the telescope".split()
rd_parser = nltk.RecursiveDescentParser(grammar1)
for tree in rd_parser.nbest_parse(sent):
    print tree
```

The output of this code is:

```
(S
  (NP (Det the) (N man))
  (VP (V saw) (NP Bob) (PP (P with) (NP (Det the) (N telescope)))))
```

This output can be easily turned into an instance of NLTK's `Tree()` class and read into a `draw()` function to output a visual syntax tree, as follows:

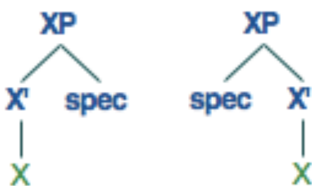
```
tree1 = nltk.Tree('S', [nltk.Tree('NP', [nltk.Tree('Det', ['the']),
        nltk.Tree('N', ['man'])]), nltk.Tree('VP', [nltk.Tree('V', ['saw']),
        nltk.Tree('NP', ['Bob']), nltk.Tree('PP', [nltk.Tree('P', ['with']),
        nltk.Tree('NP', [nltk.Tree('Det', ['the']), nltk.Tree('N', ['telescope'])
])])])])])

tree1.draw()                  # tree1.png
(http://cs.marlboro.edu/courses/fall2011/jims_tutorials/elias/2011-11-29.attachments/tree1.png)
```

Chapter nine continued with CFGs, and introduced X̄ (X-bar, X') syntax, named after Noam Chomsky's X̄ theory. The theory attempts to identify syntactic features presumably common to all those human languages that fit in a presupposed framework.[5] One such feature is the X̄, a non-terminal subdivision of the *X-Phrase* (X double bar, X̿), that can be expressed in syntax rules as "XP -> specifier X̄". Using NLTK's `Tree()` class, this can be encoded as:

```
tree = nltk.Tree('XP', [nltk.Tree('X\'', ['X']), nltk.Tree('spec')])
tree = nltk.Tree('XP', [nltk.Tree('spec'), nltk.Tree('X\'', ['X'])])
```

Using the `draw()`method, this gives the following syntax trees:




Chapter nine also adds subcategorization ([(±) A, B, C]), a phrase with (or without) constraints A, B, and/or C, to the heads (nodes). These features allow for more specific grammars, such as in languages with number/gender agreements, such as the following Spanish example,

[5] http://en.wikipedia.org/wiki/X-bar_theory

which recognizes the following sentences:

- "Un cuadro hermoso"
- "Unos cuadros hermosos"
- "Una cortina hermosa"
- "Unas cortinas hermosas"

```
% start S
 # Grammar productions
 S -> NP[GND=?g, NUM=?n]
 NP[GND=?g, NUM=?n] -> A[GND=?g, NUM=?n] NP[GND=?g, NUM=?n]
 NP[GND=?g, NUM=?n] -> N[GND=?g, NUM=?n] ADJ[GND=?g, NUM=?n]
 # Lexical productions
 # Articles
 # Masculine
 A[GND=m, NUM=s] -> "un"
 A[GND=m, NUM=p] -> "unos"
 # Feminine
 A[GND=f, NUM=s] -> "una"
 A[GND=f, NUM=p] -> "unas"
 # Nouns
 # Masculine
 N[GND=m, NUM=s] -> "cuadro"
 N[GND=m, NUM=p] -> "cuadros"
 # Feminine
 N[GND=f, NUM=s] -> "cortina"
 N[GND=f, NUM=p] -> "cortinas"
 # Adjectives
 # Masculine
 ADJ[GND=m, NUM=s] -> "hermoso"
 ADJ[GND=m, NUM=p] -> "hermosos"
 # Feminine
 ADJ[GND=f, NUM=s] -> "hermosa"
 ADJ[GND=f, NUM=p] -> "hermosas"
# -------------------------
```

The final chapter I looked at this semester, chapter 10, was about sentence analysis, logic, and discourse representation. Apart from the discourse representation, I had seen something about each section in other texts. Discourse is written as first order logic and NLTK has methods to represent logic in a table.

Overall I am happy with what was achieved in the tutorial, even though there were some areas, like Markov models that required much more time than the others. I hope to use this information, especially the NLTK, in future projects, which I expect will be related to machine

translation.