Introduction to Natural Language Processing
Baseline QA system
Stanford Parser and Dependencies
Announcements

- Out of town on Thursday
- No office hours on Thursday this week or next
Homework 6
Baseline System

- Due tomorrow night
  - "Easy" questions
  - Don't expect to get close to 100% precision and recall
  - Get something working that returns a reasonable answer for all the questions

- We'll release HW7 on Thursday along with the first development set
  - 2 more fables & 1 more blog

- HW7 will include slightly more difficult questions
Baseline System
Baseline System

- To get started do something simple

**First Steps**
- Look for lexical overlap between the question and each sentence
- Rank each sentence by number of words in common
- Return the one with the highest overlap
Baseline System

Next Steps

- Classify questions into types
- Build simple parsers for each question type:
  - Who, what, when, where, etc.
  - E.g., identify type of expected answer
- Take word order into account
- Stem/lemmatize words
- Look for named entities of the proper type near keywords
- Rephrase question
Beyond this HW

- More advance (not for this HW)
  - Use the parse trees and dependency graphs
  - Use wordnet to identify synonymous answers
  - Use verbnet to look for specific argument types
  - Use the Story Intention Graph to answer deep questions about the character motivations
```python
def get_bow(tagged_tokens, stopwords):
    return set([t[0].lower() for t in tagged_tokens if t[0].lower() not in stopwords])
# qtokens: is a list of pos tagged question tokens with SW removed
# sentences: is a list of pos tagged story sentences
# stopwords is a set of stopwords

def baseline(qbow, sentences, stopwords):
    # Collect all the candidate answers
    answers = []
    for sent in sentences:
        # A list of all the word tokens in the sentence
        sbow = get_bow(sent, stopwords)

        # Count the # of overlapping words between the Q and the A
        # & is the set intersection operator
        overlap = len(qbow & sbow)

        answers.append(((overlap, sent))

    # Sort the results by the first element of the tuple (i.e., the count)
    # Sort answers from smallest to largest by default, so reverse it
    answers = sorted(answers, key=operator.itemgetter(0), reverse=True)

    # Return the best answer
    best_answer = (answers[0])[1]
    return best_answer
```
Parse Tree Demo (Cont.)
Simple Subtree Match

• Given our tree
  • Does it contain the subtree?
  • (VP (*) (PP))
  • * is a wildcard (can be anything)
def matches(pattern, root):
    if root is None and pattern is None:  
        return root  
    elif pattern is None:
        return root
    elif root is None:
        return None

    plabel = pattern if isinstance(pattern, basestring) else pattern.node
    rlabel = root if isinstance(root, basestring) else root.node

    if plabel == "*":  
        return root
    elif plabel == rlabel:
        for pchild, rchild in zip(pattern, root):
            match = matches(pchild, rchild)
            if match is None:
                return None
        return root

    return None
Code in VM
Stanford Dependencies

- Lots of dependency relations
- Only a subset shown here
- Several different types
  - Basic
    - Form a tree structure
    - Each word is the dependent of another
  - Collapsed
    - More direct relationships between content words
    - Might form a cycle
Stanford Dependencies (Arguments)

- **arg** – Argument
  - **comp**: complement
    - **acom**: adjectival complement
    - **ccomp**: clausal complement with internal subject
    - **xcomp**: clausal complement with external subject
  - **obj**: object
    - **dobj**: direct object
    - **iobj**: indirect object
    - **pobj**: object of preposition
  - **subj**: subject
    - **nssubj**: nominal subject
    - **csubj**: clausal subject
    - **xsubj**: controlling subject (not basic)

*This is categorized differently in the Stanford Dependency Manual*
Stanford Dependencies (Modifiers)

- Mod: modifier
  - amod: adjectival modifier
  - appos: appositional modifier
  - advcl: adverbial clause modifier
  - vmod: reduced, non-finite verbal modifier
  - advmod: adverbial modifier
    - neg: negative modifier
  - quantmod: quantifier modifier
  - nn: noun compound modifier
  - poss: possession modifier
  - prep: prepositional modifier
Stanford Dependencies

- **acomp**: Adjectival Complement
  - An adjectival phrase which functions like an object of the verb
  - "She looks very beautiful"
  - acomp(looks, beautiful)

- **advmod**: Adverbial Modifier
  - An adverb or ADVP that modifies the meaning of a word
  - "Genetically modified food"
  - advmod(modified, genetically)

- **amod**: Adjectival Modifier
  - An adjectival modifier of an NP, which modifies the meaning of the NP
  - "Sam eats red meat"
  - amod(meat, red)
Stanford Dependencies

- **nsubj**: Nominal Subject
  - The syntactic subject of a clause
  - "Clinton defeated Dole"
  - nsubj(defeated, Clinton)

- **nsubjpass**: Passive nominal subject
  - A noun phrase that is the syntactic subject of a passive clause
  - "Dole was defeated by Clinton"
  - nsubjpass(defeated, Dole)

- **dobj**: Direct Object
  - The direct object of a verb phrase
  - "She gave me a raise"
  - dobj(gave, raise)
Stanford Dependencies

- **iobj**: Indirect Object
  - The indirect object of a verb phrase
  - "She gave me a raise"
  - iobj(gave, me)

- **tmod**: Passive nominal subject
  - A temporal modifier
  - "Last night, I swam in the pool"
  - tmod(swam, night)
Stanford Dependencies

- **appos**: Appositional modifier
  - An NP immediately to the right of another NP that serves to define or modify that NP
  - "Sam, my brother, arrived"
  - appos(Sam, brother)

- **ccomp**: Clausal complement
  - A dependent clause that has its own internal subject.
  - "He says that you like to swim"
  - ccomp(says, like)
Bell, a company which is based in LA, makes and distributes computer products.

```
nsbj(makes-11, Bell-1)
det(company-4, a-3)
appos(Bell-1, company-4)
nsubjpass(based-7, which-5)
auxpass(based-7, is-6)
rcmod(company-4, based-7)
prep(based-7, in-8)
pobj(in-8, LA-9)
root(ROOT-0, makes-11)
cc(makes-11, and-12)
conj(makes-11, distributes-13)
nn(products-15, computer-14)
dobj(makes-11, products-15)
```
Stanford Dependencies (Collapsed)

- Dependencies involving prepositions, conjuncts, and references to relative clauses are collapsed

- Useful for simplifying relation extraction

- Generally collapsed has several advantages, but NLTK doesn't read them properly
Stanford Dependencies (Collapsed)

- Dependencies involving prepositions, conjuncts, and references to relative clauses are collapsed
- Useful for simplifying relation extraction

dependencies:
- **nsubj**(makes-11, Bell-1)
- **det**(company-4, a-3)
- **appos**(Bell-1, company-4)
- **nsubjpass**(based-7, which-5)
- **auxpass**(based-7, is-6)
- **rcmod**(company-4, based-7)
- **prep**(based-7, in-8)
- **pobj**(in-8, LA-9)
- **root**(ROOT-0, makes-11)
- **cc**(makes-11, and-12)
- **conj**(makes-11, distributes-13)
- **nn**(products-15, computer-14)
- **dobj**(makes-11, products-15)
- Dependencies involving prepositions, conjuncts, and references to relative clauses are collapsed
- Useful for simplifying relation extraction

- `nsubj(makes-11, Bell-1)`
- `det(company-4, a-3)`
- `appos(Bell-1, company-4)`
- `nsubjpass(based-7, which-5)`
- `auxpass(based-7, is-6)`
- `rcmod(company-4, based-7)`
- `prep_in(based-7, LA-9)`
- `root(ROOT-0, makes-11)`
- `cc(makes-11, and-12)`
- `conj(makes-11, distributes-13)`
- `nn(products-15, computer-14)`
- `dobj(makes-11, productes-15)`
Reading in Dependency Graphs

- Each tree spans multiple rows
- A blank line separates parses
- Slightly more involved
  - Read in the string for one tree (i.e., up to a blank line)
  - Create the DependencyGraph from that string
Reading in Dependency Graphs

- Each DependencyGraph consists of a list of nodes
  - nodelist

- Each node is a dict with the following keys
  - **head**: index of the parent (the root doesn't have a head)
  - **word**: the lexical item
  - **rel**: the grammatical relation between the item and the head
  - **tag**: the part of speech tag of the node
  - **deps**: the list of dependent nodes
  - **address**: index of the item in the sentence (starting from 1)
# Read the lines of an individual dependency parse

def read_dep(fh):
    dep_lines = []
    for line in fh:
        line = line.strip()
        if len(line) == 0:
            return nltk.DependencyGraph("\n".join(dep_lines))
        dep_lines.append(line)

    return nltk.DependencyGraph("\n".join(dep_lines))
    if len(dep_lines) > 0 else None
Code in VM