13-0: **Outline**

- n-grams
  - Applications of n-grams
- review - Context-free grammars
- Probabilistic CFGs
- Information Extraction

13-1: **Advantages of IR approaches**

- Recall that IR-based approaches use the “bag of words” model.
- TFIDF is used to account for word frequency.
  - Takes information about common words into account.
  - Can deal with grammatically incorrect sentences.
  - Gives us a “degree of correctness”, rather than just yes or no.

13-2: **Disadvantges of IR**

- No use of structural information.
  - Not even co-occurrence of words
- Can’t deal with synonyms or dereferencing pronouns
- Very little semantic analysis.

13-3: **Advantages of classical NLP**

- Classical NLP approaches use a parser to generate a parse tree.
- This can then be used to transform knowledge into a form that can be reasoned with.
  - Identifies sentence structure
  - Easier to do semantic interpretation
  - Can handle anaphora, synonyms, etc.

13-4: **Disadvantages of class. NLP**

- Doesn’t take frequency into account
- No way to choose between different parses for a sentence
- Can’t deal with incorrect grammar
- Requires a lexicon.
- Maybe we can incorporate both statistical information and structure.

13-5: **n-grams**

- The simplest way to add structure to our IR approach is to count the occurrence not only of single tokens, but of *sequences* of tokens.

  - So far, we've considered words as tokens.
- A token is sometimes called a *gram*
- an n-gram model considers the probability that a sequence of n tokens occurs in a row.
  - More precisely, it is the probability $P(token_i | token_{i-1}, token_{i-2}, ..., token_{i-n})$

13-6: **n-grams**

- We could also choose to count *bigrams*, or 2-grams.
- The sentence "Every good boy deserves fudge" contains the bigrams "every good", "good boy", "boy deserves", "deserves fudge"
- We could continue this approach to 3-grams, or 4-grams, or 5-grams.
- Longer n-grams give us more accurate information about content, since they include phrases rather than single words.
- What's the downside here?

13-7: **Sampling theory**

- We need to be able to estimate the probability of each n-gram occurring.
  - We could do this by collecting a corpus and counting the distribution of words in the corpus.
  - If the corpus is too small, these counts may not be reflective of an n-gram's true frequency.
  - Many n-grams will not appear at all in our corpus.
- For example, if we have a lexicon of 20,000 words, there are:
  - $20,000^2$ = 400 million distinct bigrams
  - $20,000^3$ = 8 trillion distinct trigrams
  - $20,000^4 = 1.6 \times 10^{17}$ distinct 4-grams

13-8: **Application: segmentation**

- One application of n-gram models is *segmentation*
- Splitting a sequence of characters into tokens, or finding word boundaries.
  - Speech-to-text systems
  - Chinese and Japanese
  - genomic data
  - Documents with other characters, such as   representing space.
- The algorithm for doing this is called *Viterbi segmentation*
  - (Like parsing, it's a form of dynamic programming)

13-9: **Viterbi segmentation**

```
input: a string S, a 1-gram distribution P
n = length(S)
words = array[n+1]
best = array[n+1] = 0.0 * (n+1)
best[0] = 1.0

for i = 1 to n
   for j = 0 to i - 1
     word = S[j:i]   ##get the substring from j to i
     w = length(word)
     if (P[word] x best[i - w] >= best[i])
        best[i] = P[word] x best[i - w]
        words[i] = word
### now get best words
result = []
i = n
while i > 0
  push words[i] onto result
  i = i - len(words[i])
return result, best[i]
```

13-10: **Example**

```
Input ‘‘cattlefish’’ P(cat) = 0.1, P(cattle) = 0.3, P(fish) = 0.1.
all other 1-grams are 0.001.
   best[0] = 1.0
i: 1, j: 0 word: 'c'. w = 1
0.001 * 1.0 >= 0.0
   best[1] = 0.001
   words[1] = 'c'

i = 2, j = 0 word = 'ca', w = 2
0.001 * 1.0 >= 0.0
   best[2] = 0.001
   words[2] = 'ca'

i = 2, j = 1 word = 'a', w = 1
0.001 * 0.001 < 0.001
```

13-11: **Example**

```
i = 3, j = 0, word='cat', w=3
0.1 * 1.0 > 0.0
   best[3] = 0.1
   words[3] = 'cat'

i = 3, j = 1, word = 'at', w=2
0.001 * 0.001 < 0.1

i = 3, j = 2, word = 't', w=1
0.001 * 0.001 < 0.1
```

13-12: **Example**

```
i=4, j=0, word='catt', w=4
0.001 * 1.0 > 0.0
   best[4] = 0.001
   words[4] = 'catt'

i=4,j=1 word = 'att', w=3
0.001 * 0.001 < 0.001

i=4, j=2, word='tt', w=2
0.001 * 0.001 < 0.001

i=4, j=3, word='t', w=1
0.001 * 0.1 < 0.001
```

13-13: **Example**

```
i=5, j=0, word='cattl', w=5
0.001 * 1.0 > 0.0
  best[5] = 0.001
  word[5] = 'cattl'

i=5, j=1, word='attl', w=4
0.001 * 0.001 < 0.001

i=5, j=2, word='ttl', w=3
0.001 * 0.001 < 0.001

i=5, j=3, word='tl', w=2
0.001 * 0.1 < 0.001

i=5, j=4, word='l', w=1
0.001 * 0.001 < 0.001
```

13-14: **Example**

```
i=6, j=0, word='cattle', w=6
0.3 * 1.0 > 0.0
  word[6] = 'cattle'
  best[6] = 0.3

etc ...
```

13-15: **Example**

```
best: [1.0 0.001 0.001 0.1 0.001 0.001 0.3 0.001 0.001 0.2]
words: ['c' 'ca' 'cat' 'catt' 'cattl' 'cattle' 'cattlef' 'cattlefi'
'cattlefis' 'fish']

i = 10
push 'fish' onto result
i = i-4
push 'cattle' onto result
i = 0
```

13-16: **What's going on here?**

- The Viterbi algorithm is *searching* through the space of all combinations of substrings.
  - States with high probability mass are pursued.
- The 'best' array is used to prevent the algorithm from repeatedly expanding portions of the search space.
- This is an example of dynamic programming (like chart parsing)

13-17: **Application: language detection**

- n-grams have also been successfully used to detect the language a document is in.
- Approach: consider *letters* as tokens, rather than words.
- Gather a corpus in a variety of different languages (Wikipedia works well here.)
- Process the documents, and count all two-grams.
- Estimate probabilities for Language L with $\frac{count}{\#of2-grams}$ Call this P_L
- Assumption: different languages have characteristic two-grams.

13-18: **Application: language detection**

- To classify a document by language:
  - Find all two-grams in the document. Call this set T.
  - For each language L, the *likelihood* that the document is of language L is: $P_L(t_1) \times P_L(t_2) \times ... \times P_L(t_n)$
  - The language with the highest likelihood is the most probable language.
    - (this is a form of Bayesian inference - we'll spend more time on this later in the semester.)

13-19: **Going further**

- *n*-grams and segmentation provide some interesting ideas:
  - We can combine structure with statistical knowledge.
  - Probabilities can be used to help guide search
  - Probabilities can help a parser choose between different outcomes.
- But, no structure used apart from colocation.
- Maybe we can apply these ideas to grammars.

13-20: **Reminder: CFGs**

- Recall context-free grammars from the last lecture
- Single non-terminal on the left, anything on the right.

  - S -¿ NP VP
  - VP -¿ Verb — Verb PP
  - Verb -¿ 'run' — 'sleep'
- We can construct sentences that have more than one legal parse.
  - "Squad helps dog bite victim"
- CFGs don't give us any information about which parse to select.

13-21: **Probabalistic CFGs**

- A probabalisitc CFG is just a regular CFG with probabilities attached to the right-hand sides of rules.
  - The have to sum up to 1
- They indicate how often a particular non-terminal derives that right-hand side.

13-22: **Example**

```
S -> NP VP (1.0)
PP -> P NP (1.0)
VP -> V NP (0.7)
VP -> VP PP (0.3)
P -> with (1.0)
V -> saw (1.0)
NP -> NP PP (0.4)
NP -> astronomers (0.1)
NP -> stars (0.18)
NP -> saw (0.04)
NP -> ears (0.18)
NP -> telescopes (0.1)
```

13-23: **Disambiguation**

- The probability of a parse tree being correct is just the product of each rule in the tree being derived.
- This lets us compare two parses and say which is more likely.

13-24: **Disambiguation**

```
              S (1.0)
            /        \
       NP(0.1)        VP (0.7)
          |            |      \
     astronomers      V (1.0)    NP (0.4)
                       |          |      \
                      saw      NP (0.18)  PP(1.0)
                                  |         |      \
                                stars     P(1.0)      NP(0.18)
                                            |           |
                                           with        ears
```

```
P1 = 1.0*0.1*0.7*1.0*0.4*0.18*1.0*1.0*0.18 = 0.0009072
```

```
           S (1.0)
         /        \
    NP(0.1)        VP (0.3)
       |            |      \
  astronomers      VP (0.7)      PP(1.0)
                    |     \        |      \
                  V (1.0) NP (0.18) P(1.0)      NP(0.18)
                    |       |        |           |
                   saw    stars     with        ears
```

```
P1 = 1.0*0.1*0.3*0.7*1.0*0.18*1.0*1.0*0.18 = 0.00068
```

13-25: **Faster Parsing**

- We can also use probabilities to speed up parsing.
- Recall that both top-down and chart pasring proceed in a primarily depth-first fashion.
  - They choose a rule to apply, and based on its right-hand side, they choose another rule.
- Probabilities can be used to better select which rule to apply, or which branch of the search tree to follow.
- This is a form of best-first search.

13-26: **Information Extraction**

- An increasingly common application of parsing is *information extraction*.
- This is the process of creating structured information (database or knowledge base entries) from unstructured text.

13-27: **Information Extraction**

- Example:
  - Suppose we want to build a price comparison agent that can visit sites on the web and find the best deals on flatscreen TVs?
  - Suppose we want to build a database about video games. We might do this by hand, or we could write a program that could parse wikipedia pages and insert knowledge such as madeBy(Blizzard, WorldOfWarcraft) into a knowledge base.

13-28: **Extracting specific information**

- A program that fetches HTML pages and extracts specfic information is called a *scraper*.
- Simple scrapers can be built with regular expressions.
  - For example, prices typically have a dollar sign, some digits, a period, and two digits.
  - $[0-9]+.[0-9]{2}

- This approach will work, but it has several limitations
  - Can only handle simple extractions
  - Brittle and page specific

13-29: **Steps in information extraction**

- A more robust system will need to take advantage of sentence structure.
- A typical system will have the following components:
  - Sentence segmenter.
  - Tokenizer.
  - Part of speech tagger.
  - Chunker.
  - Named Entity detector.
  - Relation extractor.

13-30: **POS tagging**

- There are a number of approaches to part-of-speech tagging.
  - We can write rules based on a word's structure. ("-ed" is a past tense verb)
  - We can learn rules based on labeled data.
    - Most common tag - ZeroR.
    - We can use contextual information - n-grams.
    - We can combine them, and learn more complex rules.

13-31: **Chunking**

- A chunk is a larger part of a sentence, such as a noun phrase.
- This will help us identify entities and relations.
- We can identify chunks with a chunk grammar:
  - $NP : < DT >? < JJ > * < NN >$
- Once we've tagged words with parts of speech, we use a parser to identify chunks.
- This can be done top-down or bottom up.

13-32: **Named Entities**

- These are noun phrases that refer to specific individuals, places, or organizations.
- How can we identify them, and what type of entity they are?
- e.g. University of San Francisco: NP - Organization, Barack Obama: NP - Person.
  - Maybe we have a *gazetteer* (lookup table), but this is very brittle.
- We can also build a classifier to label entities.
  - Input: token with a part-of-speech label

- Output: whether it is a Named Entity, and its type.

13-33: **Relation extraction**

- Once we have Named Entities, we would like to know relations between them.
  - In(USF, San Francisco)
- We can write a set of augmented regular expressions to do this.
  - ¡ORG¿(.+)VP in(.+)¡CITY¿ will match ¡organization¿ verb-phrase in blah ¡city¿.
- There will be false positives; getting this highly accurate takes some care.
- We can trade off precision and accuracy here - more restrictive regular expressions might miss some relations, but avoid adding false positives.

13-34: **Summary**

- We can combine the best of probabilistic and classical NLP approaches.
- n-grams take advantage of co-occurrence information.
  - Segmenting, language detection
- CFGs can be augmented with probabilities
- Speeds parsing, deals with ambiguity.
- Information extraction is an increasingly common application.
- Still no discussion of semantics; just increasingly complex syntax processing.