Statistical methods in NLP
Part-of-speech tagging

UNIVERSITY OF GOTHENBURG

Richard Johansson

February 23, 2016

overview of today's lecture

- HMM tagging recap
  - assignment 3
- evaluating and comparing taggers

overview

HMM tagging recap

assignment 3 overview and implementation hints

statistical recipes for tagger evaluation

the next few weeks

UNIVERSITY OF GOTHENBURG

tagging in general: the task

- we are given a list of words such as `['The', 'cat', 'sleeps']`
- our task is to predict a list of tags such as `['DT', 'NN', 'VBZ']`
- this is a **sequence tagging** problem

a probabilistic model of tagging

- the typical probabilistic formulation of a tagger starts from Bayes' rule:

$$
\begin{aligned}
\arg\max_T P(T|W) &= \arg\max_T \frac{P(W|T)P(T)}{P(W)} \\
&= \arg\max_T P(W|T)P(T)
\end{aligned}
$$

- $P(T)$ is like a language model, but for tag sequences instead of word sequences

making the probabilities practical

- we need to make assumptions about $P(T)$ and $P(W|T)$
- in a **bigram tagger**, the probability of the next tag depends **only** on the previous tag (Markov assumption):

$$P(t_n|t_1,\ldots,t_{n-1}) \approx P(t_n|t_{n-1})$$

  - this is called the **transition probability**
- the probability of a word depends **only** on its tag:

$$P(w_n|\text{tags}, \text{other words}) \approx P(w_n|t_n)$$

  - this is called the **emission probability**

hidden Markov models

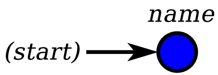
$$P(t_n|t_{n-1}) \qquad P(w_n|t_n)$$

- a model where we have an unknown underlying sequence is called a **hidden Markov** model (HMM)

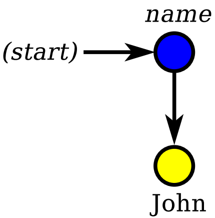
generative story in hidden Markov models

(start)

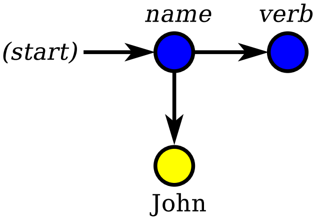
generative story in hidden Markov models



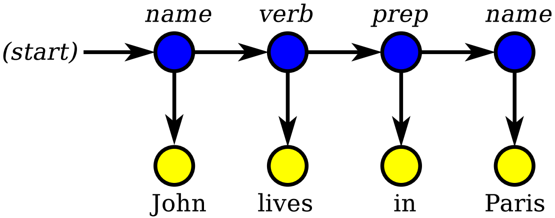
generative story in hidden Markov models



generative story in hidden Markov models



generative story in hidden Markov models



how can we estimate the probabilities?

- to estimate $P(t_n|t_{n-1})$ and $P(w_n|t_n)$, we need a corpus where the part-of-speech tags have been annotated (by humans)

```
The        DT
rifles     NNS
were       VBD
n't        RB
loaded     VBN
.          .

As         IN
interest   NN
rates      NNS
rose       VBD
,          ,
...
```

- in the next lecture, we'll also consider the case where we don't have an annotated corpus

estimating the probabilities

- just like we did with the Naive Bayes classifier, we estimate the probabilities by counting frequencies (MLE):

$$P_{MLE}(\text{noun}|\text{verb}) = \frac{\text{count}(\text{verb, noun})}{\text{count}(\text{verb})} \qquad P_{MLE}(cat|\text{noun}) = \frac{\text{count}(\text{noun: } cat)}{\text{count}(\text{noun})}$$

smoothing...

- smoothing may be useful, in particular if the corpus is small
- for instance, Laplace smoothing for transition probabilities:

$$P(t_n|t_{n-1}) = \frac{\text{count}(t_{n-1}, t_n) + \lambda}{\text{count}(t_{n-1}) + \lambda \cdot T}$$

  where T is the number of distinct tags
- and for emission probabilities:

$$P(w|t) = \frac{\text{count}(w, t) + \lambda}{\text{count}(t) + \lambda \cdot V}$$

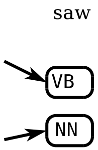
  where V is the number of distinct words
- usually there is some "special treatment" for the emission probability $P(w_n|t_n)$ if w_n is unseen in the training corpus
  - taking for instance punctuation, capitalization, numbers, suffixes into account

tagging

- how do we use our probability model to tag?
- conceptually: enumerate all possible tag sequences; use the probabilities to find the best one
- however in long sentences, the number of possible tag sequences is very large
- the **Viterbi algorithm** finds the most probable underlying tag sequence
  - Viterbi runs in **linear time** with respect to the length of the sentence

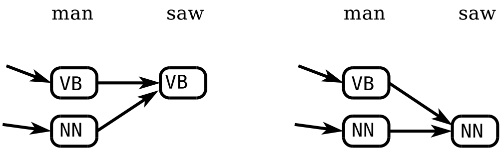
the Viterbi algorithm

- for each possible tag t_i of a word w_i, we compute **the best tag sequence leading to t_i**
  - for instance: for the word *saw*, we find the best sequence ending with *saw* as a verb, and the best ending with *saw* as a noun



the trick

- to compute the best path ending with *saw* as a verb, **consider the best paths for the previous word** and the **transition probabilities**
- assume the previous word is e.g. *man*, which can be a noun or a verb
- select the highest of
  - the LP of the best path ending in *man* as a verb + the LP of the transition verb → verb
  - the LP of the best path ending in *man* as a noun + the LP of the transition noun → verb



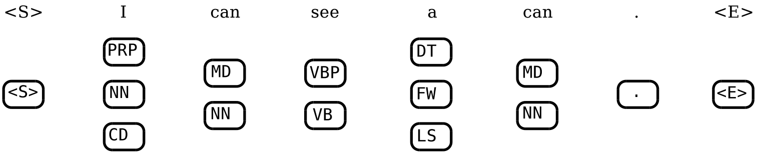
tagging a sentence

- apply the Viterbi algorithm step by step
- after the last token of the sentence, add a special dummy end token
  - this token will emit a dummy end tag with probability 1
- the best tag sequence for the whole sentence is the best path ending in the dummy tag
- finally, retrace your steps from the dummy item to get the tags
  - so you need backpointers

Viterbi example

<S> I can see a can . <E>

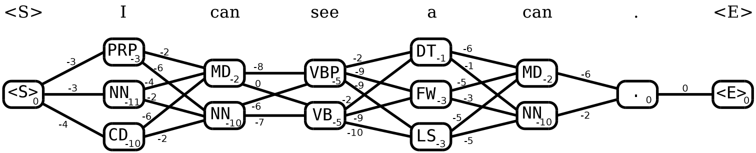
Viterbi example



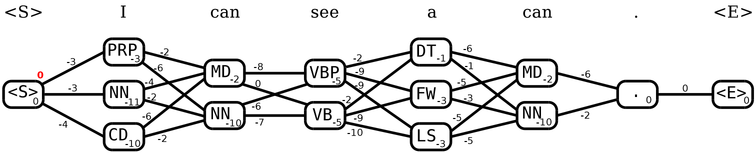
Viterbi example

<S>	I	can	see	a	can	.	<E>
	PRP$_{-3}$	MD$_{-2}$	VBP$_{-5}$	DT$_{-1}$	MD$_{-2}$		
<S>$_{0}$	NN$_{-11}$	NN$_{-10}$	VB$_{-5}$	FW$_{-3}$	NN$_{-10}$	.$_{0}$	<E>$_{0}$
	CD$_{-10}$			LS$_{-3}$			

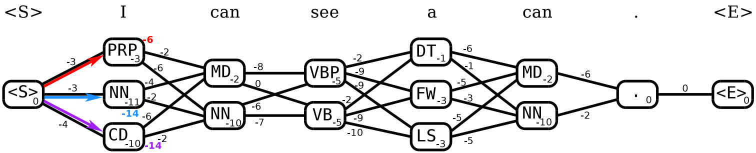
Viterbi example



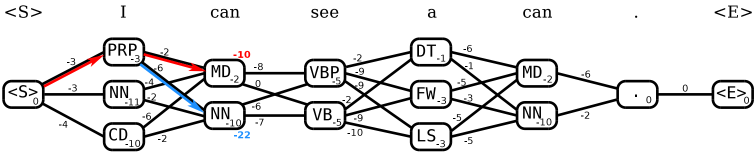
Viterbi example



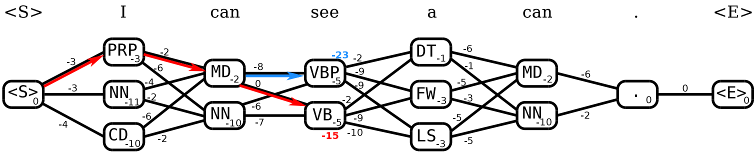
Viterbi example



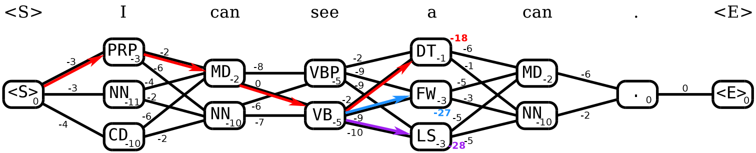
Viterbi example



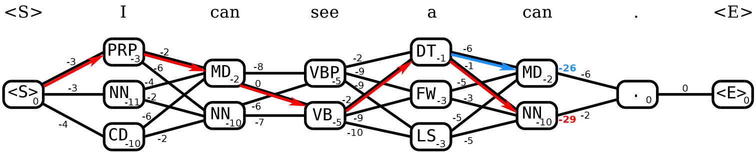
Viterbi example



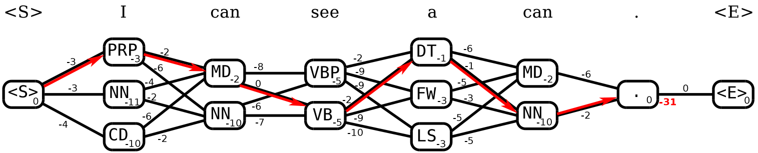
Viterbi example



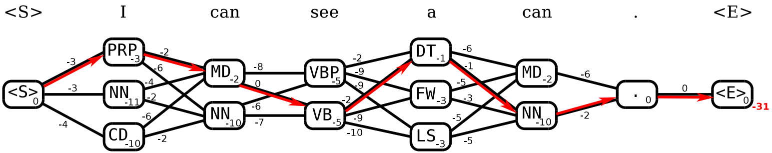
Viterbi example



Viterbi example



Viterbi example



using more context

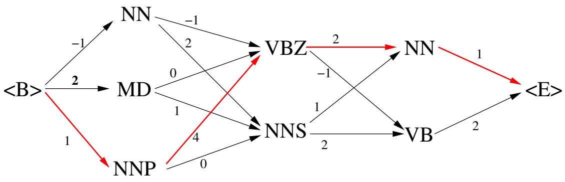
- tagging accuracy can possibly be improved by using more contextual information
- in a **trigram tagger**, we use transition probabilities such as

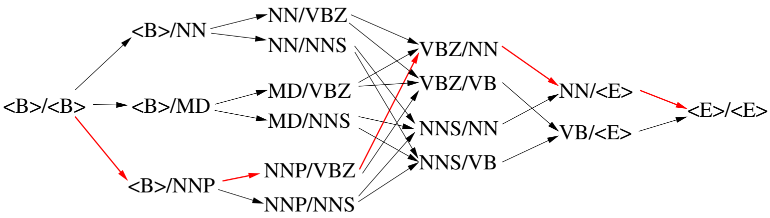
$$P(t_n | t_{n-1}, t_{n-2})$$

- smoothing becomes more important as you use more context

Search spaces...

- example: *Will plays golf*





overview

HMM tagging recap

assignment 3 overview and implementation hints

statistical recipes for tagger evaluation

the next few weeks

UNIVERSITY OF GOTHENBURG

assignment 3

- write a bigram part-of-speech tagger in Python
  - estimate the emission and transition probabilities
  - implement the Viterbi algorithm
  - evaluate the tagger on a test set
  - a little bit of error analysis
- the code template contains some comments that can be useful

some hints: estimation (1)

- you need to estimate the two types of probabilities:

$$P_{MLE}(\text{noun}|\text{verb}) = \frac{\text{count}(\text{verb, noun})}{\text{count}(\text{verb})} \qquad P_{MLE}(cat|\text{noun}) = \frac{\text{count}(\text{noun: } cat)}{\text{count}(\text{noun})}$$

- so you need to have data structures that count
  - ...occurrences of a tag (e.g. noun)
  - ...occurrences of a tag bigram (e.g. verb+noun)
  - ...occurrences of a word and tag (e.g. *cat*+noun)

some hints: estimation (2)

- it can be useful to use a "double dictionary" pattern

```
word_tag_counter[word][tag] += 1
```

- to get rid of key checks that clutter the code, you can use a `defaultdict(Counter)`
  - (recall: `Counter` is a frequency table)

some hints: estimation (3)

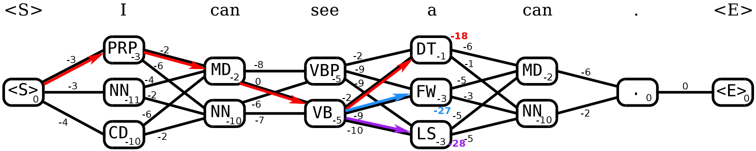
- the pseudocode says *determine the possible tags for this word*
- the most practical solution to this is to use all the tags that you observed for a word
  - and possibly more, if you have a tag lexicon
- if you store your emission probabilities using the double dictionary pattern, it's easy to find the allowed tags (and their probabilities) for a word

some hints: estimation (4)

- the code will be a bit simpler if you make sure that there are transition probabilities for all possible transitions
  - even those you haven't seen, so use smoothing!
- also, it's probably best if you use special tags for the start and end of the sentence

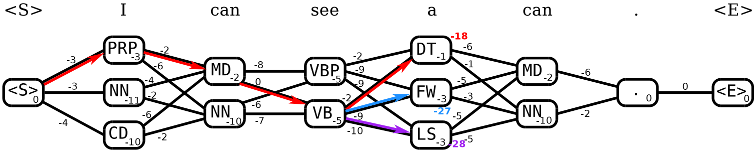
some hints: Viterbi (1)

- what do you think we should use to represent the **links** that we have drawn?



some hints: Viterbi (1)

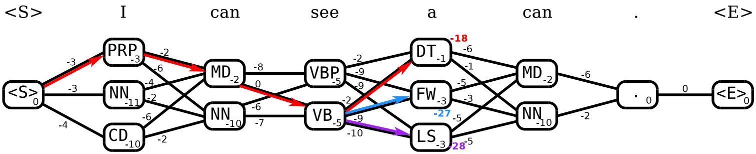
- what do you think we should use to represent the **links** that we have drawn?



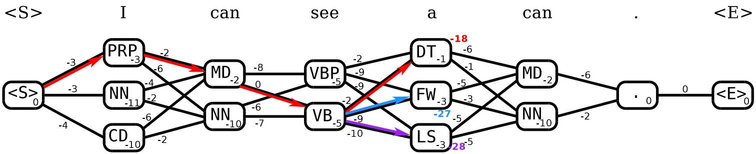
- we need something that remembers
  - the (log) **probability** of the path ending in that link
  - the **tag**
  - the **previous link**
- simplest solution is probably to use a **tuple** of these three

some hints: Viterbi (2)

- at each step of the algorithm, we keep a list of links
- there's one link for each possible tag at that step
- the link represents the best path leading up to that tag

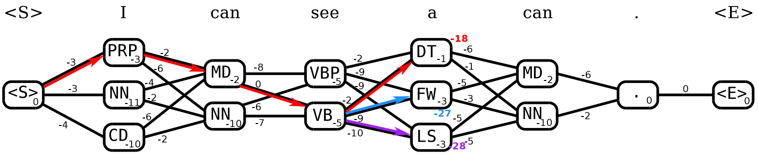


some hints: Viterbi (3)



- in the first step, the list contains a single dummy link
  - the log probability is 0 (100% chance of being here!)
  - special start tag (for instance `<S>`)
  - no previous link (use `None` or similar)

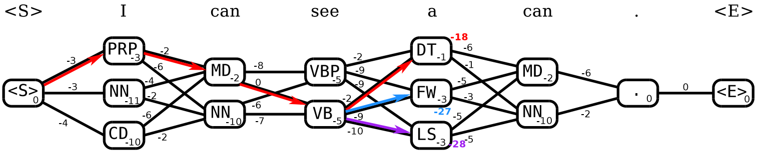
some hints: Viterbi (3)



- in the first step, the list contains a single dummy link
  - the log probability is 0 (100% chance of being here!)
  - special start tag (for instance <S>)
  - no previous link (use `None` or similar)
- let's assume we want to build the link for the `DT` tag
- find the previous link that maximizes the sum of
  - log probability in the previous link
  - log of transition probability to `DT`
  - log of emission probability of *a* for `DT`

UNIVERSITY OF GOTHENBURG

some hints: Viterbi (4)



- in the last step, we end the sequence with another dummy link
- then we need to follow the links backwards to find the tag sequence
  - (see pseudocode in the Python file)

some hints: unseen words

- you'll have to decide what to do in case you encounter a word that's not contained in your emission table
- a few possible solutions:
  - select the most frequent tag (e.g. noun)
  - allow all possible tags, but ignore the emission
  - allow all possible tags, and try to estimate the probability of emitting a rare word
  - use a Naive Bayes model to estimate the emission probability, based on some features such as capitalization, the presence of numbers, etc
  - categorize words (e.g. "capitalized", "number") and estimate special probabilities (see notes by Collins)

overview

HMM tagging recap

assignment 3 overview and implementation hints

statistical recipes for tagger evaluation

the next few weeks

evaluating a tagger

- we typically evaluate a tagger like this:

$$\text{tagging accuracy} = \frac{\text{number of correctly tagged words}}{\text{number of words in the corpus}}$$

example of how this can be done in Python

- goldstandard and guess are lists of tagged sentences
  - and each sentence is a list of word–tag pairs
- note the following Python tricks:
  - zip takes two lists, and returns a list of pairs
  - sum on a list of booleans counts the number of True

```
def evaluate_tagger(goldstandard, guess):
    n_tokens = 0
    n_correct = 0
    for sen_gold, sen_guess in zip(goldstandard, guess):
        n_correct += sum(tt == gt
                         for tt, gt in zip(sen_gold, sen_guess))
        n_tokens += len(sen_gold)
    return n_correct / n_tokens
```

statistical recipes we've seen so far

- for the classifiers, we considered three scenarios:
  - how to make a confidence interval for the accuracy?
  - how to compare the the accuracy to some given value?
  - how to compare two classifiers?
- we saw recipes to address these questions, and they were built on the binomial distribution
- **but**: it's not statistically sound to use these techniques to evaluate a tagger

statistical recipes when the output is complex

- we'll now see how these statistical questions can be addressed when evaluating a tagger
- actually, the techniques are general and can be used when evaluating other kinds of applications, including
  - bracket precision and recall for a phrase-structure parser
  - attachment accuracy for a dependency parser
  - BLEU for machine translation
  - word error rate for speech recognition

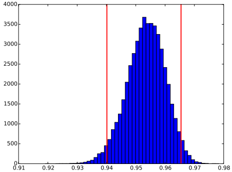
bootstrap resampling

- the variation in our estimate depends on the distribution of possible test sets
- in theory, we could find a confidence interval by considering the distribution of all possible test sets, but this can't be done in practice
- the trick in **bootstrapping** – invented by Bradley Efron – is to assume that we can simulate the distribution of possible test sets **by picking randomly from the original test set**

bootstrapping a confidence interval, pseudocode

- we have a test set T consisting of k sentences
- we compute a confidence interval by generating N random test sets and finding the interval where most estimates end up

repeat N times
  $T^* =$ pick k sentences randomly from T
  $a =$ estimated accuracy of the tagger on T^*
  store a in a list A
return 2.5% and 97.5% percentiles of A



bootstrapping a confidence interval in Python (part 1)

```
def bootstrap_ci(goldstandard, guess, N):

  # for instance [ (8,9), (6,6), (12,14), ... ]
  evals = [ evaluate_sentence(sen_gold, sen_guess)
            for sen_gold, sen_guess in zip(goldstandard, guess) ]

  # for instance [ 0.87, 0.92, 0.88, 0.89, ...]
  A = [ random_testset_accuracy(evals) for _ in range(N) ]

  lower = scipy.percentile(A, 2.5)
  upper = scipy.percentile(A, 97.5)

  return lower, upper
```

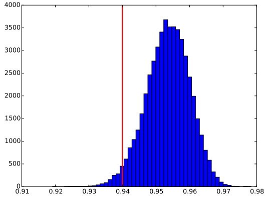
bootstrapping a confidence interval in Python (part 2)

```
def evaluate_sentence(sen_gold, sen_guess):
    n_correct = sum(t1 == t2
                    for t1, t2 in zip(sen_gold, sen_guess))
    return n_correct, len(sen_gold)


def random_testset_accuracy(evals):
    # evals is for instance [ (8,9), (6,6), (12,14), ... ]

    n_words = 0
    n_correct = 0
    for _ in range(len(evals)):
        sentence_eval = random.choice(evals)
        n_correct += sentence_eval[0]
        n_words += sentence_eval[1]
    return n_correct / n_words
```

bootstrapping for comparing to a fixed value

- is the tagging accuracy significantly greater than 0.94?
- we compute the p-value by checking how often the accuracy falls below 0.94

repeat N times
    $T^* =$ pick k sentences randomly from T
    $a =$ estimated accuracy of the tagger on T^*
    if $a < 0.94$
        increase counter C
return C/N



bootstrapping for comparing two taggers

- is tagger A significantly better than tagger B?
- we use a lot of randomly generated test sets, and compute the p-value by checking how often tagger B outperforms tagger A

repeat N times
  $T^* =$ pick k sentences randomly from T
  $a_A =$ estimated accuracy of tagger A on T^*
  $a_B =$ estimated accuracy of tagger B on T^*
  if $a_B > a_A$
    increase counter C
return C/N

overview

HMM tagging recap

assignment 3 overview and implementation hints

statistical recipes for tagger evaluation

the next few weeks

the computer assignments

- assignment 3: tagger implementation
- February 25 and March 1
- report deadline: March 10

next lectures

- March 3: unsupervised and semisupervised methods
- March 10: machine translation (with Prasanth)
- March 15 and 17: VG assignment lab sessions (and catchup)