2nd year project - Lecture 5

Traditional language models and sequence labeling with Hidden Markov models

Until now

- RegEx, command line tools
- Experimental setup, annotation
- Tokenization, POS tagging, classification

Today's overview

- Language Models (predicting the next word with traditional n-gram LMs)
- Sequence labeling with Hidden Markov Models

Learning outcomes

- What is a language model?
- How does an n-gram language work?
- What are strengths/weaknesses of n-gram language models?
- How to evaluate a language model?
- How to use Viterbi to do sequence labeling?

Part I: Predicting the next word: A Simple (?) Exercise

Please turn your homework ___
(Example from J&M chapter 3)

Part I: Predicting the next word: A Simple (?) Exercise

Please turn your homework ___
(Example from J&M chapter 3) "in" or "over" are possible words, but probably not "refigerator" or "the"

Let's play a game

Go to menti.com, code: 70056703 TODO

Have you heard about ChatGPT?

RO

Hi assistant. Why is NLP the most interesting field?

NLP (Natural Language Processing) is considered one of the most interesting fields in artificial intelligence and computer science as it deals with helping computers understand, interpret, and generate human language. This field has a wide range of real-world applications, from language translation to sentiment analysis, and text-to-speech synthesis, making it both challenging and impactful. Additionally, since human language is complex and constantly evolving, NLP presents an ongoing opportunity for researchers and developers to advance the state-of-the-art and improve its effectiveness.

What is a Language Model (LM)?

Goal: to compute the probability of a sentence:

$$P(\text{today is a great day}) = ??$$

Related task: probability of an upcoming word:

$$P(\text{day}|\text{today is a great}) = ??$$

A model that computes either of these is called a **Language Model** (LM).

An LM is a generative statistical model of language. It can be used to - rank sentences:

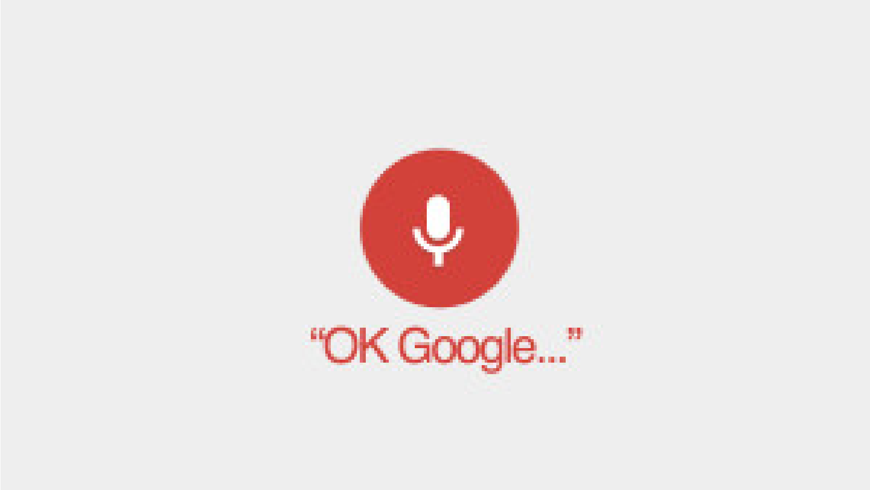
$$P(\text{today is a great day}) = 0.01$$

$$P(\text{today is an overspecific day}) = 0.0000000001$$

- calculate the probability of the next word, e.g., to rank word choices:

$$P(\text{day}|\text{great}) > P(\text{day}|\text{blue})$$

Uses cases: Speech Recognition



$$P(\text{where is the nearest beach}) > P(\text{where is the nearest breach})$$

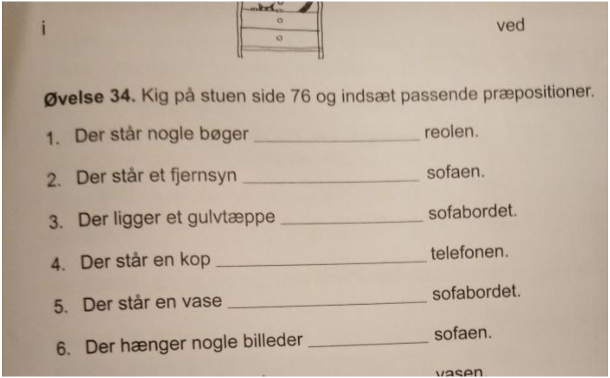
Uses cases: Spelling correction

Use Cases: Machine Translation

$>$ Wir sind guter Hoffnung
translates to:
$>$ We are good hope
or
$>$ We are in good hope

Can you think of more use cases?

Make Danish homework:



i ved

Øvelse 34. Kig på stuen side 76 og indsæt passende præpositioner.

1. Der står nogle bøger ______________ reolen.
2. Der står et fjernsyn ______________ sofaen.
3. Der ligger et gulvtæppe ______________ sofabordet.
4. Der står en kop ______________ telefonen.
5. Der står en vase ______________ sofabordet.
6. Der hænger nogle billeder ______________ sofaen.

vasen

- Does it show we understand (the) language?

- Relatively new: language models for multitask learning!

Formally

A language model (LM) models the probability

$$P(w_1, \dots, w_d)$$

of observing sequences of words

$$(w_1, \dots, w_d).$$

Without loss of generality (Chain rule):

$$P(w_1, \ldots, w_d) = P(w_1)P(w_2|w_1)P(w_3|w_1, w_2)\ldots$$

$$= P(w_1)\prod_{i=2}^{d} P(w_i|w_1, \ldots, w_{i-1})$$

Without loss of generality (Chain rule):

$$P(w_1, \ldots, w_d) = P(w_1)P(w_2|w_1)P(w_3|w_1, w_2) \ldots$$

$$= P(w_1) \prod_{i=2}^{d} P(w_i|w_1, \ldots, w_{i-1})$$

Example:

$$P(\text{the nice house}) = P(\text{the}) * P(\text{nice}|\text{the}) * P(\text{house}|\text{the nice})$$

Without loss of generality (Chain rule):

$$P(w_1, \ldots, w_d) = P(w_1)P(w_2|w_1)P(w_3|w_1, w_2) \ldots$$

$$= P(w_1) \prod_{i=2}^{d} P(w_i|w_1, \ldots, w_{i-1})$$

Example:

$$P(\text{the nice house}) = P(\text{the}) * P(\text{nice}|\text{the}) * P(\text{house}|\text{the nice})$$

or:

$$P(\text{the nice house}) = P(\text{the}| < S >) * P(\text{nice}|\text{the}) * P(\text{house}|\text{the nice})$$

So each time we compute a related task: $P(w|h)$, the probability of a word w given some history h (for now, all words up to w_{d-1}).

However, it is impossible to estimate a sensible probability for each history

$$\mathbf{x} = w_1, \ldots, w_{i-1}$$

Possible solutions:

- n-gram based LMs (count-based)
- neural LMs (lecture 7, 11, and 12)

Method 1: Count-based Language Models (traditional)

Where do we get the estimates for the model parameters from?

Method 1: Count-based Language Models (traditional)

Where do we get the estimates for the model parameters from?
We typically use maximum likelihood estimates (counts!):

$$P(\text{house}|\text{the nice}) = \frac{C(\text{the nice house})}{C(\text{the nice})} \tag{1}$$

C = count

Do you see a problem?

Do you see a problem?

Another example: Could we just count and divide?

$$P(\text{the}|\text{its water is so transparent that})$$
$$= \frac{C(\text{its water is so transparent that the})}{C(\text{its water is so transparent that})}$$

C = count

The (crucial!) Markov assumption

$$P(\text{started}|\text{The meeting of the government last Tuesday has})$$

The **idea of the n-gram**: instead of considering the entire history, we can **approximate** the history by the last few words (Markov assumption)

$$P(\text{started}|\text{Tuesday has})$$

Change representation

truncate history to last $n-1$ words:

$$\mathbf{f}(\mathbf{x}) = w_{i-(n-1)}, \ldots, w_{i-1}$$

$$P(\text{bigly}|\text{...,blah, blah, blah, we, will, win}) = P(\text{bigly}|\text{we, will, win})$$

Change representation

truncate history to last $n-1$ words:

$$\mathbf{f}(\mathbf{x}) = w_{i-(n-1)}, \dots, w_{i-1}$$

$$P(\text{bigly}|\text{...,blah, blah, blah, we, will, win}) = P(\text{bigly}|\text{we, will, win})$$

We make a Markov assumption of conditional independence:

$$P(x_1, ..., x_d) = \prod_{i=1}^{d} P(x_i|x_1, ..., x_{i-1}) \approx \prod_{i=1}^{d} P(x_i|x_{i-(n-1)}, ..., x_{i-1})$$

Types of n-gram models (based on the n-th order)

Unigram LM Set $n = 1$:

$$P(w_i | w_1, \ldots, w_{i-1}) = P(w_i).$$

$P(\text{bigly}|\text{we, will, win}) = P(\text{bigly})$

Higher-order LMs

A **bigram** LM conditions only on previous word (window or history of $n = 2$ words):

$$P(x_i | x_{i-1})$$

Trigram model uses a history of two words (window of $n = 3$ words), etc:

$$P(x_i | x_{i-2}, x_{i-1})$$

Baseline: Uniform LM

Same probability for each word in a *vocabulary*
(**vocab**
):

$$P(w_i|w_1,\ldots,w_{i-1}) = \frac{1}{|\textbf{vocab}|}.$$

$P(\text{big}) = P(\text{bigly}) = \frac{1}{|\textbf{vocab}|}$

Example: Estimating bigram probabilities from a corpus

$$P(w_i \mid w_{i-1}) = \frac{c(w_{i-1}, w_i)}{c(w_{i-1})}$$

<s> I am Sam </s>
<s> Sam I am </s>
<s> I do not like green eggs and ham </s>

$P(\texttt{I} \mid \texttt{<s>}) = \frac{2}{3} = .67$ $\quad P(\texttt{Sam} \mid \texttt{<s>}) = \frac{1}{3} = .33$ $\quad P(\texttt{am} \mid \texttt{I}) = \frac{2}{3} = .67$

$P(\texttt{</s>} \mid \texttt{Sam}) = \frac{1}{2} = 0.5$ $\quad P(\texttt{Sam} \mid \texttt{am}) = \frac{1}{2} = .5$ $\quad P(\texttt{do} \mid \texttt{I}) = \frac{1}{3} = .33$

Sampling

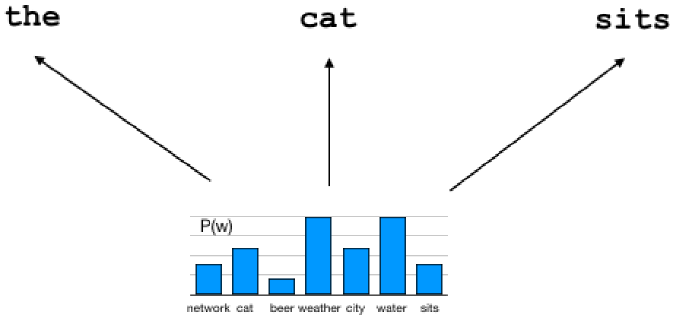
- Sampling from an LM is easy and instructive
- Usually, the better the LM, the better the samples

(*the following slides are adapted from S.Riedel*).

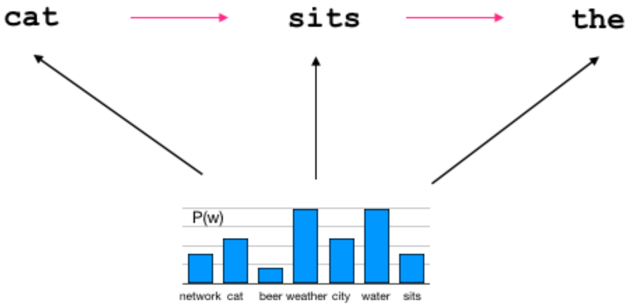
Sample **incrementally**, one word at a time

Sample from a unigram LM

- We can sample **incrementally** from a Language Model, one word at a time



word salad? *sequences / sequential data!*



Evaluation

- **Extrinsic**: how it improves a downstream task? (e.g., Machine translation, speech recognition..)
- **Intrinsic**: how good does it model language? (i.e., how good is the language model by itself?)

Intrinsic Evaluation

Predict next word, win if prediction match words in actual corpus (or you gave it high probability):

- Our horrible trade agreements with [???]
- Why don't we use accuracy?

Perplexity

Given test sequence $(w_1, \ldots, w_N)$ perplexity is the **inverse probability of the test set, normalized by the number of words**:

$$perplexity(w_1, \ldots, w_N) = P(w_1, \ldots, w_N)^{-\frac{1}{N}} = \sqrt[N]{\frac{1}{P(w_1, \ldots, w_N)}}$$

$$= \sqrt[N]{\prod_i^N \frac{1}{P(w_i|w_1, \ldots, w_{i-1})}}$$

For a bigram LM:

$$perplexity(w_1, \ldots, w_N) = \sqrt[N]{\prod_i^N \frac{1}{P(w_i|w_{i-1})}}$$

Perplexity

Given test sequence $(w_1, \ldots, w_N)$ perplexity is the **inverse probability of the test set, normalized by the number of words**:

$$perplexity(w_1, \ldots, w_N) = P(w_1, \ldots, w_N)^{-\frac{1}{N}} = \sqrt[N]{\frac{1}{P(w_1, \ldots, w_N)}}$$

$$= \sqrt[N]{\prod_i^N \frac{1}{P(w_i | w_1, \ldots, w_{i-1})}}$$

For a bigram LM:

$$perplexity(w_1, \ldots, w_N) = \sqrt[N]{\prod_i^N \frac{1}{P(w_i | w_{i-1})}}$$

Range: $1 - \infty$ (lower is better!)

- Training 38 million words, test 1.5 million words, WSJ

N-gram Order	Unigram	Bigram	Trigram
Perplexity	962	170	109

- Training 38 million words, test 1.5 million words, WSJ

N-gram Order	Unigram	Bigram	Trigram
Perplexity	962	170	109

- Why not 4-grams?

- Training 38 million words, test 1.5 million words, WSJ

N-gram Order	Unigram	Bigram	Trigram
Perplexity	962	170	109

- Why not 4-grams?
- Why not 20-grams?

20-grams:

- Overfitting: train perplexity very low, dev very high
- When generating, will look like fluent text
  - Mostly repeats the training data!

Interpretation

Consider a LM:

- For which at each position there are exactly **2** words with $\frac{1}{2}$ probability
- What's the perplexity of a text under this LM?

Then

- $\mathit{perplexity}(w_1, \ldots, w_T) = \sqrt[T]{2 \cdot 2 \cdot \ldots} = 2$
- Perplexity $\approx$ average number of choices

But what if a word does not occur in the ranking?

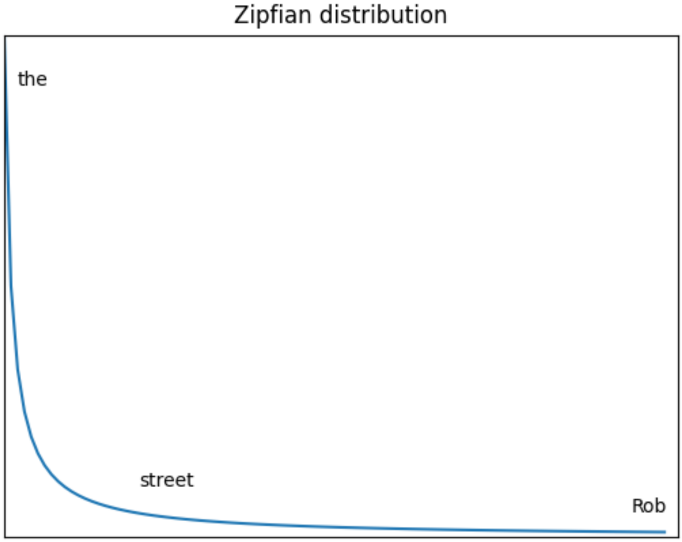
- Perplexity is undefined!
- Happens with every word not seen during training

Problem: model assigns **zero probability** to words not in the vocabulary.

The Long Tail

New words not specific to our corpus:

- long **tail** of words that appear only a few times* each individual one has low probability, but probability of seeing any long tail word is high


Zipfian distribution
the
street
Rob

Example: Estimating a bigram LM from the Berkeley restaurant corpus

- **Out of 9222 sentences**

	i	want	to	eat	chinese	food	lunch	spend
i	5	827	0	9	0	0	0	2
want	2	0	608	1	6	6	5	1
to	2	0	4	686	2	0	6	211
eat	0	0	2	0	16	2	42	0
chinese	1	0	0	0	0	82	1	0
food	15	0	15	0	1	4	0	0
lunch	2	0	0	0	0	1	0	0
spend	1	0	1	0	0	0	0	0

(slides from J&M)

- Normalize by unigrams:

i	want	to	eat	chinese	food	lunch	spend
2533	927	2417	746	158	1093	341	278

- Result:

	i	want	to	eat	chinese	food	lunch	spend
i	0.002	0.33	0	0.0036	0	0	0	0.00079
want	0.0022	0	0.66	0.0011	0.0065	0.0065	0.0054	0.0011
to	0.00083	0	0.0017	0.28	0.00083	0	0.0025	0.087
eat	0	0	0.0027	0	0.021	0.0027	0.056	0
chinese	0.0063	0	0	0	0	0.52	0.0063	0
food	0.014	0	0.014	0	0.00092	0.0037	0	0
lunch	0.0059	0	0	0	0	0.0029	0	0
spend	0.0036	0	0.0036	0	0	0	0	0

Out-of-Vocabulary (OOV) Tokens

There will always be words with zero counts in your training set.
Why is this a problem?

Out-of-Vocabulary (OOV) Tokens

There will always be words with zero counts in your training set. Why is this a problem?

- Perplexity is based on inverse probability of test set
- Since we cannot divide by 0, we cannot compute perplexity at all at this point
- If probability of a word in the test set is 0, the entire probability of the test set is 0
- Underestimating probability of unseen words
- Downstream application performance suffers

Out-of-Vocabulary (OOV) Tokens

There will always be words with zero counts in your training set.
Solutions:

- —Remove unseen words from test set (bad)—
- Replace unseen words with out-of-vocabulary token, estimate its probability
- Move probability mass to unseen words

Replace unseen words with out-of-vocabulary token, estimate its probability

- Mark unknown words at test time as OOV
  - Replace all words that appear fewer than n times with OOV token
  - Choose a vocabulary in advance, then mark all words not in that set as OOV
  - Can create categories of OOV's. For example OOV-ing OOV-capFirst etc.

Out-of-Vocabulary (OOV) Tokens

There will always be words with zero counts in your training set.
Solutions:

- —Remove unseen words from test set (bad)—
- Replace unseen words with out-of-vocabularly token, estimate its probability
- **Move probability mass to unseen words**

Move probability mass to unseen words: Smoothing

Maximum likelihood

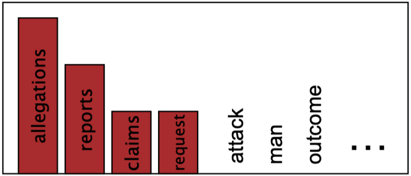
- **underestimates** true probability of some words
- **overestimates** the probabilities of other words
- Solution: *smooth* the probabilities and **move mass** from seen to unseen events.

The intuition of smoothing (from Dan Klein)

When we have sparse statistics:

P(w | denied the)
- 3 allegations
- 2 reports
- 1 claims
- 1 request

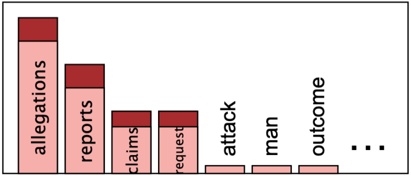
7 total



Steal probability mass to generalize better

P(w | denied the)
- 2.5 allegations
- 1.5 reports
- 0.5 claims
- 0.5 request
- 2 other

7 total



JM

Laplace Smoothing / Additive Smoothing

Add **pseudo counts** to each event in the dataset Pretend we saw each word one more time than we did. Add-1 estimate (Laplace):

$$P_{MLE}(w_i|w_{i-1}) = \frac{C(w_{i-1}, w_i)}{C(w_{i-1})}$$

$$P_{Add1}(w_i|w_{i-1}) = \frac{C(w_{i-1}, w_i) + 1}{C(w_{i-1}) + V}$$

Interpolation

- Laplace Smoothing assigns mass **uniformly** to the words that haven't been seen in a context.

Not all unseen n-grams (in a context) are equal

With **interpolation** we can do better:

- give more mass to words likely under the $n-1$-gram model.
- Use $P(\text{of})$ for estimating $P(\text{of}|\text{man})$

With **interpolation** we can do better:

- give more mass to words likely under the $n-1$-gram model.
- Use $P(\text{of})$ for estimating $P(\text{of}|\text{man})$
- Combine n-gram model and a back-off (n-1) model:

$$P_\alpha(w_i|w_{i-n+1},\ldots,w_{i-1}) = \alpha \cdot P'(w_i|w_{i-n+1},\ldots,w_{i-1}) + (1-\alpha)\cdot$$

$$P''(w_i|w_{i-n+2},\ldots,w_{i-1})$$

Can we find a good α parameter? Tune on some **development set**!

Limitations of n-gram LMs

- What about similar words?
  - she *bought* a bicycle
  - she *purchased* a bicycle

cannot share strength
among similar words

- Long-distance dependencies?
  - for *programming* she yesterday purchased her own brand new *laptop*
  - for *running* she yesterday purchased her brand new *sportswatch*

cannot handle long-distance dependencies

Summary

- LMs model probability of sequences of words
- Defined in terms of ”next-word” distributions conditioned on history
- N-gram models truncate history representation
- Often trained by maximising log-likelihood of training data and ...
- smoothing to deal with sparsity

Sequence labeling

Named Entity Recognition

```
| Barack | Obama | was |  born | in | Hawaii |
| B-PER  | I-PER | O   |  O    | O  | B-LOC  |
```

Part-Of-Speech Tagging

Assign each token in a sentence its **part-of-speech tag**.

```
I       PRON
see     VERB
the     DET
light   NOUN
!       PUNCT
```

Sequence Labelling

- Input Space X_s: sequences of items to label
- Output Space Y_s: sequences of output labels
- Model: $s_{params}(x, y)$
- Prediction: $argmax_y s_{params}(x, y)$
- Is a particular type of structured prediction problem

Markov Chains

- **Dependencies** between consecutive labels

Formally:

- Set of n states

$Q = q_1 q_2 ... q_N$

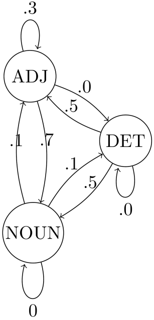
- Transition probability matrix

$A = a_{11} a_{12} ... a_{NN}$

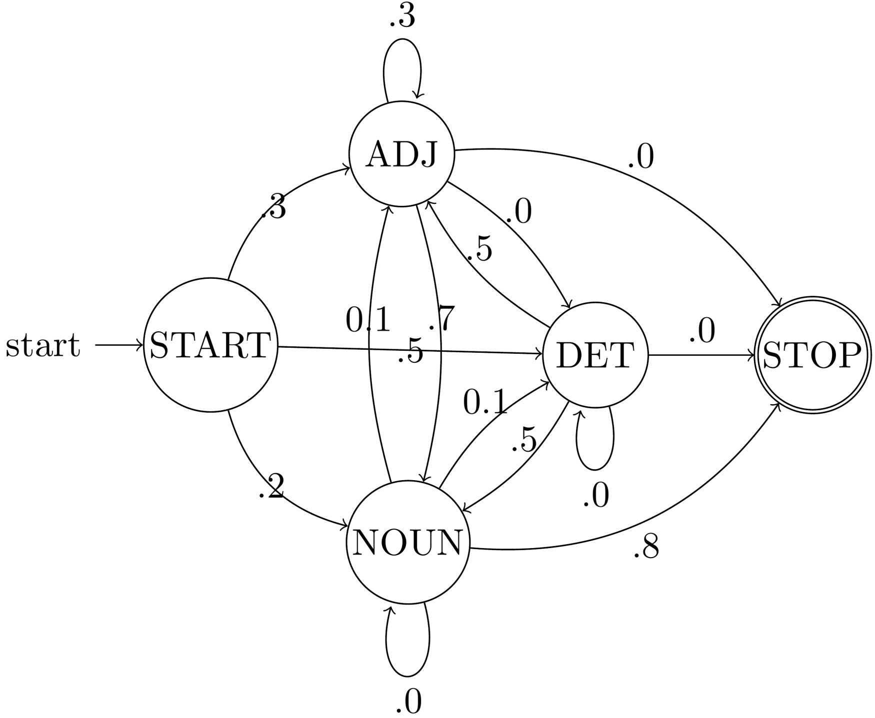
- Initial probability distribution

$\pi = \pi_1 \pi_2 ... \pi_N$

Consider the following description (Markov Chain) of a language



containing only noun phrases (NP).


.3
ADJ
.0
.3
.0
.5
start
START
0.1
.7
.5
DET
.0
STOP
0.1
.5
.2
.0
NOUN
.8
.0

Can be seen as a matrix; all nodes can be connected (or not):

	outgoing				
	START	ADJ	DET	NOUN	STOP
START	.0	.0	.0	.0	.0
ADJ	.3	.3	.5	.1	.0
DET	.5	.0	.0	.1	.0
NOUN	.2	.7	.5	.0	.0
STOP	.0	.0	.0	.8	.0

Markov assumption, the past doesn't matter!:

$$P(q_i = a|q_1 ... q_{i-1}) = P(q_i = a|q_{i-1}) \tag{2}$$

Markov assumption, the past doesn't matter!:

$$P(q_i = a | q_1 ... q_{i-1}) = P(q_i = a | q_{i-1}) \tag{2}$$

Where do we get these probabilities from?

- We need an annotated corpus, a corpus in which POS tags were annotated (by humans)

```
I      PRON
see    VERB
the    DET
light  NOUN
!      PUNCT
```

Where do we get these probabilities from?

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

$$P(t_n|t_{n-1}) = \frac{count(t_{n-1}, t_n)}{count(t_{n-1})}$$

Where do we get these probabilities from?

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

$$P(t_n|t_{n-1}) = \frac{count(t_{n-1}, t_n)}{count(t_{n-1})}$$

Example

$$P(NOUN|ADJ) = \frac{count(ADJ, NOUN)}{count(ADJ)}$$

Hidden Markov Models:

Extension of Markov chains:

- labels are hidden states (can not directly be observed)
- **generative model**

- Sequence of T observations from vocabulary

$O = o_1 o_2 ... o_T$

- Sequence of observations likelihoods (emission probabilities)

$B = b_i(o_t)$

Hidden Markov Model (HMM) tagging as decoding

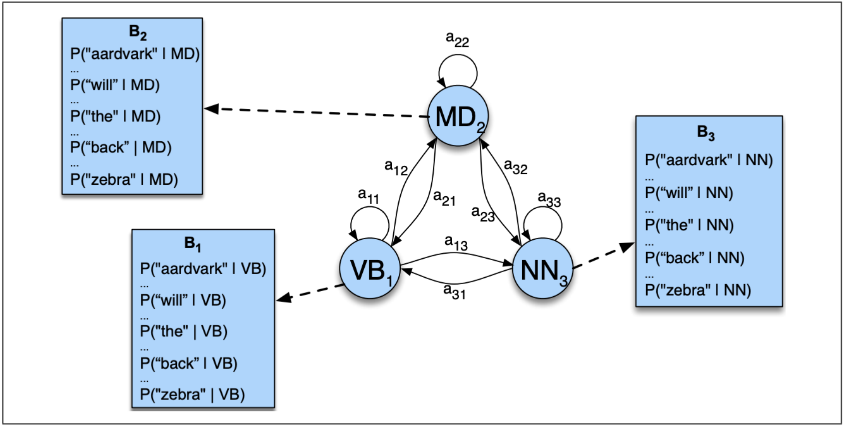


Figure 8.4 An illustration of the two parts of an HMM representation: the *A* transition probabilities used to compute the prior probability, and the *B* observation likelihoods that are associated with each state, one likelihood for each possible observation word.

Hidden Markov Model (HMM) tagging

For part of speech tagging, the goal of HMM decoding is to choose the tag sequence t_1^n that is most probable given the observation sequence of n words words w_1^n:

$$\hat{t}_1^n = \underset{t_1^n}{\operatorname{argmax}} P(t_1^n | w_1^n) \tag{8.13}$$

The way we'll do this in the HMM is to use Bayes' rule to instead compute:

$$\hat{t}_1^n = \underset{t_1^n}{\operatorname{argmax}} \frac{P(w_1^n | t_1^n) P(t_1^n)}{P(w_1^n)} \tag{8.14}$$

Furthermore, we simplify Eq. 8.14 by dropping the denominator $P(w_1^n)$:

$$\hat{t}_1^n = \underset{t_1^n}{\operatorname{argmax}} P(w_1^n | t_1^n) P(t_1^n) \tag{8.15}$$

(Taken from J&M)

Hidden Markov Model (HMM) tagging as decoding

$$\hat{t}_1^n = \underset{t_1^n}{\operatorname{argmax}} P(t_1^n|w_1^n) \approx \underset{t_1^n}{\operatorname{argmax}} \prod_{i=1}^{n} \overbrace{P(w_i|t_i)}^{\text{emission}} \overbrace{P(t_i|t_{i-1})}^{\text{transition}}$$

Hidden Markov Model (HMM) tagging as decoding

Two assumptions:

- The first is that the probability of a word appearing depends only on its own tag and is independent of neighboring words and tags:

$$P(w_1, .., w_n | t_1, .., t_n) \approx \prod_{i=1}^{n} P(w_i | t_i)$$

- The second assumption, the bigram assumption, is that the probability of the tag is dependent only on the previous tag, rather than the entire tag sequence:

$$P(t_1, .., t_n) \approx P(t_i | t_{i-1})$$

Beware!, the probability of a word given a tag (even though the word is known):

$$P(w_n|t_n) = \frac{count(w, t_n)}{count(t_n)}$$

$$P(cat|NOUN) = \frac{count(cat, NOUN)}{count(NOUN)}$$

It is a generative model!

How do we use our probability model to tag?

Naive decoding approach 2

What's the simplest approach to assign a tag to every word?

Naive decoding approach 2

What's the simplest approach to assign a tag to every word?

```
def tag(word_seq):
    return ['noun' for _ in word_seq]
```

Naive decoding approach 2

What's the simplest approach to assign a tag to every word?

```
def tag(word_seq):
    return ['noun' for _ in word_seq]
```

More competetive: naively predict the most likely tag for every word in the sequence. For every word:

$$argmax_t P(w|t)$$

=**Most Frequent Class Baseline:** It is good practice to compare a classifier against a baseline at least as good as the most frequent class baseline (assigning each token to the class it occurred in most often in the training set).

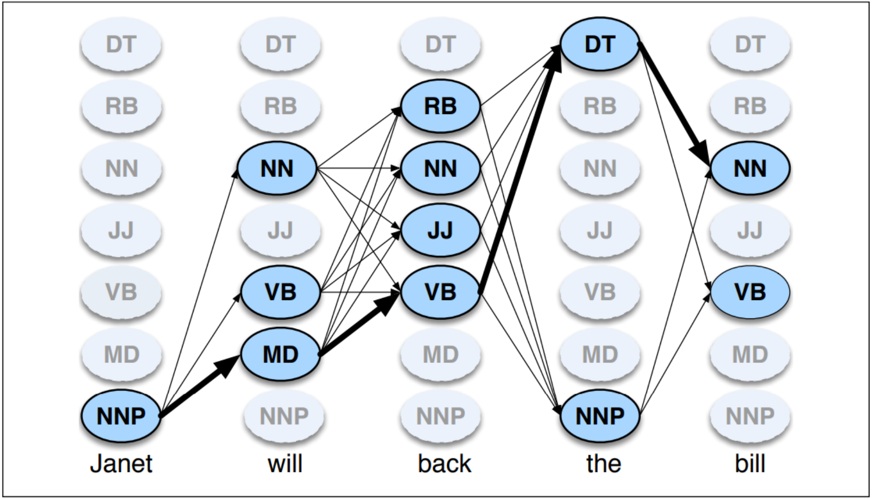
A better approach?

conceptually: enumerate all possible tag sequences; use the probabilities to find the *most likely path*

Viterbi Algorithm

Dynamic algorithm using a **trellis** with **backpointers**

- the previous approach becomes soon intractable: in long sentences, the number of possible tag sequences is very large
- the **Viterbi algorithm** finds the most probable underlying tagsequence
- because of the Markov assumption, this algorithm is very efficient



(taken from J&M)

function VITERBI(*observations* of len T,*state-graph* of len N) **returns** *best-path*, *path-prob*

create a path probability matrix *viterbi[N,T]*
for each state s **from** 1 **to** N **do** ; initialization step
$viterbi[s,1] \leftarrow \pi_s * b_s(o_1)$
$backpointer[s,1] \leftarrow 0$
for each time step t **from** 2 **to** T **do** ; recursion step
for each state s **from** 1 **to** N **do**
$$viterbi[s,t] \leftarrow \max_{s'=1}^{N} viterbi[s',t-1] * a_{s',s} * b_s(o_t)$$
$$backpointer[s,t] \leftarrow \operatorname{argmax}_{s'=1}^{N} viterbi[s',t-1] * a_{s',s} * b_s(o_t)$$
$$bestpathprob \leftarrow \max_{s=1}^{N} viterbi[s,T]$$; termination step
$$bestpathpointer \leftarrow \operatorname{argmax}_{s=1}^{N} viterbi[s,T]$$; termination step
bestpath ← the path starting at state *bestpathpointer*, that follows backpointer[] to states back in time
return *bestpath*, *bestpathprob*

Figure 8.10 Viterbi algorithm for finding the optimal sequence of tags. Given an observation sequence and an HMM $\lambda = (A, B)$, the algorithm returns the state path through the HMM that assigns maximum likelihood to the observation sequence.

(taken from J&M)

In the following example, we will perform the task of word-level language identification in code-switched data. The hidden states are ‘EN‘, ‘ES‘, ‘punct‘, and the input is tokenized text:

```
It’s | payday | ! | tires | tu | dinero | ! |
EN   | EN     | P | ES    | ES | ES     | P |
```

This example is taken from: Iliescu, Grand, Qirko and van der Goot (2021). They do unsupervised language prediction on the word-level!

	EN	ES	P	$< /S >$
$< S >$	0.5	0.5	0.0	0.0
EN	0.8	0.1	0.1	0.0
ES	0.1	0.7	0.2	0.0
P	0.1	0.2	0.0	0.7

- Note that sum of rows is 1.0 (the change of having an outgoing arc is 1.0)
- Note that $< S >$ and $< /S >$ are only possible as start/end, so the missing row/column is filled with 0.0's in practice

Lets use negative log probabilities:

- More precision
- More efficient

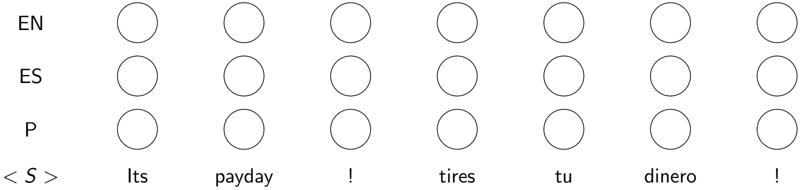
Lets use negative log probabilities:

- More precision
- More efficient

```
import math
val1 = .5
val2 = .3
print(val1 * val2)
logProb = -math.log(val1) + -math.log(val2)
print(logProb)
print(math.exp(-logProb))
```

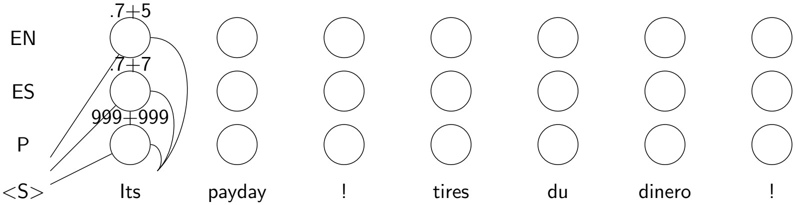
	EN	ES	P	$</S>$
$<S>$	0.7	0.7	999	999
EN	0.22	2.3	2.3	999
ES	2.3	0.36	1.6	999
P	2.3	1.6	999	.36

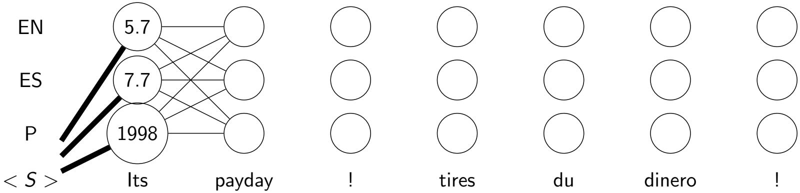
- note that 999 is used as a very low probability (because 0.0 does not exist in log())


EN
ES
P
$< S >$
Its
payday
!
tires
tu
dinero
!

$p(\text{EN}|<S>) = .7$ $p(\text{ES}|<S>) = .7$ $p(\text{P}|<S>) = 999$

$p(\text{its}|\text{EN}) = 5$ $p(\text{its}|\text{ES}) = 7$ $p(\text{its}|\text{P}) = 999$

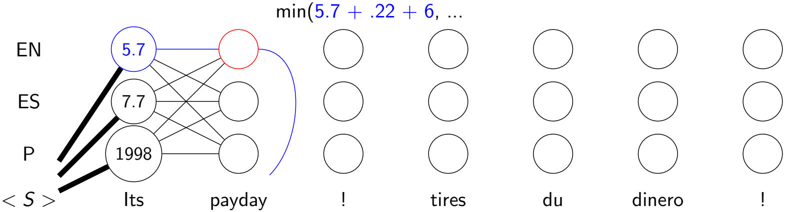



EN
ES
P
$< S >$
5.7
7.7
1998
Its
payday
!
tires
du
dinero
!

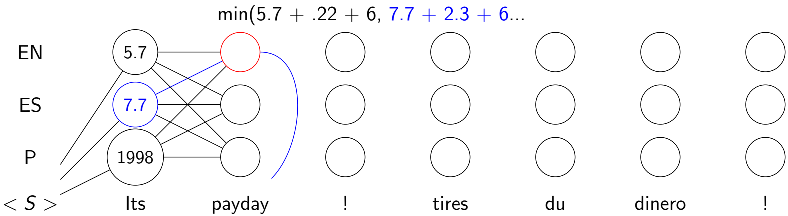
$p(t-1 = EN) = 5.7$

$p(EN|EN) = .22$

$p(payday|EN) = 6$

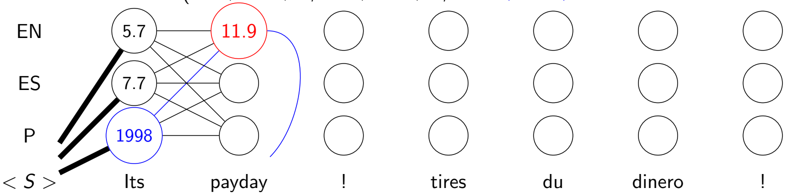


$$p(t-1=ES)=7.7$$
$$p(EN|ES)=2.3$$
$$p(payday|EN)=6$$



$p(t-1=P) = 1998$
$p(EN|P) = 2.3$
$p(payday|EN) = 6$

min(5.7 + .22 + 6, 7.7 + 2.3 + 6, 1998 + 2.3 + 6

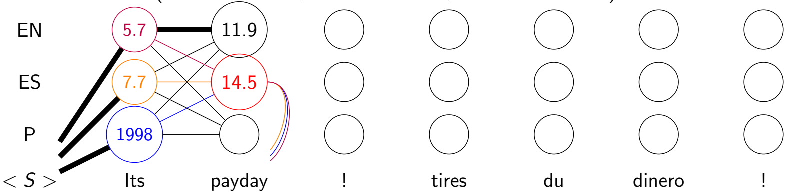


$$p(t-1=EN)=5.7 \qquad p(t-1=ES)=7.7 \qquad p(t-1=P)=1988$$

$$p(ES|EN)=2.3 \qquad p(ES|ES)=.36 \qquad p(ES|P)=1.6$$

$$p(payday|ES)=6.5 \qquad p(payday|ES)=6.5 \qquad p(payday|ES)=6.5$$

$$\min(5.7+2.3+6.5,\ 7.7+.36+6.5,\ 1998+1.6+6.5)$$

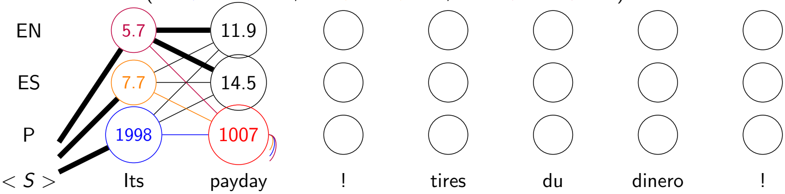


$p(t-1=EN)=5.7$ $\quad$ $p(t-1=ES)=7.7$ $\quad$ $p(t-1=P)=1988$

$p(P|EN)=2.3$ $\quad$ $p(P|ES)=1.6$ $\quad$ $p(P|P)=999$

$p(payday|P)=999$ $\quad$ $p(payday|P)=999$ $\quad$ $p(payday|P)=999$

min$(5.7+2.3+999, 7.7+1.6+999, 1998+999+999)$

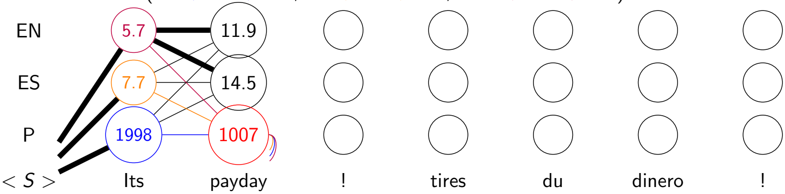


$p(t-1=EN)=5.7$ $\quad$ $p(t-1=ES)=7.7$ $\quad$ $p(t-1=P)=1988$

$p(P|EN)=2.3$ $\quad$ $p(P|ES)=1.6$ $\quad$ $p(P|P)=999$

$p(payday|P)=999$ $\quad$ $p(payday|P)=999$ $\quad$ $p(payday|P)=999$

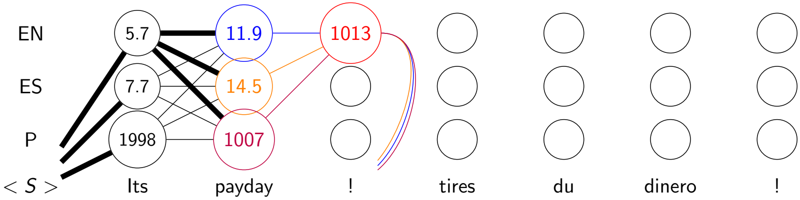
$$\min(5.7+2.3+999,\ 7.7+1.6+999,\ 1998+999+999)$$

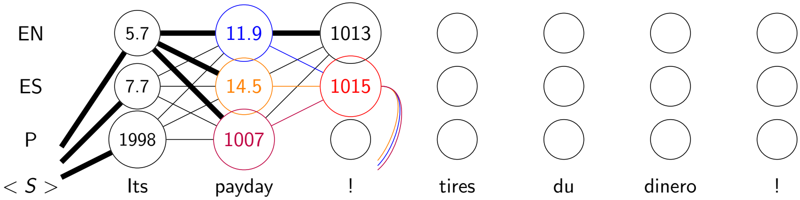


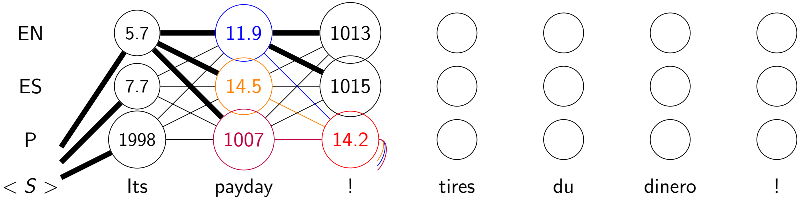
So the probability of an arc is made up of three parts:

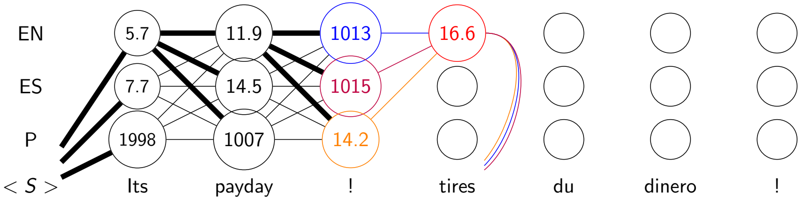
- probability of word given the label: **emission probability**
- probability of the label sequence: **transition probability**
- probability of previous state (history)

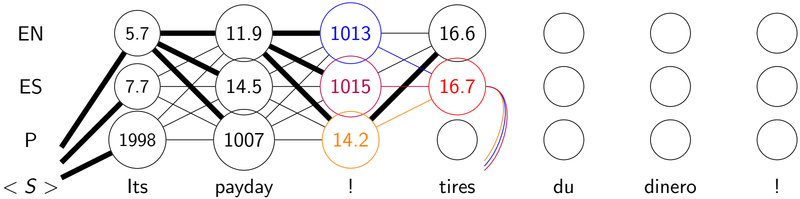
Menti: 7005 6703 TODO


EN
ES
P
< S >
5.7
7.7
1998
Its
11.9
14.5
1007
payday
1013
!
tires
du
dinero
!

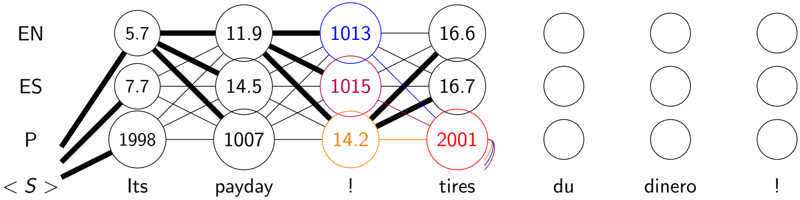

EN
ES
P
< S >
5.7
7.7
1998
11.9
14.5
1007
1013
1015
Its
payday
!
tires
du
dinero
!


EN
ES
P
< S >
5.7
7.7
1998
11.9
14.5
1007
1013
1015
14.2
Its
payday
!
tires
du
dinero
!

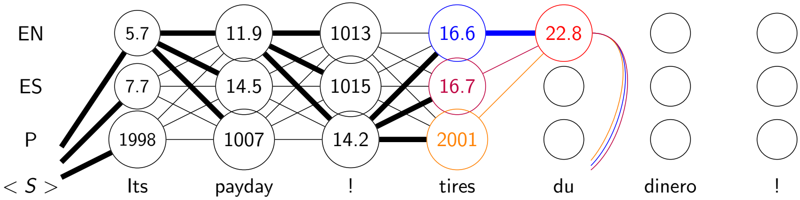

EN
ES
P
$< S >$
5.7
7.7
1998
11.9
14.5
1007
1013
1015
14.2
16.6
Its
payday
!
tires
du
dinero
!



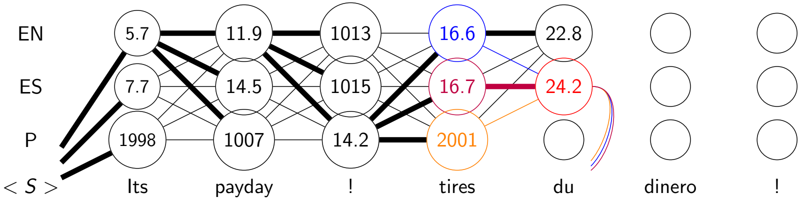
EN
ES
P
< S >
5.7
7.7
1998
Its
11.9
14.5
1007
payday
1013
1015
14.2
!
16.6
16.7
tires
du
dinero
!



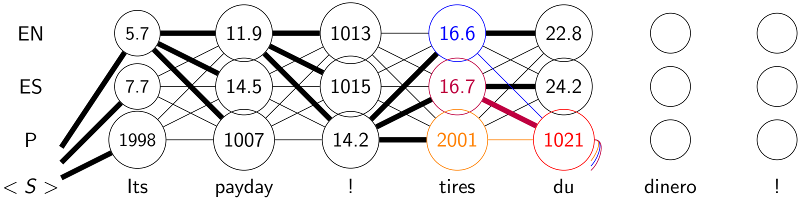
EN
ES
P
< S >
5.7
7.7
1998
Its
11.9
14.5
1007
payday
1013
1015
14.2
!
16.6
16.7
2001
tires
du
dinero
!



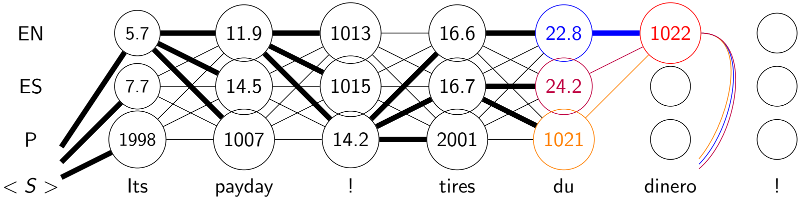
EN
ES
P
< S >
5.7
7.7
1998
Its
11.9
14.5
1007
payday
1013
1015
14.2
!
16.6
16.7
2001
tires
22.8
du
dinero
!

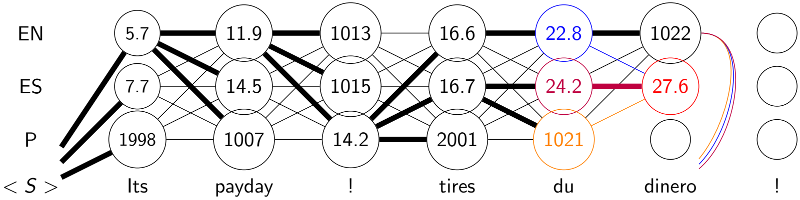


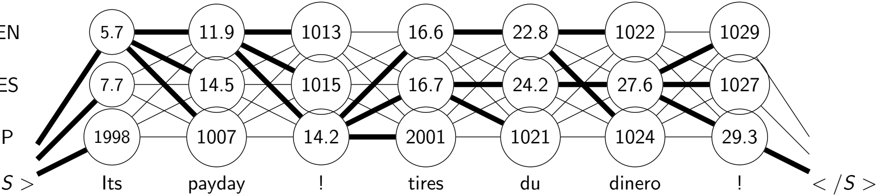
EN
ES
P
< S >
5.7
7.7
1998
Its
11.9
14.5
1007
payday
1013
1015
14.2
!
16.6
16.7
2001
tires
22.8
24.2
du
dinero
!

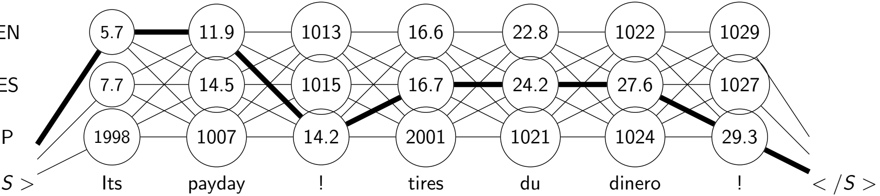


EN
ES
P
< S >
5.7
7.7
1998
Its
11.9
14.5
1007
payday
1013
1015
14.2
!
16.6
16.7
2001
tires
22.8
24.2
1021
du
dinero
!


EN
ES
P
< S >
5.7
7.7
1998
11.9
14.5
1007
1013
1015
14.2
16.6
16.7
2001
22.8
24.2
1021
1022
Its
payday
!
tires
du
dinero
!


EN
ES
P
< S >
5.7
7.7
1998
Its
11.9
14.5
1007
payday
1013
1015
14.2
!
16.6
16.7
2001
tires
22.8
24.2
1021
du
1022
27.6
dinero
!


EN
ES
P
S >
5.7
11.9
1013
16.6
22.8
1022
1029
7.7
14.5
1015
16.7
24.2
27.6
1027
1998
1007
14.2
2001
1021
1024
29.3
Its
payday
!
tires
du
dinero
!
< /S >



EN
ES
P
$S >$
5.7
11.9
1013
16.6
22.8
1022
1029
7.7
14.5
1015
16.7
24.2
27.6
1027
1998
1007
14.2
2001
1021
1024
29.3
Its
payday
!
tires
du
dinero
!
$< /S >$

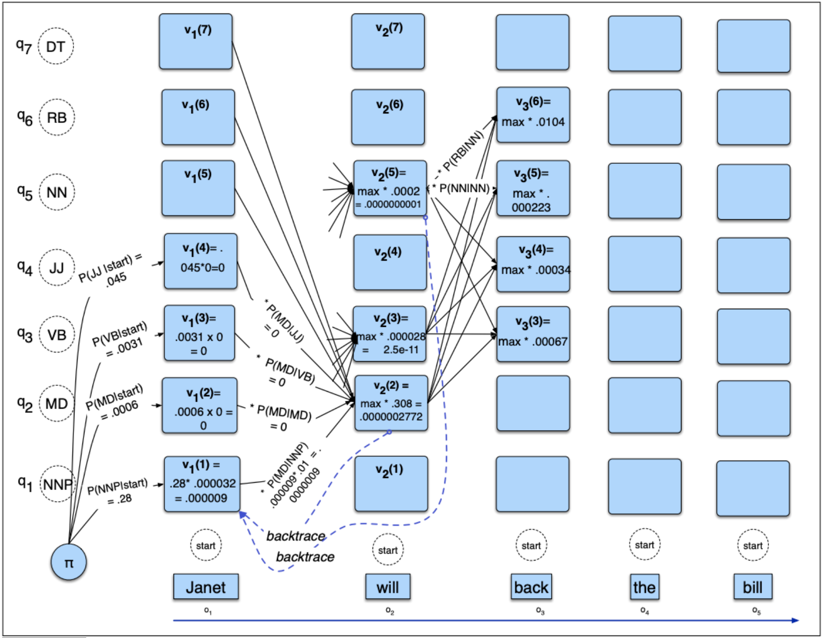


Figure 8.9 The first few entries in the individual state columns for the Viterbi algorithm. Each cell keeps the probability of the best path so far and a pointer to the previous cell along that path. We have only filled out columns 1 and 2; to avoid clutter most cells with value 0 are left empty. The rest is left as an exercise for the

Maximum Entropy Markov Models

Maximum entropy $==$ logistic regression

- Instead of using probabilities we are going to use machine learning to predict the tag
- Need to define features! (how?)

Original Viterbi

$$v_t(j) = max_{i=1}^{N} \overbrace{v_{t-1}(i)}^{\text{prev.}} \overbrace{a_{ij}}^{\text{trans.}} \overbrace{b_j(o_t)}^{\text{emis.}} \tag{3}$$

MEMM version

$$v_t(j) = max_{i=1}^{N} \overbrace{v_{t-1}(i)}^{\text{prev.}} \overbrace{P(s_j|s_i)}^{\text{trans.}} \overbrace{P(s_j|o_t)}^{\text{emis.}} \tag{4}$$

Note that $P(s_j|s_i)$ and $P(s_j|o_t)$ are simplifications!

Original Viterbi

$$v_t(j) = max_{i=1}^{N} \overbrace{v_{t-1}(i)}^{\text{prev.}} \overbrace{a_{ij}}^{\text{trans.}} \overbrace{b_j(o_t)}^{\text{emis.}} \tag{3}$$

MEMM version

$$v_t(j) = max_{i=1}^{N} \overbrace{v_{t-1}(i)}^{\text{prev.}} \overbrace{P(s_j|s_i)}^{\text{trans.}} \overbrace{P(s_j|o_t)}^{\text{emis.}} \tag{4}$$

Note that $P(s_j|s_i)$ and $P(s_j|o_t)$ are simplifications! MEMM version (optimized)

$$v_t(j) = max_i^{N} \overbrace{v_{t-1}(i)}^{\text{prev.}} \overbrace{P(s_j|s_i, o_t)}^{\text{trans. + emmis.}} \tag{5}$$

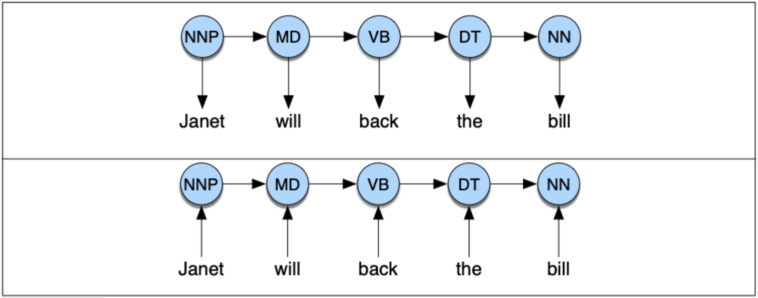


Figure 8.12 A schematic view of the HMM (top) and MEMM (bottom) representation of the probability computation for the correct sequence of tags for the *back* sentence. The HMM computes the likelihood of the observation given the hidden state, while the MEMM computes the posterior of each state, conditioned on the previous state and current observation.

(taken from J&M)

Because features are dynamic (generated during run-time), we provide feature templates:

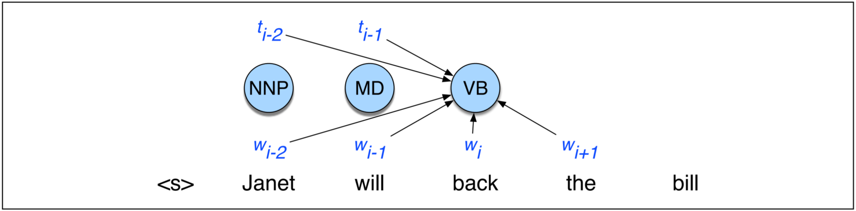


Figure 8.13 An MEMM for part-of-speech tagging showing the ability to condition on more features.

(taken from J&M)

Because features are dynamic (generated during run-time), we provide feature templates:

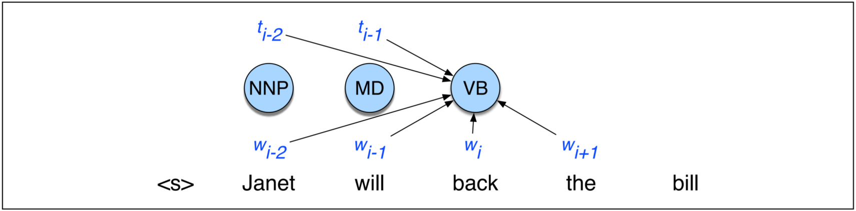


Figure 8.13 An MEMM for part-of-speech tagging showing the ability to condition on more features.

(taken from J&M) Can also include other properties of the word (prefix, suffix, punctuation, capitalization)

Other extensions

- $>$ 2-grams
- Smoothing
- Beam search

Going beyond bigrams

- unigrams: our most-frequent baseline
- bigrams: viterbi
- trigrams: ?

$$argmax_{t_1^n} \prod_{i=1}^{n} P(w_i|t_i)P(t_i|t_{i-1}) \tag{6}$$

becomes:

$$argmax_{t_1^n} \prod_{i=1}^{n} P(w_i|t_i)P(t_i|t_{i-1}t_{i-2}) \tag{7}$$

Instead of considering N (number of tags) states, we now have to consider N^2

- adding much more complexity for a small gain!
- ideally combined with beam search

Interpolation (again)

$$P(t_n|t_{n-1}) = \lambda * \frac{count(t_{n-1}, t_n)}{count(t_{n-1})} * (1 - \lambda)\frac{count(t_n)}{count(anytag)}$$

- emission probabilities: UNK token
- usually there is some "special treatment" for the emission probability $P(w_n|t_n)$ if w_n is unseen in the training corpus by taking for instance punctuation, capitalization, numbers, suffixes into account

Interpolation (again)

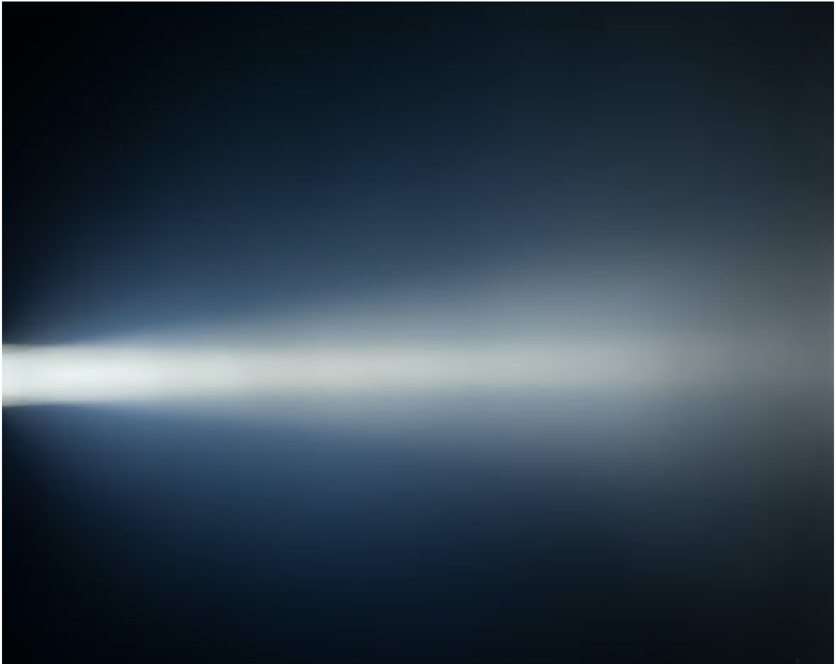
$$P(t_n|t_{n-1}) = \lambda * \frac{count(t_{n-1}, t_n)}{count(t_{n-1})} * (1 - \lambda)\frac{count(t_n)}{count(anytag)}$$

- emission probabilities: UNK token
- usually there is some "special treatment" for the emission probability $P(w_n|t_n)$ if w_n is unseen in the training corpus by taking for instance punctuation, capitalization, numbers, suffixes into account
- "special treatment" breaks some probability assumptions (sum to 1)
- What value for λ?

Interpolation

Because many counts are low for POS tagging, we use $a = 0.01$ instead of $a = 1$ for laplace smoothing in the assignment!

Beam search



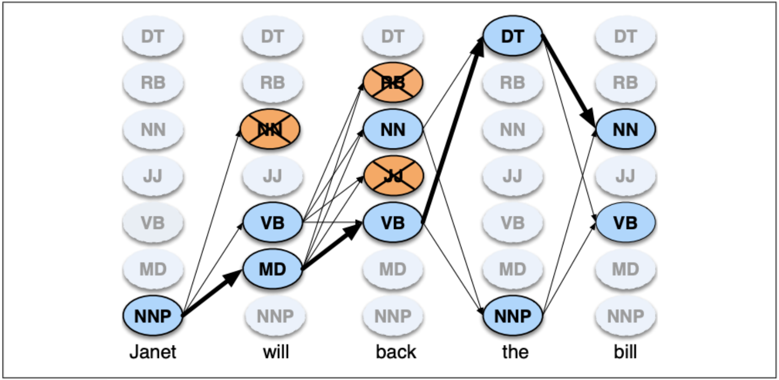


Figure 8.11 A beam search version of Fig. 8.6, showing a beam width of 2. At each time t, all (non-zero) states are computed, but then they are sorted and only the best 2 states are propagated forward and the rest are pruned, shown in orange.

(taken from J&M)

Summary

- HMM and Viterbi
- MEMM
- limitations and extensions

Alternatives

- CRF
- RNNs (Recurrent Neural Networks)