Lecture 2 - Text Corpora

Corpora

- A **corpus** body of utterances, as words or sentences, assumed to be *representative* of and used for lexical, grammatical, or other linguistic analysis
- To understand and model how language works, we need empirical evidence; ideally, this should be **naturally-occuring** corpora
- Aside from utterances, corpus datasets include **metadata** - side information about where the utterances come from, such as author, date, topic, publication, etc.
- Corpora with **linguistic annotations** are of particular interest for *core NLP* and therefore this course, where humans have read the text and marked categories and/or structures describing their syntax and/or meaning
    - Can be derived automatically from the original data artifact (such as star ratings) too
    - Consistency of human annotators is a big issue - unambiguous rules are required to resolve disagreements

Sentiment Analysis

- Goal: predict the *opinion* expressed in a piece of text
    - Either *positive* or *negative*
    - Or *rating* on a scale
- The simplest way is to count then number of words with positive and negative denotations/connotations

Building a Sentiment Analyzer

1. What is the input for each prediction? Sentence? Full review text? Text + metadata?
2. What are the possible outputs? Positive or negative? Stars?
3. How will it decide?
    a. When a system's behaviour is determined solely by manual rules or databases, it is said to be **rule-based**, **symbolic**, or **knowledge-driven** (early days of computational linguistics)
    b. **Learning** is the act of collecting statistics or patterns automatically from corpora to govern the system's behaviour (dominant in most areas of contemporary NLP)
        i. **Supervised learning** is when the data provides example input-output pairs
        ii. Core behaviour: **training**
        iii. Refining behaviour: **tuning**
4. How will you measure its effectiveness? This requires data! Require copra to evaluate

- Before you build a system, choose a dataset for evaluation
- Why is data-driven evaluation important?
    - Good science requires controlled experimentation
    - Good engineering requires benchmarks
    - Your intuitions about typical inputs are probably wrong
- Often you should have multiple evaluation datasets: one for *development* as you hack on your system, and one reserved for *final* testing
    - So that we don't optimize for the benchmark only! (for instance by overfitting)
- **Gold labels**: correct labels
- **Evaluation**
    - Simplest measure: $accuracy = \frac{\#\ correct}{\#\ total}$

Simple counting

A simple sentiment classification algorithm

- Use a **sentiment lexicon** to count positive and negative words:

Positive:

absolutely	beaming	calm
adorable	beautiful	celebrated
accepted	believe	certain
acclaimed	beneficial	champ
accomplish	bliss	champion
achieve	bountiful	charming
action	bounty	cheery
active	brave	choice
admire	bravo	classic
adventure	brilliant	classical
affirm	bubbly	clean
...		...

Negative:

abysmal	bad	callous
adverse	banal	can't
alarming	barbed	clumsy
angry	belligerent	coarse
annoy	bemoan	cold
anxious	beneath	collapse
apathy	boring	confused
appalling	broken	contradictory
atrocious		contrary
awful		corrosive
		corrupt
		...

- Simplest rule: count positive and negative words in the text and predict whichever is greater!
- Problems with simple counting
  - Hard to know whether words that *seem* positive or negative tend to actually be used that way
    - Sense ambiguity, e.g. "It was awfully beautiful."
    - Sarcasm/irony, e.g. "Oh yeah it was definitely wonderful!"
    - Text could mention expectations or opposing viewpoints, in contrast to author's actual opinion, e.g. "I was expecting a great movie as my friends described it as a brilliant classic but I think it was just awful."
    - To address this problem, use a data-driven method: use **frequency counts** from *training* corpus to ascertain which words tend to be positive or negative

Solution:

  - Opinion words may be describing, for example, a character's attitude rather than being an evaluation of the film, e.g. "The villain in the movie did horrible things to the main characters."
  - Some words act as semantic modifiers of other opinion-bearing words/phases so interpreting the full meaning requires sophistication:
    - I **can't** stand this movie
    - I **can't** believe how great the movie is

Preprocessing and Normalisation

- Normal written conventions often do not reflect the 'logical' organisation of textual symbols
  - For example, some punctuation marks are written adjacent to the previous or following word, even though they are not part of it
  - The details vary according to language and style guide
- Given a string of raw text, a **tokeniser** adds logical boundaries between separate words/punctuation **tokens** (occurrences) not separated by spaces:

```
['Daniels', 'made', 'several', 'appearances', 'as',
↪ 'C-3PO', 'on', 'numerous', 'TV', 'shows', 'and',
↪ 'commercials', ',', 'notably', 'on', 'a', 'Star',
↪ 'Wars-themed', 'episode', 'of', 'The', 'Donny',
↪ 'and', 'Marie', 'Show', 'in', '1977', ',',
↪ 'Disneyland', "'s", '35th', 'Anniversary', '.']
```

  - To a large extent, this can be automated by rules. But there are always difficult cases (e.g. "C-3PO)
- English tokenisation conventions vary somewhat, for example, with respect to:
  - **Clitics** (contracted forms): 's, n't, 're, etc.
  - **Hyphens** in compounds like *president-elect*

- Word-level tokenisation is just part of the larger process of preprocessing or normalisation, which may also include:
    - Encoding conversion
    - Removal of markup
    - Insertion of markup
    - Case conversion
    - Sentence boundary detection (called **sentence tokenisation**)

```
+ <s><\s>
Edit distance
Mask for BERT
```

- It should be evident that a *large number of decisions* have to be made, many of them dependent on the eventual intended use fo the output, before a satisfactory preprocessor for such data can be produced
    - *Documenting* those decisions and their implementation is then a key step in establishing the credibility of any subsequent experiments
        - Documentation is especially important if the dataset is to be distributed publicly

Choice of Training and Evaluation Data

- We know that the way people use language varies considerably depending on **context**. Factors include:
    - **Mode of communication**: speech (in person, telephone, …), writing (print, SMS, web, …)
    - **Topic:** chit-chat, politics, sports, physics
    - **Genre:** news story, novel, Wikipedia article, persuasive essay, political address, tweet
    - **Audience:** formality, politeness, complexity (e.g. child-directed speech)
    - In NLP, **domain** is a cover-term for all these factors
- Statistical approaches typically assume that the training data and the test data are sampled from the same distribution
    - I.e. if you say an example data point, it would be hard to guess whether it was from the training or test data
- Things can go awry if the test data is appreciably different, for example
    - Different tokenisation conventions
    - New vocabulary
    - Longer sentences
    - More colloquial/less edited style
    - Different distribution of labels
- **Domain adaptation** techniques attempt to correct for this assumption when something about the source/characteristics of the test data is known to be different

Lecture 10 - Methods in Annotation and Evaluation

Annotation

- Annotation costs time and money, you need to decide on
  - Source data: genre? Size? Licensing?
  - Annotation scheme: complexity? Guidelines?
  - Annotators: expertise? Training?
  - Annotation software: graphical interface? Scanning papers?
  - Quality control: multiple annotation? Adjudication process?
- Text might be ambiguous
- There may be grey area between categories in the annotation scheme
  - Multiple equally valid decisions can be plausible

Documenting conventions, help annotators produce consistent data, and to help end users interpret the annotations correctl y

Solve IAA drawback:
Cohen's kappa:

- Raw agreement rate counts all annotation decisions equally.
- Some measures take knowledge about the annotation scheme into account (e.g., counting singular vs. plural noun as a minor disagreement compared to noun vs. preposition).
- What if some decisions (e.g., POS tags) are far more frequent than others?
  - If 2 annotators both tagged *hell* as a noun, what is the chance that they agreed by accident? What if they agree that it is an interjection (rare tag)—is that equally likely to be an accident?
  - **Chance-corrected** measures such as Cohen's kappa (κ) adjust the agreement score based on label probabilities.
  - . . . but they make modeling assumptions about how "accidental" agreement would arise; important that these match the reality of the annotation process!
  - More below on hypothesis testing/statistical significance.

Inter-Annotator Agreement (IAA)

- An important way to estimate the reliability of annotations is to have multiple people independently annotate a common sample, and measure *inter-annotator agreement*
  - **Raw agreement rate** is the proportion of labels in agreement
    - Some measures take knowledge about the annotation scheme into account (e.g. counting singular vs plural noun as a minor disagreement compared to noun vs preposition)
- The agreement rate can be thought of an upper bound (**human ceiling**) on the accuracy of a system evaluated on that dataset

Cross-Validation

- What if our dataset is **too small** to have a nice train/test or train/dev/test split?
- **K-fold cross-validation:** partition the data into k pieces and treat them as mini held-out sets; each **fold** is an experiment with a different held-out set, using the rest of the data for training
  - After k folds, every data point will have a held-out prediction
  - If we are tuning the system via cross-validation, it is still important to have a separate blind test set

Careful of overfitting on dev set

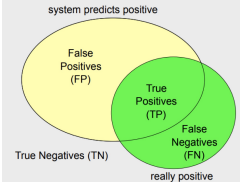
Measuring a model's performance

- **Precision:** proportion of model's answers that are right
- **Recall:** proportion of test data that model gets right

- For isolating performance on a particular label in multi-label tasks, or
- For chunking, phrase structure parsing, or anything where word-by-word accuracy isn't appropriate.
- F_1-score: Harmonic mean of precision (proportion of model's answers that are right) and recall (proportion of test data that model gets right).
- E.g., for the POS tag NN:

Confusion Matrix

	Actually Positive (1)	Actually Negative (0)
Predicted Positive (1)	True Positives (TPs)	False Positives (FPs)
Predicted Negative (0)	False Negatives (FNs)	True Negatives (TNs)



Want:
Large diagonal, small FP FN

$$\text{Precision } P = \frac{|\text{tokens correctly tagged NN}|}{|\text{all tokens automatically tagged NN}|} = \frac{\text{TP}}{\text{TP+FP}}$$

$$\text{Recall } R = \frac{|\text{tokens correctly tagged NN}|}{|\text{all tokens gold-tagged NN}|} = \frac{\text{TP}}{\text{TP+FN}}$$

$$F_1 = \frac{2 \cdot P \cdot R}{P+R}$$

FP: orig 0; pred 1
FN: orig 1; pred 0

Bounds

- Upper bounds
  - Turing test: when using human Gold Standard, check the agreement of humans against that standard
- Lower bounds
  - Baseline: performance of a simpler model
  - Majority baseline: model always picks the most frequent/likely class

Significance

- When we are evaluating a model against each other or to a bound, how do we decide **if the differences we find are significant?**

Between 2 model

i.e.see if differences are 'real' or within margin of error (i.e., likely due to chance)

- In other words, should we interpret the differences as down to pure chance? Or is something more going on?

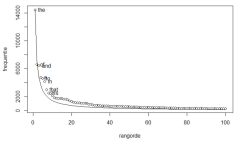
Significance Test

- **Parametric** when the underlying distribution is **normal**
  - T-test, Z-test, etc.
- **Non-parametric** otherwise
  - Usually we do need non-parametric tests: remember Zipf's Law!
  - Can use McNemar's test or variants of it
    Or Stochastic / pemutation tests
    (esp .complex predictions, such as parse trees)

discrete power law distribution

Word frequency distribution

Follow by Ngram



50% of words occur once

Error analysis: quantitative & qualitative; improve system;

Confusion matrix

>1 right answer:

Evaluation: 'Turing Test'

- Classify candidate paraphrases as high, medium or low probability.
- Measure correlation between human vs. machine's judgements.
- Result was 0.64. Is that good?
- Upper bound: average correlation between two human judges! That's 0.74.

Create test set:

Alternative to [test set domain = training set domain]

- Creating test sets from other domains but with the same guidelines
- Split training sets to simulate the shift (e.g., take different years of WSJ)

Building systems robust to the shifts in distribution is a big challenge, and an exciting topic

Lecture 3 - N-gram Language Models

Probability of a sentence

- How likely is it to occur in natural language
  - Consider only a specific language (English, or even more specifically British English)
  - Not including meta-language (e.g. linguistic discussion)
  - P("the cat slept peacefully") > P("slept the peacefully cat")
    - Use case: generative NLP (abstractive summarisation)
  - P("she studies morphosyntax") > P("she studies more faux syntax")
    - Use case: audio transcription
  - It's very difficult to know the true probability of an arbitrary sequence of words
  - But we can define a **language model** to give us a good *approximation*
  - Like all models, language models will be good at capturing *some* things and *less* good for others
    - We might want different models for different tasks

Use cases of a Language Model

- Spelling correction
  - Sentence probabilities help decide correct spelling
- Automatic speech recognition
  - Sentence probabilities help decide between similar-sounding options
- Machine translation
  - Sentence probabilities help decide word choice and word order
- Prediction
  - Language models can be used for prediction as well as correction
  - E.g. predictive text correction/completion on your mobile phone

    predict as you are typing [ineff…]
  - In this case, a language model may be defined over sequences of characters instead of (or in addition to) sequences of words

Estimating Probabilities

- We want to know the probability of a word sequence **w** = $w_1 ... w_n$ occurring in English
- Assume we have some **training data**: large corpus of general English text
- We can use this data to **estimate** the probability of **w** (even if we never see it in the corpus!)

Notation

- I will often omit the random variable in writing probabilities, using $P(x)$ to mean $P(X = x)$.
- When the distinction is important, I will use
  - $P(x)$ for *true* probabilities
  - $\hat{P}(x)$ for *estimated* probabilities
  - $P_{\mathrm{E}}(x)$ for estimated probabilities using a particular estimation method E.
- But since we almost always mean estimated probabilities, I may get lazy later and use $P(x)$ for those too.

Relative Frequency Estimation

- Intuitive way to estimate discrete probabilities

$$P_{RF}(x) = \frac{C(x)}{N}$$

  Where:
  - $C(x)$ is the count of x in a large dataset
  - $N = \sum_{x'} C(x')$ is the total number of items in the dataset
- This method is also known as **maximum-likelihood estimation (MLE)**

Problems

- Using MLE on full sentences doesn't work well for language model estimation

- All sentences that have never occurred get zero probability even if they are grammatical (and meaningful)
- In general, MLE thinks anything that *hasn't occurred* will never occur (P=0)
  - And similarly for word sequences that have never occurred

Trigram general case:

- To estimate $P(\vec{w})$, use chain rule and make an indep. assumption:

$$P(w_1, \ldots w_n) = \prod_{i=1}^{n} P(w_i | w_1, w_2, \ldots w_{i-1})$$
$$\approx P(w_1) P(w_2|w_1) \prod_{i=3}^{n} P(w_i | w_{i-2}, w_{w-1})$$

- Then estimate each trigram prob from data (here, using MLE):

$$P_{MLE}(w_i | w_{i-2}, w_{i-1}) = \frac{C(w_{i-2}, w_{i-1}, w_i)}{C(w_{i-2}, w_{i-1})}$$

Sparse data

- In fact, even things that occur once or twice in our training data are a problem
- The sparse data problem is that there are **not enough observations to estimate probabilities well simply by counting observed data**
- For sentences, many (most!) will occur rarely if ever in our training data, so we need to do something far smarter

N-gram models

- One way to tackle the sparse data problem is to estimate **P(w)** by combining the probabilities of *smaller parts* of the sentence, which will occur more frequently
- This is the intuition behind n-gram language models

Deriving an N-Gram Model

Joint probability

$$P(S = \text{"the cat slept quietly"}) = P(the,\ cat,\ slept,\ quietly)$$
$$= P(quitely \mid the,\ cat,\ slept)\ x\ P(the,\ cat,\ slept)$$
$$= P(quitely \mid the,\ cat,\ slept)\ x\ P(slept \mid the,\ cat)\ x\ P(the,\ cat)$$
$$= P(quitely \mid the,\ cat,\ slept)\ x\ P(slept \mid the,\ cat)\ x\ P(cat \mid the)\ x\ P(the$$

The chain rule for a joint probability, $P(X,Y) = P(Y|X)P(X)$

- More generally, we can use the chain rule $P(w_1, \ldots w_n) = \prod_{i=1}^{n} P(w_i | w_1, w_2, \ldots w_{i-1})$
- But many of these conditional probabilities are just as sparse
  - If we want $P(the,\ cat,\ slept,\ quietly)$ we still need $P(quitely \mid the,\ cat,\ slept)$
- So we make an **independence assumption**: the probability of a word only depends on a fixed number of previous words (called **history**)

trigram model: $P(w_i|w_1, w_2, \ldots w_{i-1}) \approx P(w_i|w_{i-2}, w_{i-1})$
bigram model: $P(w_i|w_1, w_2, \ldots w_{i-1}) \approx P(w_i|w_{i-1})$
unigram model: $P(w_i|w_1, w_2, \ldots w_{i-1}) \approx P(w_i)$

- This assumption is not always a good one, but it does reduce the sparse data problem
- If we use MLE, we consider: Use counts!
  - $P_{MLE}(\text{mast}|\text{before, the}) = \frac{C(\text{before, the, mast})}{C(\text{before, the})}$ $\quad P_{MLE}(w_i|w_{i-2}, w_{i-1}) = \frac{C(w_{i-2}, w_{i-1}, w_i)}{C(w_{i-2}, w_{i-1})}$

Beginning/End of Sequence

	w_{-1}	w_0	w_1	w_2	w_3	w_4	w_5
(1)	<s>	<s>	he	saw	the	yellow	</s>
(2)	<s>	<s>	feeds	the	cats	daily	</s>

assume $w_{-1} = w_0 =$ <s> and $w_{n+1} =$ </s> so:

$$P(\vec{w}) = \prod_{i=1}^{n+1} P(w_i|w_{i-2}, w_{i-1})$$

Use punctuation instead of <s>:

Alternatively, we could model all sentences as one (very long) sequence, including punctuation:

Lowercased
two cats live in sam 's barn . sam feeds the cats daily . yesterday , he saw the yellow cat catch a mouse . [...]

Now, trigrams like $P(.|\text{cats daily})$ and $P(,|.\ \text{yesterday})$ tell us about behavior at sentence edges.

Costs (negative log probabilities)

prob=1, cost=0; prob=0,cost=infinity

- Word probabilities are typically very small
- Multiplying lots of small probabilities quickly gets so tiny that we cannot represent the numbers accurately, even with double precision floating point
- So in practice, we typically use **negative log probabilities** (also called **costs**)
  - Since lower probabilities range from 0 to 1, negative log probabilities range from 0 to infinity
  - *Lower* cost = *higher* probability
  - Instead of *multiplying* probabilities, we *add* negative log probabilities

Problems

- N-gram models can be too simplistic, length of a 'context' often varies: can be shorter or longer than an arbitrary *N*
  - Longer histories may capture more but are also more sparse
- Still suffers from assigning zero probabilities to not-seen sequences

Entropy of English

- Given the start of a text, can we guess the next word?
- For humans, the measured entropy is only about 1.3.
  - Meaning: on average, given the preceding context, a human would need only 1.3 y/n questions to determine the next word.
  - This is an upper bound on the true entropy, which we can never know (because we don't know the true probability distribution).

Per-word cross-entropy of the unigram model is about 11.

Bigram: 6

ASCII uses an average of 24 bits per word

Lecture 4 - Language Models: Evaluation and Smoothing

Evaluating a Language Model

- Intuitively, a trigram model captures more context than a bigram model, so it should be a 'better' model
  - That is, it should *more accurately predict the probabilities of sentences*
  - But how can we measure this?

Types of Evaluation in NLP

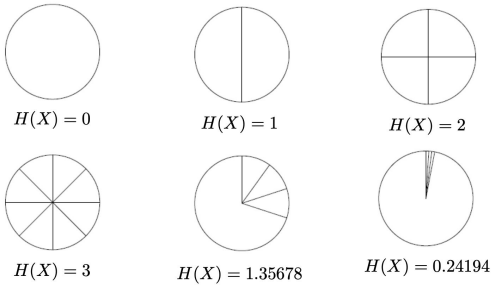
- **Extrinsic:** measure performance on a downstream application
  - For a language model, plugging it into a machine translation, automated speech recognition, etc.
  - The most reliable evaluation, but can be time-consuming
  - And of course, we still need an evaluation measure for the downstream system!
- **Intrinsic:** design a measure that is *inherent* to the current task
  - Can be much quicker/easier during development cycle
  - But not always easy to figure out what the right measure is
    - Ideally, on that correlates well with extrinsic measures

Entropy

can't evaluate an LM with accurac
since nominator (predicted word) may be a perfectly good guess (valid but counted wrong)

$$H(X) = \sum_x -P(x)log_2P(x) = E[-log_2P(X)]$$ = expected value of (-log2P(X))

- Intuitively, a measure of uncertainty/disorder
  - A measure of how 'surprising' a probability distribution is measures confidence in the model's predictions
- Example: where the area of a section is proportional to its probability



Entropy as Yes/No Questions

- Entropy is the answer to how many yes-no questions (bits) do we need to find out the outcome (or to *encode* the outcome)
- Uniform distributions with 2^n outcomes require n yes-no questions
- Average number of bits needed to encode $X \geq entropy\ of\ X$ shannon's second theorem

Estimates and Cross Entropy

- A good model should have low uncertainty (entropy) about what comes next
  - Lower cross entropy means that a model is better at predicting the next element (e.g. the next word)
- Cross entropy measures how close $\widehat{P}$ (estimate) is to P (true):

$$H(P,\widehat{P}) = \sum_x -P(x)log_2\widehat{P}(x)$$

- Note that cross-entropy $\geq$ entropy

- A model's uncertainty can be no less than the true uncertainty
- But we still don't know $P(x)$

Estimating Cross Entropy

- For $w_1, ..., w_n$ with a large n, per-word cross-entropy is well approximated by:

$$H_M(w_1, ..., w_n) = -\frac{1}{n} log_2 P_M(w_1, ..., w_n)$$

- That is just the average negative log probability our model assigns to each word in the sequence (i.e. normalised for sequence length)

Better LM = better compression for the task

Perplexity

- Language model performance is often reported as **perplexity** rather than cross-entropy

$$perplexity = 2^{cross-entropy}$$

- The average branching factor at each decision point is 2, if our distribution were uniform
- So, 6 bits of cross-entropy means our model's perplexity is $2^6 = 64$; equivalent uncertainty to a uniform distribution over 64 outcomes

Interpreting Measures

- Cross entropy of a language model on some corpus is 5.2
- Is this *good*?
- No way to tell! Cross entropy depends on both the *model* and the *corpus*
  - **Some languages are simply more predictable** (e.g. casual speech vs. academic writing)
    - So lower cross entropy could mean that the corpus is 'easy', rather than the model is good
  - We can only compare different models on the same corpus
    - Measured on 'held-out' data

Could only compare CE of 2 models on same corpus

Sparse data, again

- Remember that MLE assigns zero probability to never observed sequences
- Meaning that cross-entropy would be infinite (because of $log_2 0$)
- Basically right: our model says that something should *never* occur, so our model is infinitely wrong (or rather, we are infinitely surprised) when it does occur!
- Even with a unigram model we we will run into words we never saw before
  - So even with short(est) n-grams, we need better ways to estimate probabilities from sparse data

Smoothing

= everything *in training* sums to 1
Everything not seen = 0

- The flaw of MLE is that it estimates probabilities that make the training data maximal;y probable, by making everything else (i.e. unseen data) minimally probable
- **Smoothing** methods address the problem by stealing probability mass from seen events and (*somehow*) reallocating it to unseen events
- There are lots of different methods based on different kinds of assumptions
- In smoothing, we have to ensure that all the probabilities still sum up to 1

Add-One (Laplace) Smoothing

- Just pretend we saw everything one more time than we did

$$P_{+1}(w_i \mid w_{i-2},\ w_{i-1}) = \frac{C(w_{i-2}, w_{i-1}, w_i) + 1}{C(w_{i-2}, w_{i-1}) + v}$$

where v is the vocabulary size.

- Problems

For very large vocabulary siz

  - Laplace smoothing steals *way* too much from seen events
    - In fact, MLE is pretty good for frequent events, so we shouldn't want to change these much
  - Assumes that we know the vocabulary size in advance
    - And also that the size of our vocabulary is fixed
    - To remediate, we can just add a single 'unknown' (UNK) item, and use this for all unknown words

Add-α (Lidstone) Smoothing

- We can improve things by adding α <

$$P_{+\alpha}(w_i \mid w_{i-2},\ w_{i-1}) = \frac{C(w_{i-2}, w_{i-1}, w_i) + \alpha}{C(w_{i-2}, w_{i-1}) + \alpha v}$$

Choose alpha by dev set (minimizes dev set cross-entropy & avoid overfitting)

Good-Turing Smoothin

- Previous methods changed the denominator, which can have big effects on frequent events
- Good-Turing changes the numerator only
- Think of Good-Turing like this:
  - MLE divides count c of n-gram by count n of history: $P_{ML} = \frac{c}{n}$
  - Good-Turing uses **adjusted counts** c^* instead: $P_{GT} = \frac{c^*}{n}$
- There are even better methods!

- Push every probability total down to the count class below.
- Each *count* is reduced slightly (Zipf): we're discounting

c	N_c	P_c	$P_c[total]$	$c*$	$P*_c$	$P*_c\,[total]$
0	N_0	0	0	$\frac{N_1}{N_0}$	$\frac{\frac{N_1}{N_0}}{N}$	$\frac{N_1}{N}$
1	N_1	$\frac{1}{N}$	$\frac{N_1}{N}$	$2\frac{N_2}{N_1}$	$\frac{2\frac{N_2}{N_1}}{N}$	$\frac{2N_2}{N}$
2	N_2	$\frac{2}{N}$	$\frac{2N_2}{N}$	$3\frac{N_3}{N_2}$	$\frac{3\frac{N_3}{N_2}}{N}$	$\frac{3N_3}{N}$

- c: count
  N_c: number of different items with count c
  P_c: MLE estimate of prob. of that item
  $P_c[total]$: MLE total probability mass for *all* items with that count.
  $c*$: Good-Turing smoothed version of the count
  $P*_c$ and $P*_c\,[total]$: Good-Turing versions of P_c and $P_c[total]$

- Basic idea is to arrange the discounts so that the amount we *add* to the total probability in row 0 is matched by all the discounting in the other rows.
- Note that we only know N_0 if we actually know what's missing.
- And we can't always estimate what words are missing from a corpus.
- But for bigrams, we often assume $N_0 = V^2 - N$, where V is the different (observed) words in the corpus.

= vocab size

$$c* = (c+1)\frac{N_{c+1}}{N_c}$$
$$P*_c = \frac{c*}{N} = \frac{(c+1)\frac{N_{c+1}}{N_c}}{N}$$

General formula & intuition:

- Since counts tend to go down as c goes up, the multiplier is < 1.

Nc+1 / Nc

- The sum of all discounts is $\frac{N_1}{N_0}$. We need it to be, given that this is our GT count for row 0!

= added count for zero occurance

Good-Turing justification: 0-count items

- Estimate the probability that the next observation is previously unseen (i.e. will have count 1 once we see it)

$$P(\text{unseen}) = \frac{N_1}{n}$$

This part uses MLE!

- Divide that probability equally amongst all unseen events

Divide by N0

$$P_{GT} = \frac{1}{N_0}\frac{N_1}{n} \quad \Rightarrow \quad c^* = \frac{N_1}{N_0}$$

Good-Turing justification: 1-count items

- Estimate the probability that the next observation was seen once before (i.e. will have count 2 once we see it)

$$P(\text{once before}) = \frac{2N_2}{n}$$

- Divide that probability equally amongst all 1-count events

Divide by N1

$$P_{GT} = \frac{1}{N_1}\frac{2N_2}{n} \quad \Rightarrow \quad c^* = \frac{2N_2}{N_1}$$

- Same thing for higher count items

Lecture 5 - More Smoothing and the Noisy Channel Model

Problems with Good-Turing

- Assumes we know the vocabulary size (i.e. no unseen words)

Solution:
- Use UNK

- Does not allow 'holes' in the counts (i.e. if $N_i > 0$ then $N_{i-1} > 0$)
  - Use linear regression to estimate
- Applies discounts even to high-frequency items
- Assigns equal probabilities to all unseen events
  - A better solution is to **use information from lower order N-grams (shorter histories)**
    - Beer drinkers (likely)
    - Beer eaters (unlikely)
  - Two methods:
    - Interpolation
    - Backoff

Interpolation

- Combine higher and lower N-gram models, since they have different strengths and weaknesses:
  - **Higher-order** n-grams are **sensitive to more context**, but have **sparse** counts
  - **Lower-order** n-grams have **limited context** but **robust** counts
- If P_N is N-gram estimate (from MLE, GT, etc; $N = 1\ to\ 3$), use:

$$P_{INT}(w_3|w_1,w_2) = \lambda_1 P_1(w_3) + \lambda_2 P_2(w_3 \mid w_2) + \lambda_3 P_3(w_3 \mid w_1, w_2)$$

E.g. $P_{INT}(three \mid I,\ spent) = \lambda_1 P_1(three) + \lambda_2 P_2(three \mid spent) + \lambda_3 P_3(three \mid I,\ spent)$

- Note that all λ_i s must sum to 1!

Fitting Interpolation Parameters

- In general, any weighted combination of distributions is called a **mixture model**
- So λ_i s are **interpolation parameters** or **mixture weights**
- The values of the λ_i s are chosen to optimise perplexity on a held-out dataset

Katz Back-Off

- Solve the problem in a similar way to Good-Turing smoothing
- Discount the trigram-based probability estimates
- This leaves some probability mass to share among the estimates from the lower order models
- Instead of distributing the mass *uniformly* over unseen items, use it for **backoff estimates**

$$P_{\mathrm{BO}}(w_n|w_{n-N+1:n-1}) = \begin{cases} P^*(w_n|w_{n-N+1:n-1}), & \text{if } C(w_{n-N+1:n}) > 0 \\ \alpha(w_{n-N+1:n-1}) P_{\mathrm{BO}}(w_n|w_{n-N+2:n-1}), & \text{otherwise.} \end{cases}$$

Diversity of Histories

- "York" almost always directly follows "New", say in a corpus
- So, in unseen bigram contexts, "York" should have low probability
  - Lower than predicted by unigram model as used in interpolation/backoff

Kneser-Ney Smoothing

- Kneser-Ney smoothing takes diversity of histories into account

- Kneser-Ney smoothing takes diversity of histories into account
- Count of distinct histories for a word:

$$N_{1+}(\bullet w_i) = |\{w_{i-1} : c(w_{i-1}, w_i) > 0\}|$$

= number of different contexts word w has appeared in
= number of bigram types it completes

- Recall: maximum likelihood est. of unigram language model:

$$P_{ML}(w_i) = \frac{C(w_i)}{\sum_w C(w)}$$

- In KN smoothing, replace raw counts with count of histories:

$$P_{KN}(w_i) = \frac{N_{1+}(\bullet w_i)}{\sum_w N_{1+}(\bullet w)}$$

it gives you the probability of "appearing in new contexts"

Kneser-Ney in practice

- Original version used backoff, later "modified Kneser-Ney" introduced using interpolation
- Fairly complex equations, but until recently the best smoothing method for word n-grams

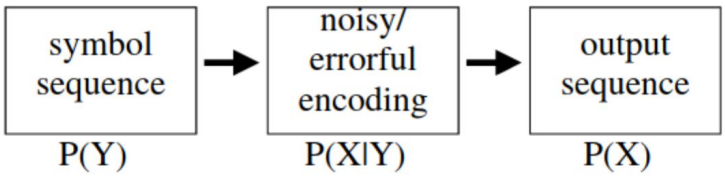
Distributed Representations and Word Similarity

By similar vectors
capture both semantic and syntactic similarity.

- E.g. Word2Vec
- Use neural networks to project words into a continuous space, so words that appear in similar contexts have similar representation
- Can $P(salmon \mid caught,\ two)$ tell us anything about $P(swordfish \mid caught,\ two)$? If C(salmon) >> C(swordfish)
  - N-gram models say no
  - But we know that both are fish and can be caught

Noisy Channel

- We imagine that someone tries to communicate a sequence to us, but noise is introduced; we only see the output sequenced
-



Application	Y	X
Speech Recognition	spoken words	acoustic signal
Machine Translation	words in L_A	words in L_B
Spelling Correction	intended words	typed words

- $P(Y)$ is the language model Intend
- $P(X|Y)$ is the distribution describing the 'likelihood' of the output given the intention; we call it the **noise model**
- $P(X)$ is the resulting distribution over what we actually see
- Given some particular observation x, we want to recover the most probable y that was intended

Noisy Channel as Probabilistic Inference

Y that maximize P(y|x)

- Mathematically, what we want is $argmax_y P(y|x)$
- Rewrite using Bayes' rule:

$$argmax_y P(y|x) = argmax_y \frac{P(x|y)P(y)}{P(x)} = argmax_y P(x|y)P(y)$$

  - $P(x|y)$ is the noise model
    - Varies heavily depending on the application: acoustic model, translation model, misspelling model, etc.
  - $P(y)$ is the language model

Task: given trained LM and noise model, find most possible intended words
Solution: search algorithm

        - Fairly same for different applications
- Training conditional probabilities often requires *input/output pairs* which are often limited:
    - Misspelled words with their corrections, transcribed speech, translated text
- But language models can be trained on *huge unannotated* corpora: a better model; can help improve overall performance
- Assume we have a way to compute $P(x|y)$ and $P(y)$. Can we do the following:
    - Consider all possible intended words y
    - For each y, compute $P(x|y)P(y)$
    - Return the y with the highest $P(x|y)P(y)$ value

  No, we can't. Without constraints, there are (nearly) infinite number of possible y s.

Lecture 8 - Spelling correction, Edit Distance and EM

need algorithms for acquiring noisy channel models from data, and in particular the noise model.

Assumes:
- we have a large dictionary of real words;
- we don't split or merge 'words' in the input string; and
- we only consider corrections that differ by a single character (insertion, deletion, or substitution) from the non-word.

Then we can do the following to correct each non-word x_i

- Generate a list of all words y_i that differ by 1 character from x_i.
- Compute $P(\vec{x}|\vec{y})P(\vec{y})$ for each $\vec{y}$ and return the $\vec{y}$ with highest value.

Simple noise model:

Assume that the typed *character* x_i depends only on intended character y_i (ignoring context). Ignoring word, single characters are independent

So, substitution $\mathrm{o} \rightarrow \mathrm{e}$ is equally probable regardless of whether the word is effort, spoon, or whatever.

Then for each *observed sequence* $\vec{x}$, made up of a sequence of characters (including spaces) $x_1, \ldots x_n$, we have

$$P(\vec{x}|\vec{y}) = \prod_{i=1}^{n} P(x_i|y_i)$$

For example, $P(\mathrm{no}|\mathrm{not}) = P(\mathrm{n}|\mathrm{n})P(\mathrm{o}|\mathrm{o})P(\text{-}|\mathrm{t})$

Get P(xi | yi) (for each character pair)

- Simply count how many times each character (including empty character for del/ins) was used in place of each other character.
- The table of these counts is called a **confusion matrix**.
- Then use MLE or smoothing to estimate probabilities.
  + smoothing
- Learn $P(x|y)$ from a corpus of character alignments.

Confusion Matrix

	Actually Positive (1)	Actually Negative (0)
Predicted Positive (1)	True Positives (TPs)	False Positives (FPs)
Predicted Negative (0)	False Negatives (FNs)	True Negatives (TNs)

y\x	A	B	C	D	E	F	G	H
A	...	1	0	2	5	5	1	3
B	0	136	1	0	3	2	0	4
C	1	6	111	5	11	6	36	5
D	1	17	4	157	6	11	0	5
E	2	10	0	1	98	27	1	5
F	1	0	0	1	9	73	0	6
G	1	3	32	1	5	3	127	3
H	2	0	0	0	3	3	0	4
...	...	...	...	...	...	...	...	...

- We saw G when the intended character was C 36 times.

actual: n o - m u u c h e f f e r t
intended: n o t m - u c h e f f o r t

Require lots of effort to annotate individual correspondence

want a more general algorithm that can compute edit distances between any two arbitrary words:

Problems

1. Independence assumption is unrealistic.
2. Assumption restricting possible intended words is unrealistic.
3. We may not have a corpus of alignments!

Now: Approach that solves problems 1 and 2: edit distance

need algorithms for acquiring noisy channel models from data, and in particular the noise model.

Edit Distance

- The task: find the **optimal character alignment** between two words (the one with the fewest character changes: the **minimum edit distance** or MED)
- Example: *if all changes count equally,* MED(stall, table) is 3:

```
S T A L L
  T A L L      deletion
  T A B L      substitution
  T A B L E    insertion
```

  - Written as an alignment:

```
S T A L L
d | | s | i
- T A B L E
```

- There may be multiple best alignments
  - And lots of *non-optimal* alignments
- For now, all costs are equal: $cost(ins) = cost(del) = cost(sub) = 1$ In the following example, we'll assume cost(ins) = cost(del)= 1 and cost(sub) = 2.
  - But we can choose whatever costs we want! They can even depend on the particular characters involved

Finding an optimal alignment

- Brute force doesn't scale well
  - The number of alignments to check grows exponentially with the length of the sequences
- Instead we will use *dynamic programming* algorithm
  - Strings of length n and m require $O(mn)$ time and $O(mn)$ space.

Chart

TO

FROM		T	A	B	L	E
	0	←1	←2	←3	←4	←5
S	↑1					
T	↑2					
A	↑3					
L	↑4					
L	↑5					?

Step1: initialize (As left)

Step2: from top, top left, left

Step 3: what character have added

Step 4: compare 3 costs

Step 5: store lowest & pointer

- Chart$[i, j]$ stores two things:
  - $D(\text{stall}[0..i], \text{table}[0..j])$: the MED of substrings of length i, j Minimal edit distance
  - **Backpointer(s):** which sub-alignment(s) used to create this one.

Deletion:	Move down	Cost =1
Insertion:	Move right	Cost=1
Substitution:	Move down and right	Cost=2 (or 0 if the same)

Sum costs as we expand out from cell (0,0) to populate the entire matrix

		T	A	B	L	E
	0	←1				
S	↑1	←↖↑2				
T	↑2	↖1				
A	↑3					
L	↑4					
L	↑5					

e.g. Step 3:

From top: Added T Deletion

From top left: Added both T, nothing

From Left: added T, Insertion

- Now compute $D(\text{ST}, \text{T})$. Take the min of three possibilities:
  - D(ST, -) + cost(ins) = 2 + 1 = 3.
  - D(S, T) + cost(del) = 2 + 1 = 3.
  - D(S, -) + cost(ident) = 1 + 0 = 1.

		T	A	B	L	E
	0	←1	←2	←3	←4	←5
S	↑1	←↖↑2	←↖↑3	←4	←5	←6
T	↑2	↖1	←2	←3	←4	←5
A	↑3	↑2	↖1	←2	←3	←4
L	↑4	↑3	↑2	←↖↑3	↖2	←3
L	↑5	↑4	↑3	←↖↑4	↖↑3	←↖↑4

Total cost = 4

Backtrack:
Move UP: deletion of original LEFT character (box)
Move LEFT: insertion of original TOP character (circle)
MOVE UP-LEFT: replacement / unchanged

Recover from "TABLE":

\S T A VB L VE \L
T A B L. E

Advantage less effort in annotation (to get the costs in catch22)

- Previously, we needed hand annotations like:

actual: n o - m u u c h e f f e r t
intended: n o t m - u c h e f f o r t

- Now, our annotation requires less effort:

actual: no muuch effert
intended: not much effort

- Computing distances and/or alignments between arbitrary strings can be used for:
  - Spelling correction
  - Morphological analysis (which words are likely to be related?)
  - Other fields entirely (e.g. comparing DNA sequences in biology)
  - Related algorithms are also used in speech recognition and time series data mining

Catch-22

- In our examples, we used costs of 1 (insertion and deletion) and 2 (substitution) to compute alignments
- We actually want to compute our alignments using the costs from our noise model. The most probable alignment under that model is what we are interested in
- But alas, until we have the alignments, we can't estimate the noise model

General Formulation

- This sort of problem actually happens a lot in NLP (and ML)
- We have some probabilistic model and want to estimate its **parameters** (here, the costs)
- The model also contains variables whose value is unknown (here, the correct character alignments)
- We would be able to estimate the parameters if we knew the values of the variables
  - And conversely, we would be able to infer the values of the variables if we knew the values of the parameters

Expectation-Maximisatio earn the costs for the operations without annotated data

- Problems of this type can often be solved using a version of EM
  1. Initialise parameters to arbitrary values (e.g. set all costs to 1)
  2. Using these parameters, compute optimal values for variables (run MED to get alignments)
  3. Now using those alignments, **recompute** the parameters
     a. Just pretend that alignments are hand annotations
     b. Estimate parameters as from annotated corpus
  4. Repeat steps 2 and 3 until parameters stop changing

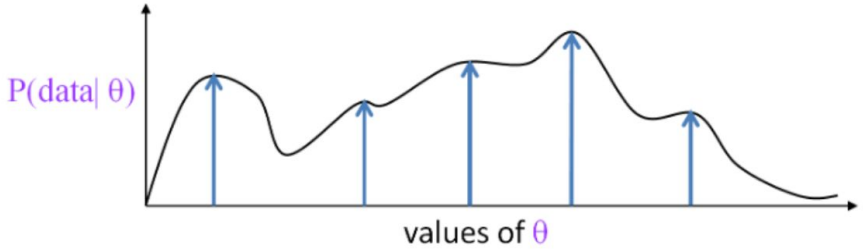
EM vs. Hard EM

- What we have just described is actually "Hard EM" (meaning: no soft/fuzzy decisions)
- Step 2 of true EM does not choose **optimal** values for variables, instead computes **expected** values
- **True EM** is guaranteed convergence to a local optimum of the **likelihood function**
  - Hard EM also converges but not to anything nicely defined mathematically
  - However it's usually easier to compute and may work fine in practice

Likelihood Function

- Let's call the parameters of our model θ
- For any value of θ, we can compute the probability of our dataset $P(data|\theta)$. This is the **likelihood**!
  - If our data includes hand-annotated character alignments, then $P(data|\theta) = \prod_{i=1}^{n} P(x_i \mid y_i)$
    = each character independent
  - If the alignments a are laten um over possible alignments: $P(data|\theta) = \sum_a \prod_{i=1}^{n} P(x_i| y_i, a)$
- The likelihood $P(data|\theta)$ is a function of θ, and can have multiple local optima

- Schematically (but θ is really multidimensional)



- EM will converge to one of these local optima; hard EM won't necessarily
- Neither is guaranteed to find the global optimum!

Lecture 9 - **Text Classification** & Naive Bayes

Text classification

- We might want to categorise the *content* of the text:
  - Spam detection (binary: spam or not)
  - Sentiment analysis (binary or multiway)
    - Movie, restaurant, or product reviews (positive/negative or 1-5 stars)

Rules & joint prob table:

$$P(C,Q) = P(C)P(Q\,|\,C)$$

$$P(Q\,|\,\neg C) = \frac{P(Q,\neg C)}{P(\neg C)}$$

	Q	R	
C	$1/12$		$1/2$
$\neg C$	$1/4$		$1/2$
	$1/3$	$2/3$	

- N-gram models can sometimes be used for classification but
  - For many tasks, sequential relationships between words are largely irrelevant: we can just consider the document as a bag of words
    - $Document \rightarrow [(word,\ count)]$
    - Some normalisation can be done beforehand, such as case conversion
  - On the other hand, we may want to include other kinds of features (e.g. PoS tags) that N-gram models don't include
-

$$\begin{aligned}
\hat{s} &= \arg\max_{s\in S} P(s|\vec{f}) \\
&= \arg\max_{s\in S} \frac{P(\vec{f}|s)P(s)}{P(\vec{f})} && \text{Bayes} \\
&= \arg\max_{s\in S} P(\vec{f}|s)P(s) && P(\vec{f}) \text{ is fixed} \\
&\approx \arg\max_{s\in S} P(s)\prod_{j=1}^{n} P(f_j|s) && \text{cond. independence}
\end{aligned}$$

Naive Bayes

- Given document d and a set of categories C (say, spam/not-spam), we want to assign d to the most probable category $\hat{c}$

$$\hat{c} = argmax_{c\in C} P(c|d) = argmax_{c\in C} \frac{P(d|c)P(c)}{P(d)} = argmax_{c\in C} P(d|)P(c)$$

$$\begin{aligned}
\hat{c} &= \underset{c\in C}{\text{argmax}}\, P(c|d) \\
&= \underset{c\in C}{\text{argmax}}\, \frac{P(d|c)P(c)}{P(d)} \\
&= \underset{c\in C}{\text{argmax}}\, P(d|c)P(c)
\end{aligned}$$

- Just as in spelling correction, we need to define $P(d|c)$ and $P(c)$
  - $P(c)$ is the **prior probability** of class c before observing any data
    - Simply estimated by MLE: $\widehat{P}(c) = \frac{N_c}{N}$
    - In other words, the proportion of training documents belonging to class c

Modelling $P(d|c)$ (feature probabilities)

- We represent each document d as the set of features (words) it contains: $f_1, f_2, ..., f_n$
  - So $P(d|c) = P(f_1, f_2, ..., f_n|c)$
- As in language models, we cannot accurately estimate $P(f_1, f_2, ..., f_n|c)$ due to sparse data
- So, we make a **Naive Bayes assumption**: features are conditionally independent given the class: $P(f_1, f_2, ..., f_n|c) \approx P(f_1|c)P(f_2|c)...P(f_n|c)$
- That is, the probability of a word occurring depends *only* on the class
  - Not on which words occurred before or after (as in N-grams)
  - Or even which words occurred at all
- Effectively, we only care about the **count of each feature** in each document

Given a document with features $f_1, f_2, \ldots f_n$ and set of categories C, choose the class $\hat{c}$ where

$$\hat{c} = \underset{c\in C}{\text{argmax}}\, P(c)\prod_{i=1}^{n} P(f_i|c)$$

- $P(c)$ is the **prior probability** of class c before observing any data.
- $P(f_i|c)$ is the probability of seeing feature f_i in class c.

Calculating the feature probabilities

$P(f_i|c)$ is normally estimated with simple smoothing:

$$\hat{P}(f_i|c) = \frac{count(f_i, c) + \alpha}{\sum_{f \in F} (count(f, c) + \alpha)}$$

All words in class c + alpha*F

- $count(f_i, c)$ is the number of times f_i occurs in class c
- F is the set of possible features
- α is the smoothing parameter, optimised on held-out data

e.g.

	the	your	model	cash	Viagra	class	account	orderz	spam?
doc 1	12	3	1	0	0	2	0	0	-
doc 2	10	4	0	4	0	0	2	0	+
doc 3	25	4	0	0	0	1	1	0	-
doc 4	14	2	0	1	3	0	1	1	+
doc 5	17	5	0	2	0	0	1	1	+

$\hat{P}(\text{your}|+) = \frac{(4+2+5+\alpha)}{(\text{all words in + class})+\alpha F} = (11+\alpha)/(68+\alpha F)$

Here F = 8

Alternative features

- Use only **binary** values for f_i: did this word occur in d or not?
- Use only a subset of the vocabulary for F
  - Ignore **stopwords** (function words and others with little content)
  - Choose a small task-relevant set (e.g. using a sentiment lexicon)

Choosing features

  - Can be tricky:
    - E.g. sentiment analysis might need domain-specific non-sentiment words: such as *quiet* for computer product reviews
    - And for other tasks, stopwords might be very useful features:
      - E.g. people with schizophrenia use more 2nd-person pronouns, those with depression use more 1st-person
    - Probably better to use too many irrelevant features than not enough relevant ones
- Use more complex features (bigrams, syntactic features, morphological features)

Parsing stuffs. = subwords

Costs and linearity

- Multiplying large numbers of small probabilities together is problematic, thus we use costs (negative log probability) again

Avoid numeric underflow

- In which case, we look for the **lowest cost** overall
- Naive Bayes then:

$$\hat{c} = argmin_{c \in C}(-logP(c) + \sum_{i=1}^{n} -logP(f_i|c)$$

- This amounts to classification using a linear function (in log space) of the input features
  - So Naive Bayes is called a **linear classifier**
  - As is logistic regression

Review of Naive Bayes

- Advantages

Could deal with missing & sparse data since generative model

  - Very easy to implement
  - Very fast to train, and to classify new documents (good for huge datasets)
  - Doesn't require as much training data as some other methods (good for small datasets)
  - Usually works reasonably well
  - This should be your baseline method for any classification task
- Disadvantages
  - Naive Bayes assumption is Naive
    - Consider the following categories: *travel, finance, sport*
    - Are the following features independent given the category: *beach, sun, ski, snow, pitch, palm, football, relax, ocean*?
      - No! Given *travel*, seeing *beach* makes *sun* more likely, but *ski* less likely

Solution: still use naive when correlated:

- Defining finer-grained categories might help (beach travel vs ski travel), but we usually do not want to
- In short, features are not usually independent given the class

- Accuracy of classifier can sometimes still be OK, but it will be highly **overconfident** in its decisions

Since naive

- For example, Naive Bayes sees 5 features that all point to class 1, and treats them as five independent sources of evidence
- It's like asking 5 friends for an opinion when some got theirs from each other

Evaluation: accuracy is sensible, but notice class imbalance

Hypothesis testing by BAYESIAN FACTOR (compare likelihood): $K = \frac{\Pr(D|M_1)}{\Pr(D|M_2)}$

++ Semi-supervised learning with NB

Self training = Hard EM:

1. Train NB on labeled data alone
2. Predict labels on on unlabelled data
3. Re-estimate NB (in the usual way), but now using also self-labelled data

Would Learned incorrect association between new token and label

Since:

Unlab doc 2:

$\hat{P}(\text{spam}|d) \approx 0.53$

The model was not confident in its prediction, while self-training this label as equivalent to 'gold standard'

- Advantages:
  - Simplicity and applicable to any classifier (not only NB)
- Disadvantages:
  - Does not account for uncertainty of a classifier
  - No theoretical motivation (kind of…)
- To make it work, well requires
  - discarding low-confidence predictions
  - curriculum (start with examples similar to labeled data)

Soft EM: = soft decision boundary

		Bayes	your	model	cash	Viagra	class	orderz	spam?
labeled data	lab doc 1	0	1	3	0	0	2	0	-
	lab doc 2	0	2	0	4	0	0	0	+
	lab doc 3	0	2	2	0	0	3	0	-
	lab doc 4	0	3	2	1	3	0	1	+
	lab doc 5	0	1	0	2	0	0	1	+
unlabeled data	unl doc 2	2 x 0.53	2 x0.53	0	0	0	0	0	+ (.53)
		2 x 0.47	2 x 0.47	0	0	0	0	0	- (.47)

2 labels for same datapoint

Unlab doc 2:

$\hat{P}(\text{spam}|d) \approx 0.53$

Use soft label: 0.53 of the data point is labelled as "+", 0.47 as "-"

1. Train NB on labeled data alone
2. Make soft prediction on on unlabelled data ("E-step")
3. Recompute NB parameters using the soft counts

We defined the method algorithmically, but it can be shown to optimize the likelihood of observed data (i.e. a combination labelled and unlabeled portions)

$\hat{P}(\text{your}|+) = (6 + 2 \times 0.53 + \alpha)/(20 + 4 \times 0.53 + \alpha * F)$

$\hat{P}(\text{your}|-) = (3 + 2 \times 0.47 + \alpha)/(13 + 3 \times 0.47 + \alpha * F)$

$\hat{P}(\text{Bayes}|+) = (2 \times 0.53 + \alpha)/(20 + 4 \times 0.53 + \alpha * F)$

$\hat{P}(\text{Bayes}|-) = (2 \times 0.47 + \alpha)/(13 + 4 \times 0.47 + \alpha * F)$

$\hat{P}(\text{spam}) = \frac{3 + 0.53}{5 + 1}$

This is just for one data point

Using hand generated features!!!
Feature includes class!!!

-

Adding multiple feature types (e.g. words and morphemes) often leads to even stronger correlations between features

Maximum Entropy Classifiers

- Most commonly, **multinomial logistic regression**
  - *Multinomial* if more than two possible classes
- Also called: log-linear model, one-layer neural network, single neuron classifier, etc.

Like Naive Bayes, MaxEnt assigns a document d to class $\hat{c}$, where

$$\hat{c} = \underset{c \in C}{\operatorname{argmax}} P(c|d)$$

-

Unlike Naive Bayes, we do not apply Bayes' Rule. Instead, we model $P(c|d)$ directly. (discriminative model)

Features

- Like Naive Bayes, MaxEnt models use **features** we think will be useful for classification
- However, features are treated differently in two models:
  - Naive Bayes
    - Features are *directly observed* (e.g. words in doc)
    - No difference between features and data
  - Maximum Entropy
    - We will use $\bar{x}$ to represent the observed data
    - Features are *functions* that depend on both observations $\bar{x}$ and class c
- For example, if we have three classes, our features will always come in groups of three. Imagine 9 features:

f_1 :	contains('ski') & $c = 1$	$w_1 = 1.2$
f_2 :	contains('ski') & $c = 2$	$w_2 = 2.3$
f_3 :	contains('ski') & $c = 3$	$w_3 = -0.5$
f_4 :	link_to('expedia.com') & $c = 1$	$w_4 = 4.6$
f_5 :	link_to('expedia.com') & $c = 2$	$w_5 = -0.2$
f_6 :	link_to('expedia.com') & $c = 3$	$w_6 = 0.5$
f_7 :	num_links & $c = 1$	$w_7 = 0.0$
f_8 :	num_links & $c = 2$	$w_8 = 0.2$
f_9 :	num_links & $c = 3$	$w_9 = -0.1$

=> e.g. document contains 'ski' , without link_to(), and num_links = 6

  - Travel: $\sum_i w_i f_i(\vec{x}, c = 1) = 1.2 + (0.0)(6) = 1.2$. (w1*f1 + w7*f7)
  - Sport: $\sum_i w_i f_i(\vec{x}, c = 2) = 2.3 + (0.2)(6) = 3.5$.
  - Finance: $\sum_i w_i f_i(\vec{x}, c = 3) = -0.5 + (-0.1)(6) = -1.1$.
  - Note the format: the 'actual' feature AND class
  - Training docs from class 1 that contain *ski* will have f_1 active
  - Training docs from class 2 that contain *ski* will have f_2 active
  - Etc.
- Each feature f_i has a real-value **weight** w_i learned in training

Classification with MaxEnt

Choose the class that has highest probability according to

$$P(c|\vec{x}) = \frac{1}{Z} \exp\left(\sum_i w_i f_i(\vec{x}, c)\right)$$

where the normalization constant $Z = \sum_{c'} \exp(\sum_i w_i f_i(\vec{x}, c'))$

$$\begin{aligned} \log P(y|\vec{x}) &= \log \frac{1}{Z} \exp\left(\vec{w} \cdot \vec{f}(\vec{x}, y)\right) \\ &= \log \exp\left(\vec{w} \cdot \vec{f}(\vec{x}, y)\right) + \log \frac{1}{Z} \\ &= \vec{w} \cdot \vec{f}(\vec{x}, y) - \log Z \end{aligned}$$

= linear combination of the feature values
Log linear

- Inside brackets is just a dot product: $\vec{w} \cdot \vec{f}$.
- And $P(c|\vec{x})$ is a **monotonic functio** this dot product.
- So, we will end up choosing the class for which $\vec{w} \cdot \vec{f}$ is highest
- Realise that the normalisation constant Z is not required for classification purposes

Feature Templates

- In practice, features are usually defined using **templates**

  $contains(w)$ & c

  $headerContains(w)$ & c

  $headerContains(w)$ & $linkInHeader$ & c
- NLP tasks often have few templates for 1,000s and 10,000s of features!

Training the Model

- Given annotated data, choose weights that make the labels most probable under the model
- That is, given examples $x^{(1)}, \ldots, x^{(N)}$ with labels $c^{(1)}, \ldots, c^{(N)}$, choose $\hat{w} = \underset{\vec{w}}{\operatorname{argmax}} \sum_j \log P(c^{(j)}|x^{(j)})$
- Called **conditional maximum likelihood estimation (CMLE)**
  - Like MLE, CMLE will overfit so we use tricks (**regularisation**) to avoid that

Very large data size, discriminative model > generative mode

Drawback of MaxEnt

- Supervised CMLE in MaxEnt is not so easy
  - Requires multiple iterations over the data to gradually improve weights (using *gradient ascent*)
  - Each iteration computes $P(c^{(j)}|x^{(j)})$ for all j, and each possible $c^{(j)}$
  - This can be time-consuming, especially if there are a large number of classes and/or thousands of features to extract from each training sample

Relation to Naive Bayes:

- Recall, Naive Bayes is also a linear classifier, and can be expressed in the same form
- Should the features be actually independent (will never happen), they would converge to the same solution as the amount of training data increases

NB more robust than MaxEnt:

- Imagine that in training there is one very frequent predictive feature
  - E.g., in training setiment data contained emoticons but not at test time
- The model can quickly learn to rely on this feature = overfitting
  - model is confident on examples with emoticons
  - the gradient on these examples gets close to zero
  - the model does not learn other features
- In MaxEnt, **a feature weight will depend on the presence of other predictive features** SO need regularization
- Naive Bayes will rely on all features
  - The weight of a feature is not affected by how predictive other features are
- This makes NB more robust that (**basic**) MaxEnt when test data is (distributionally) different from train data

++ MLE training: gradient descent

MaxEnt training: gradient descent

$\vec{w} \leftarrow \text{Random}()$ — Random initialization (e.g., from a Gaussian distribution)

repeat

$\vec{w} \leftarrow \vec{w} + \eta \cdot \nabla_w \sum_{j=1}^{N} \log P(c^{(j)}|x^{(j)})$

until Converged()

Learning rate: a scalar regulating how much you update on every example

Common strategy: finish when the performance on the development set stops improving (or after a fixed number of iterations)

Mini batch:

Choosing a "**batch**": Indexes of a random subset of examples (e.g., choose 10 random examples)

$\vec{w} \leftarrow \text{Random}()$

repeat

$B \leftarrow \text{RandomSubset}([1, \ldots, N])$

$\vec{w} \leftarrow \vec{w} + \eta \cdot \nabla_w \sum_{j \in B} \log P(c^{(j)}|x^{(j)})$

until Converged()

Sum only over examples in the current batch

Gradient Descent

Take log => linear combination

$$\frac{d}{dw_l} \log P(c^{(j)}|\vec{x}^{(j)}) = \text{(I)} - \text{(II)}$$

[]: 0 or 1 — Derivative of bias log Z

$$= \underbrace{[k = c^{(j)}] \cdot f_l(\vec{x}^{(j)}, k)}_{\text{(I)}} - \underbrace{P(c = k|x^{(j)}) \cdot f_l(\vec{x}^{(j)}, k)}_{\text{(II)}}$$

$$= \left([k = c^{(j)}] - P(c = k|x^{(j)})\right) f_l(\vec{x}^{(j)}, k)$$

= Expectation of the feature, under the model distribution

Close to zero if the classifier confidently predicts the correct class

$$P(c = k|x^{(j)}) \approx \begin{cases} 1 & \text{if } k = c^{(j)} \\ 0 & \text{otherwise} \end{cases}$$

If the classifier is already confident, gradient is close to 0 and no learning is happening

Lecture 11 Morphology

Morphology = structure of words
English morphology not rich

Construct words by: structure of word depends only on that wor

Stems (bit in dictionary)
Affixes
prefix, before stem
suffix, after stem
infix, middle of stem
circumfix, 'reduction' of the stem

Combine them:

Inflection (stem + grammar affix): no change to grammatical category
(*walk* → *walking*) V -> V

Derivation (stem + grammar affix): change to grammatical category
(*combine* → *combination*) V -> N

Compounding (stems together): *doghouse*

Cliticization: *I've, we're, he's*

Morphology can be concatenative
or non-concatenative (e.g. templatic morphology as in Arabic)

Morphological parsing: the problem

- English has concatenative morphology. Words can be made up of a main stem plus one or more affixes carrying grammatical information.

Surface form:	cats	walking	smoothest
Lexical form:	cat+N+PL	walk+V+PresPart	smooth+Adj+Sup

- Morphological parsing is the problem of extracting the lexical form from the surface form. (For ASR, this includes identifying the word boundaries.)
- We should take account of: (Challenging because of...)
  - Irregular forms (e.g. goose → geese) And ambiguit
  - Systematic rules (e.g. 'e' inserted before suffix 's' after s,x,z,ch,sh: fox → foxes, watch → watches)
  - Things that look like affixes but aren't (*reactive* vs. *protect*)
  - Blocking: semi-productivity of morphological rules:
    N+*ful* ↦ Adj (*graceful*, *playful*, . . .),
    but **intelligenceful* (intelligent), **gloryful* (glorious). . .
    Wrong

Parsing and generation

Parsing here means going from the surface to the lexical form.

Eg. foxes → fox +N +PL.

Generation is the opposite process: fox +N +PL → foxes.
It's helpful to consider these two processes together.

Either way, it's often useful to proceed via an intermediate form, corresponding to an analysis in terms of morphemes (= minimal meaningful units) before orthographic rules are applied.

Surface form:	foxes
Intermediate form:	fox ^ s #
Lexical form:	fox +N +PL

(^ means morpheme boundary, # means word boundary.)

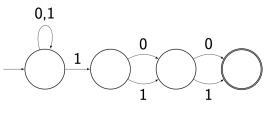
NB. The translation between surface and intermediate form is exactly the same if 'foxes' is a 3rd person singular verb!

nondeterministic finite state automaton (NFA)

state can have more than one outgoing arc

An ϵ-NFA allows an input (in parsing) or output (in generation) defined by an arc to be the empty string.

E.g., $(0|1)^*1(0|1)^2$ is captured by the following:

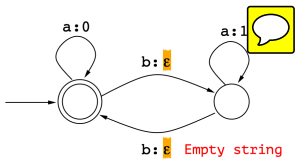


FST for Morphological Parsing:

Finite-state Transducers

- ϵ-NFAs over an input alphabet Σ can capture transitions that (optionally) produce *output* symbols (over a possibly different alphabet Π).

input alphabet $\Sigma = \{a, b\}$
output alphabet $\Pi = \{0, 1\}$:

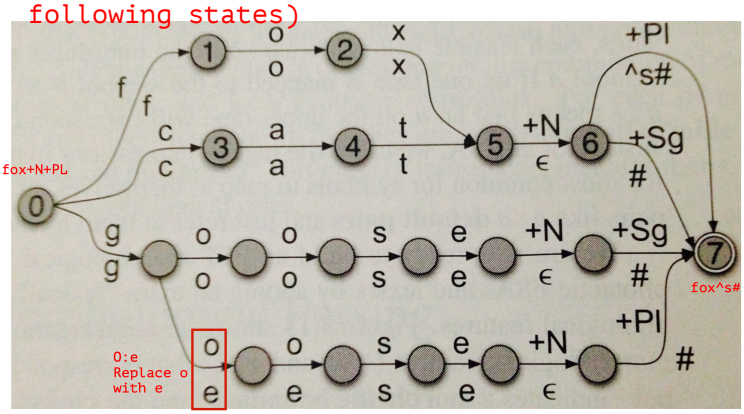


- Such a thing is called a finite state transducer.
- In effect, it specifies a (possibly multi-valued) translation from one regular language to another:
  - $abba \mapsto 00$, $aababa \mapsto 0010 \ldots$
- More than one output can be possible (e.g., an arc labelled $a : 0, 1$)

Formal definition for FSTs

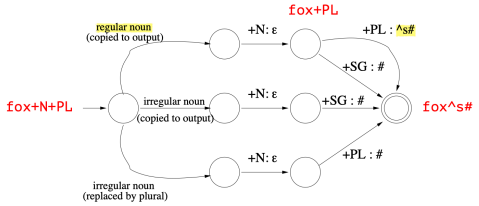
- A finite state transducer T with inputs from Σ and outputs from Π consists of:
  - states Q, S (for start), F (for 'finish')
  - a transition relation $\Delta \subseteq Q \times (\Sigma \cup \{\epsilon\}) \times (\Pi \cup \{\epsilon\}) \times Q$
- This defines a many-step transition relation $\hat{\Delta} \subseteq Q \times \Sigma^* \times \Pi^* \times Q$
  - $(q, x, y, q') \in \hat{\Delta}$ means "starting from state q, the input string x can be translated into the output string y, ending up in state q'." (Details omitted.)
- A finite state transducer can be run in either direction! From T you can obtain another transducer $\overline{T}$ by swapping inputs and outputs.

If facing null edge,
Could choose: go trough null edge /
go trough another valid edge
(Check if # transition out of state exist in following states)



Stage 1: From lexical to intermediate form

- Convert 'fox+N+PL' to 'fox ^ s # '
  . . . while taking account of irregular forms like goose/geese.
- We can do this with a transducer of the following schematic form:



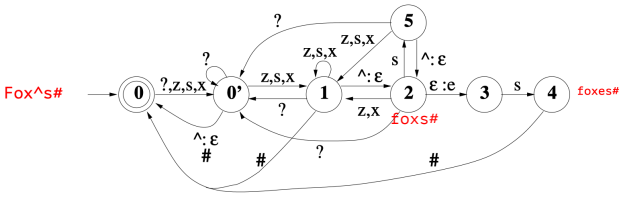
- We treat each of +N, +SG, +PL as a single symbol.
- The 'transition' labelled $+PL : \hat{}s\#$ abbreviates three transitions: $+PL : \hat{}$, $\epsilon : s$, $\epsilon : \#$

Stage 2: From intermediate to surface form

- To convert a sequence of morphemes to surface form, we apply a number of orthographic rules:
  - E-insertion: Insert e after s,z,x,ch,sh before a word-final morpheme -s.
    (fox → foxes)
  - E-deletion: Delete e before a suffix beginning with e,i.
    (love → loving)
  - Consonant doubling: Single consonants b,s,g,k,l,m,n,p,r,s,t,v are doubled before suffix -ed or -ing.
    (beg → begged)
- We shall consider a simplified form of E-insertion, ignoring ch,sh.
- This rule is oblivious to whether -s is a plural noun suffix or a 3rd person verb suffix!
  fox ^ s # → foxes

Fu yin

A transducer for E-insertion (adapted from J+M)



Here ? may stand for any symbol except z,s,x,^,#.

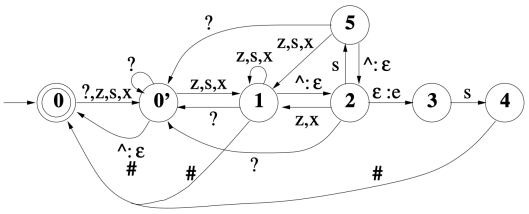
(Treat # as a 'visible space character'.)

At a morpheme boundary following z,s,x, we arrive in State 2.

If the ensuing input sequence is s#, our only option is to go via states 3 and 4.
Note that there's no #-transition out of State 5.

State 5 allows e.g. 'ex^service^men#' to be translated to 'exservicemen'.

Example: 10 ways to parse asses!



On the input string 'asses', 10 transition sequences are possible!

From surface to intermediate
a:b is treated backward,
While inserting null

- $0 \xrightarrow{a} 0' \xrightarrow{s} 1 \xrightarrow{s} 1 \xrightarrow{\epsilon} 2 \xrightarrow{e} 3 \xrightarrow{s} 4$, output ass^s
- $0 \xrightarrow{a} 0' \xrightarrow{s} 1 \xrightarrow{s} 1 \xrightarrow{\epsilon} 2 \xrightarrow{e} 0' \xrightarrow{s} 1$, output ass^es
- $0 \xrightarrow{a} 0' \xrightarrow{s} 1 \xrightarrow{s} 1 \xrightarrow{e} 0' \xrightarrow{s} 1$, output asses
- $0 \xrightarrow{a} 0' \xrightarrow{s} 1 \xrightarrow{\epsilon} 2 \xrightarrow{s} 5 \xrightarrow{\epsilon} 2 \xrightarrow{e} 3 \xrightarrow{s} 4$, output as^s^s
- $0 \xrightarrow{a} 0' \xrightarrow{s} 1 \xrightarrow{\epsilon} 2 \xrightarrow{s} 5 \xrightarrow{\epsilon} 2 \xrightarrow{e} 0' \xrightarrow{s} 1$, output as^s^es
- $0 \xrightarrow{a} 0' \xrightarrow{s} 1 \xrightarrow{\epsilon} 2 \xrightarrow{s} 5 \xrightarrow{e} 0' \xrightarrow{s} 1$, output as^ses
- Four of these can also be followed by $1 \xrightarrow{\epsilon} 2$ (output ^).

If facing null edge,
Could choose: go trough null edge /
go trough another valid edge
(Check if # transition out of state exist in following states)

Putting it all together

- FSTs can be cascaded: output from one can be input to another.
- For generation:
  - Stage 1: lexical to intermediate form; then
    Stage 2 (orthographic rules): intermediate to surface form.

  This is typically deterministic (the lexical form yields unique surface form), even though the transducers make use of non-determinism along the way.
- Running the same cascade backwards yields parsing: surface to lexical form.
- Because of ambiguity, this process is frequently non-deterministic:
  e.g. foxes might be analysed as fox+N+PL or fox+V+Pres+3SG.
- Such ambiguities are not resolved by morphological parsing itself: left to a later processing stage.

Porter Stemmer

a lexicon-free method for getting the stem of a given word:
Sometimes need to extract the stem in a very efficient fashion (such as in IR)

- ATIONAL → ATE (e.g., relational → relate)
  ING → ϵ if stem comtains a vowel (e.g. motoring → motor)
  SSES → SS (e.g., grasses → grass)

Would make errors: e.g. organization -> organ

Learning Morphological Parsers: supervised & unsupervised

Sometimes ambiguit y can't be resolved

- NFTs aren't designed to resolve morphological ambiguities
  (no representation of context).
- Whether foxes is fox+N+PL or fox+V+3SG depends on words in its neighbourhood. .

Lecture 8 - Part-of-Speech Tagging and HMMs

Sequence Labelling (Tagging)

- It is often the first step towards any syntactic analysis (which in turn, is often useful for semantic analysis)
- **Named Entity Recognition**: labels words as belonging to *persons*, *organisations*, *locations*, or none of the above

  Barack/PER Obama/PER spoke/NON from/NON the/NON White/LOC House/LOC today/NON ./NON

- **Information Field Segmentation** given specific type of text (classified advert, bibliography entry, etc.), identify which words belong to which 'fields' (prize/size/location, author/title/year)

  3BR/SIZE flat/TYPE in/NON Bruntsfield/LOC ,/NON near/LOC main/LOC roads/LOC ./NON Bright/FEAT ,/NON well/FEAT maintained/FEAT

- In sequence labelling, deciding the correct label depends on
  - The word to be labeled
  - the labels of surrounding words
  - Hidden Markov Model combines these sources of information probabilistically

Parts of Speech

- **Open-Class Words** (or **Content Words**)
  - Nouns, verbs, adjectives, adverbs
  - Mostly *content-bearing*: they refer to objects, actions and features *in the world*
  - *Open* class, since there is no limit to what these words are, new words are added all the time (e.g. selfie, Brexit, omnishambles)
- **Close-Class Words** (or **Function Words**)
  - Pronouns, determiners, prepositions, connectives
  - There are a limited number of these
  - Mostly *functional*: to *tie* the concepts of a sentence together
  - New ones are rare
    - So far none of the attempts to introduce new gender-neutral pronouns have gotten much traction

Do & be: verb but closed class

ALSO Auxiliary Verb: can, will, must, may, is, has, does, shall

Enumerator	(e)	one, three, first, second, eighteenth

- The **number** of parts of speech (tags) to have is both linguistic and also a practical consideration
  - Do you want to distinguish between proper nouns (names) and common nouns?
  - Singular and plural nouns?
  - Past and present tense verbs?

- Morphologically rich (e.g. Turkish) languages often have compound morphosyntactic tags: Noun + A3sg + P2sg + Nom
  - Hundreds or thousands of possible combinations!

PoS Tagging

- The problem of finding the best tag sequence for a sentence is also called **decoding**
- PoS tagging is **hard** because
  - Ambiguity
    - Glass of water/NOUN vs. water/VERB the plants

CC	coordinating conjunction	PRP	personal pronoun I, you, he, she, it, we they
CD	cardinal number	RB	adverb
DT	determiner	RBR	comparative adverb
IN	preposition	TO	"to"
JJ	adjective	VB	verb base form
JJR	comparative adjective	VBD	verb past tense
MD	modal Verb: Can, should, would, must, may	VBG	verb gerund -ing
NN	singular or mass noun	VBN	verb past participle -ed
NNS	noun, plural	VBP	verb non-3g present
NNP	proper noun Capitalized	VBZ	verb 3sg present third person
NNPS	proper noun, plural		

POS TAGS:

  - wind/VERB down vs. a mighty wind/NOUN (homographs)
- Sparse data
  - Words we haven't seen before (at all, or in this context)
  - Word-Tag pairs we haven't seen before (e.g. if we verb a noun)
- Relevant knowledge for PoS tagging
  - The word itself
    - Some words may only be nouns, e.g. *arrow*
    - Some words are ambiguous, e.g. *like, flies*
    - Probabilities may help, if one tag is more likely than another
  - Tags of surrounding words
    - Two determiners rarely follow each other
    - Two base form verbs rarely follow each other
    - A determiner is almost always followed by an adjective or a noun

A Probabilistic Model for Tagging = walking trough states in a graph

1. Choose a tag conditioned on previous tag (**transition probability**) $P(t_i|t_{i-1})$
2. Choose a word conditioned on its tag (**emission probability**) $P(w_i|t_i)$
   a. Because every state *emits* a word (except <s> and </s>)

- So the model assumes
  - Each tag depends only on previous tag: a bigram (or n-gram) tag model
  - Words are independent **given a tag**
- Transition probability table

$t_{i-1} \backslash t_i$	NNP	MD	VB	JJ	NN	. . .
<s>	0.2767	0.0006	0.0031	0.0453	0.0449	. . .
NNP	0.3777	0.0110	0.0009	0.0084	0.0584	. . .
MD	0.0008	0.0002	0.7968	0.0005	0.0008	. . .
VB	0.0322	0.0005	0.0050	0.0837	0.0615	. . .
JJ	0.0306	0.0004	0.0001	0.0733	0.4509	. . .
. . .	. . .	. . .	. . .	. . .	. . .	. . .

  - Leftmost column is the previous tag
  - Topmost row is the 'next' tag
  - Every row should sum up to 1

- Emission probability table

$t_i \backslash w_i$	Janet	will	back	the	. . .
NNP	0.000032	0	0	0.000048	. . .
MD	0	0.308431	0	0	. . .
VB	0	0.000028	0.000672	0	. . .
DT	0	0	0	0.506099	. . .
. . .	. . .	. . .	. . .	. . .	. . .

trellis

  - Leftmost column is the current tag
  - Topmost column is the emitted word
  - Every row should sum up to 1

- In this model, **joint probability** is defined as

$$P(S,T) = \prod_{i=1}^{n} P(t_i|t_{i-1})P(w_i|t_i) \qquad \sum_{i=1}^{n} -log_2(P(t_i|t_{i-1}))) - log_2(P(w_i|t_i))$$

  - A product of transmission and emission probabilities for each word

e.g.

- First, add begin- and end-of-sentence <s> and </s> Then:

$$p(S,T) = \prod_{i=1}^{n} P(t_i|t_{i-1})P(w_i|t_i)$$
$$= P(\text{DET}|\text{<s>})P(\text{VB}|\text{DET})P(\text{DET}|\text{VB})P(\text{JJ}|\text{DET})P(\text{NN}|\text{JJ})P(\text{</s>}|\text{NN})$$
$$\cdot P(\text{This}|\text{DET})P(\text{is}|\text{VB})P(\text{a}|\text{DET})P(\text{simple}|\text{JJ})P(\text{sentence}|\text{NN})$$

Actual Tagging with Hidden Markov Models (HMM)

FSM view: Given a sequence of words, what is the most probable state path that generated them?

- HMMs are quite similar to what we have seen earlier:
  - **N-gram model:** a model for sequences that also makes a Markov assumption but has no hidden variables

- **Naive Bayes:** a model with hidden variables (the classes) but no sequential dependencies
- **HMM:** a model for sequences with hidden variables

- Find the best tag sequence T for an untagged sentence S : $argmax_T\ P(T|S)$
  - By Bayes' Rule: $argmax_T\ P(T|S) = argmax_T\ P(S|T)P(T$
    - And $P(S|T)P(T) = P(S,T)$
- Brute-force enumeration of all the possible tag sequences takes $O(c^n)$ time for c possible tags and n words in the sentence

- $P(T)$ is the state transition sequence:

$$P(T) = \prod_i P(t_i|t_{i-1})$$

- $P(S|T)$ are the emission probabilities:

$$P(S|T) = \prod_i P(w_i|t_i)$$

The Viterbi Algorithm

- Dynamic programming algorithm to memorise smaller subproblems to save time in return of space to avoid recomputation

Decoding: inferring the hidden variables in a test instance

- Because, like spell correction etc, HMM can also be viewed as a noisy channel model.
  - Someone wants to send us a sequence of tags: $P(T)$
  - During encoding, "noise" converts each tag to a word: $P(S|T)$
  - We try to decode the observed words back to the original tags.

Lecture 13 PoS Tagging & HMMs (Viterbi)

Definition:

- Input: sequences of variable length $\mathbf{x} = (x_1, x_2, \ldots, x_{|x|}), x_i \in \mathcal{X}$
- Output: every position is associated with a label $\mathbf{y} = (y_1, y_2, \ldots, y_{|x|}), y_i \in \{1, \ldots, N\}$

To Resolve ambiguity: model interaction between labels (gesture in different frames) (= **structured prediction**) 1st & 2nd order

A "generative" model, i.e.:

- Model: Introduce a parameterized model of how both words and tags are generated $P(\mathbf{x}, \mathbf{y}|\theta)$ Original joint prob: P(S,T)
- Learning: use a labeled training set to estimate the most likely parameters of the model $\hat{\theta}$
- Decoding: $\hat{\mathbf{y}} = \arg\max_{\mathbf{y}} P(\mathbf{x}, \mathbf{y}|\theta)$

N: tags; M: vocab size

Parameters (to be estimated from the training set):

- Transition probabilities $a_{ij} = P(y^t = j \mid y^{t-1} = i)$, A - [N x N] matrix
- Emission probabilities $b_{ik} = P(x^t = k \mid y^t = i)$, B - [N x M] matrix

Emission

$$P(x^t = k|y^t = i) = b_{ik} = \frac{C_E(i,k)}{\sum_{k'} C_E(i,k')}$$

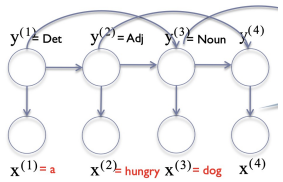
$C_E(i,k)$ is #times word k is emitted by tag i

Trans

$$P(y^t = j \mid y^{t-1} = i) = a_{ij} = \frac{C_T(i,j)}{\sum_{j'} C_T(i,j')}$$

$C_T(i,j)$ is #times tag i is followed by tag j

2nd order:



Stationarity assumption: transition probability does not depend on the position in the sequence t

higher the order, the more zeros, the more important of smoothing

- Transition probabilities $P(y^t|y^{t-1}, y^{t-2})$: [N x N x N] matrix
- Emission probabilities $P(x^t|y^t)$: [N x M] matrix

Enumerate all possible tag sequences: O(T^N); N: sequence length ; T: # of distinct pos tags

Viterbi Decoding $O(T^2N)$ time and $O(TN)$ space.

Retrace the backpointers to recover most possible sequence

a_{ij}	STOP	NN	VB	JJ	RB
START	0	0.5	0.25	0.25	0
NN	0.25	0.25	0.5	0	0
VB	0.25	0.25	0	0.25	0.25
JJ	0	0.75	0	0.25	0
RB	0.5	0.25	0	0.25	0

b_{ik}	time	flies	fast	...	...	...
NN	0.1	0.01	0.01	...	...	...
VB	0.01	0.1	0.01	...	...	...
JJ	0	0	0.1	...	...	...
RB	0	0	0.1	...	...	...

Initialization: $v_i^1 = a_{START,i} b_{i,x^1}, \quad i = 1, \ldots, N;$

$v_j^t = \left(\max_i v_i^{t-1} a_{ij}\right) b_{j,x^t}, \; j = 1, \ldots, N, \; t = 2, \ldots, |x|$

a_{ij}	STOP	NN	VB	JJ	RB
START	0	0.5	0.25	0.25	0
NN	0.25	0.25	0.5	0	0
VB	0.25	0.25	0	0.25	0.25
JJ	0	0.75	0	0.25	0
RB	0.5	0.25	0	0.25	0

b_{ik}	time	flies	fast	...	...	...
NN	0.1	0.01	0.01	...	...	...
VB	0.01	0.1	0.01	...	...	...
JJ	0	0	0.1	...	...	...
RB	0	0	0.1	...	...	...

Initialization: $v_i^1 = a_{START,i} b_{i,x^1}, \quad i = 1, \ldots, N;$

$v_j^t = \left(\max_i v_i^{t-1} a_{ij}\right) b_{j,x^t}, \; j = 1, \ldots, N, \; t = 2, \ldots, |x|$

Final: $v_{STOP}^{|x|+1} = \max_i v_i^{|x|} a_{i,STOP}$

	$time_1$	$flies_2$	$fast_3$	-
NN	0.05 (max, x 0.25)	0.05 x 0.25 x 0.01		
VB	0.0025			
JJ	0			
RB	0			
STOP	-	-	-	

	$time_1$	$flies_2$	$fast_3$	-
NN	0.05	1.25E-4	6.25E-6	-
VB	0.0025	0.0025	6.25E-7	-
JJ	0	0	6.25E-5	-
RB	0	0	6.25E-5 (x 0.5)	-
STOP	-	-	-	6.25E-5 x 0.5

probability of the most probable tagged word sequence

Viterbi Equation:

Initialization: $v_j^1 = a_{START,j} b_{j,x^1}, \quad j = 1, \ldots, N;$

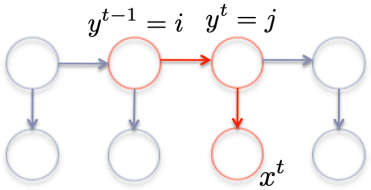
Recomputation: $v_j^t = \left(\max_i v_i^{t-1} a_{ij}\right) b_{j,x^t}, \; j = 1, \ldots, N, \; t = 2, \ldots, |x|$

Final: $v_{STOP}^{|x|+1} = \max_i v_i^{|x|} a_{i,STOP}$

Steps:
1: initialize (a*b)
2: For a word with a tag (e.g. flies with NN)
3: from each prev column element, *a*b, get greatest, store back pointer
4: each *a(stop), record greatest

Equivalently, in the log-space (and in a more generalized form):

Define: $g^t(\mathbf{x}, i, j) = \begin{cases} \log a_{ij} + \log b_{j,x^t} & t = 1, \ldots, |\mathbf{x}| \\ \log a_{ij} & t = |\mathbf{x}| + 1 \end{cases}$



The score is associated with a fragment responsible for one transition and one word generation

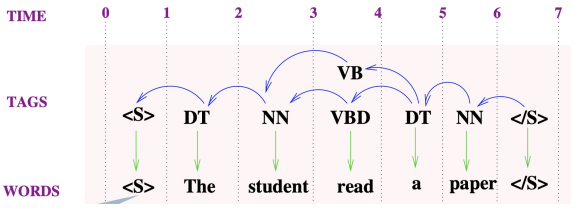
Initialization: $v_j^1 = g^1(\mathbf{x}, START, j) \quad j = 1, \ldots, N;$

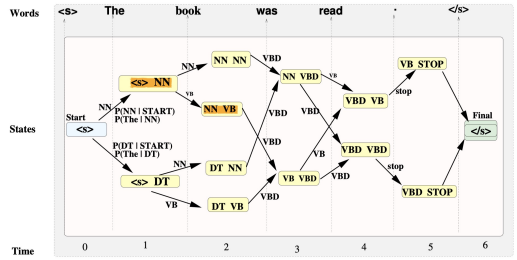
Recomputation: $v_j^t = \max_i \left(v_i^{t-1} + g^t(\mathbf{x}, i, j)\right), \; j = 1, \ldots, N, \; t = 2, \ldots, |x|$

Final: $v_{STOP}^{|x|+1} = \max_i \left(v_i^{|x|} + g^{|x|+1}(\mathbf{x}, i, STOP)\right)$

Process in batch:
1: initialize all (<s>row * emi word col)
2: In one cell: Compare: (PreCol * transCol)
3: get largest/smallest
4: * emiProb

5: Final: (preCol * <\s>Col) Get greatest





state lattice need to traverse only relevant states
2nd order: state in the lattice now include both current and the previous state

Other tasks:
Compute likelihood (probability of a sentence regardless of its tags, $P(x|\theta)$ Instead of original $P(y,x|\theta)$
= a Language Model

Since in probability:

$$P(x|\theta) = \sum_y P(x,y|\theta)$$

Forward algorithm To get LM by HMM

Initialization: $v_j^1 = a_{START,j} b_{j,x^1}, \quad j = 1,\ldots,N;$

Recomputation: $v_j^t = \left(\sum_i v_i^{t-1} a_{ij} \right) b_{j,x^t}, \quad j = 1,\ldots,N, \quad t = 2,\ldots,|x|$

Final: $v_{STOP}^{|\mathbf{x}|+1} = \sum_i v_i^{|\mathbf{x}|} a_{i,STOP}$

Unsupervised learning of parameters theta

Task:
learn the best set of parameters $\hat{\theta}$ given only an unannotated corpus of sentences.

EM:

- Initialize parameters, $A^{(0)}$ and $B^{(0)}$
- At each iteration k:
  - E-step: Compute expected counts using $A^{(k-1)}$ and $B^{(k-1)}$
  - M-step: set $A^{(k-1)}$ and $B^{(k-1)}$ using MLE on the expected counts

Repeat until doesn't converge (a stopping criteria).

Expected counts:

- With current A and B, compute probs of all possible tag sequences.
- If sequence **y** has probability p, count p for each $(y^{t-1}= i, y^t=j)$ in **y**.
- Add up these fractional counts across all possible sequences

Example

- Notionally, we compute expected counts as follows:

Possible tag sequence	Probability of the sequence
N N N	p_1
N V N	p_2
N N V	p_3

aa bb cc - Sequence of observations (words)

$C_T(N, N) = 2\,p_1 + p_3$

Forward backward algorithm

Forward-Backward (Baum-Welch) algorithm computes expected counts (using forward probabilities and backward probabilities)

$$P(y^{t-1} = i, y^t = j|x)$$