Computational Linguistics

Summary by Aurea

**Intro**

Course outline:

- 14 meetings
    - 7 in block 3; 7 in block 4
    - 10 classes, 1 invited talk, 1 research seminar, 2 Q&A sessions
- Check the course plan on Canvas for more details
- Check the pre-requisites and make sure to install everything you need

Classes:

- Knowledge clips to watch before lecture
- Textbook chapters to read before lecture
- Python notebook to download and check before lecture
- Hands-on lectures based on Python notebooks and tasks to carry out alone or in small groups
- Content course, not methods!

Take home assignment:

- Individual assignment consisting of different sub-tasks
- Worth 40 points (20% of the final grade)
- You'll have to comment the code I provide, write some code (and comment it), solve tasks and interpret the results you produce
- First attempt in block 3
- Resit in block 4 (but you'll get 22 points tops)
- Check Canvas for details about timing, deadlines, submission guidelines and evaluation criteria

Mid-term (60 points, 30% of final grade):

- 40 multiple choice questions (4 answer options)
- Open book, not proctored
- 1h

Written exam (100 points, 50% of  the final grade):

- 15 closed questions
- 8 open ended questions
- Open book, not proctored
- 2h30m

What this course is:

- An into to CL, combining statistical and neutral methods
- A course which combines linguistic and statistical theory with application-oriented content
- A mix of theory and code to implement it
- A mix of foundational content and current topics (covered at the end in the guest lecture and research seminar)

Course material: Jurafsky & Martin, Speech and language processing. 3ed (link in the course description on Canvas)

# KNOWLEDGE CLIPS 1 – Introduction

## Introduction

**What is Computational Linguistics?**

The field concerned with using automatic computational methods to analyse and synthesize natural languages (text, speech and gesture).

**Natural languages**

The study of natural languages is the domain of linguistics, which has developed several tools to formalize and describe languages.
These methods can be used to automate language processing, better describe how a system works and evaluate it sensibly.

**Automatic computational methods**

Several efficient and scalable algorithms and formalisms are used to automate language analyses: these algorithms have typically have been developed in computer science.
These algorithms are used to perform tasks accurately and quickly, in an optimal way.

**Analyse and synthetize**

Natural languages are often usefully characterized in terms of probability distributions over discrete units (words, sequential information, meaning, …).
Statistics provides tools to manipulate probability distributions correctly and use this information to resolve ambiguity.

**What is (and isn't) a natural language?**

It is:

- Conventional
- A set of related systems
- Redundant
- Subject to change
- Context dependent

It isn't:

- Formal logic
- A programming language
- Machine language

**What are the goals?**

- *Infer* the component symbols of a language, their roles, the rules for combining them, and their meaning
- *Formalize* the rules for combining symbols
- *Combine* atomic meanings
- *Understand* large portions of text
- *Produce* complex sentences

**Why should we care?**

Language is the most natural way in which people interact. CL provides tools to study these interactions and supports the implementation of automatic tools to interact using language.

## Levels of analysis

**Phonology**

Studies linguistic sound to construct inventories of sound with a linguistic role.
Different from phonetics!

**Segmentation**

The task of splitting text or speech (harder, why?) into symbol (letters, phonemes, morphemes, words, chunks, …) of the appropriate granularity.

**Morphology**

Studies how words are built up from smaller meaning-bearing units, the morphemes.
Morphological complexity varies cross-linguistically: some languages have simple morphological systems, others crazy complex ones.

**Syntax**

Studies the set of rules, principles and processes for combining symbols according to the structure of the language. Asserts whether a sentence is well-formed in a language.

**Lexical semantics**

Describes the meaning of single symbolic units (words, morphemes, collocations).
It aims to classify and decompose lexical items, compare lexical semantic structures cross-linguistically, and understand similarities across items.

**Compositional semantics**

Studies how atomic meanings are combined into larger meaningful units, such as sentences, paragraphs, and so on…

**Pragmatics**

Analyses how context influences meaning, encompassing semantics, linguistic knowledge of participants, situational context, shared knowledge, goals and intent.

**I made her duck: Ambiguity everywhere**

- *Morpho-syntactic*: her (dative vs. possessive) and duck (noun vs. verb)
- *Semantic*: make (cook, create, cause, transform) and duck (bird, action of avoiding)
- *Syntactic*: make (transitive vs. ditransitive)
- *Phonological*: I-eye, made-maid

Luckily there’s redundancy.

**What do we do?**

1. Identify the language
2. Identify the linguistic units (letters, morphemes, words, phrases, sentences, paragraphs, …)
3. Find structure beneath the surface
4. Understand what each symbol means
5. Understand what the whole message aims to convey
6. Respond appropriately

## Data

**Corpus (corpora *in plural*)**

A computer-readable collection of linguistic productions (text, speech, gestures).

Every text/speech act is produced

- By somebody
- In a specific natural language
- At a given time
- In a specific place
- And for a certain goal

The sentences in a language are infinite.

A corpus is a tiny fraction of the language it aims to represent. Good models require the corpus to be representative of the phenomenon to be modelled.

Different types of corpora:

- Crawled/manually curated
- Balanced/imbalanced
- Single author/more authors
- Diachronic/synchronic
- Written/spoken/mixed
- Single language/multi-language/parallel
- …

Examples: "The British National Corpus", "Wikipedia", "SubtLex", "CHILDES"…

**Lexicon**

List of words enriched with some attributes which specify some property (or properties) of each word in a given domain, e.g. concreteness, imaginability, perceptual salience, sentiment, …

**Thesaurus**

A resource that groups words by how similar their meaning is, from synonymy to antonymy, sometimes enriched with other relations and definitions.

**WordNet**

The most famous and widely used thesaurus in CL. It represents word senses and relations among them. It also includes glosses.

It consists of three databases (nouns; verbs; adjectives and adverbs)

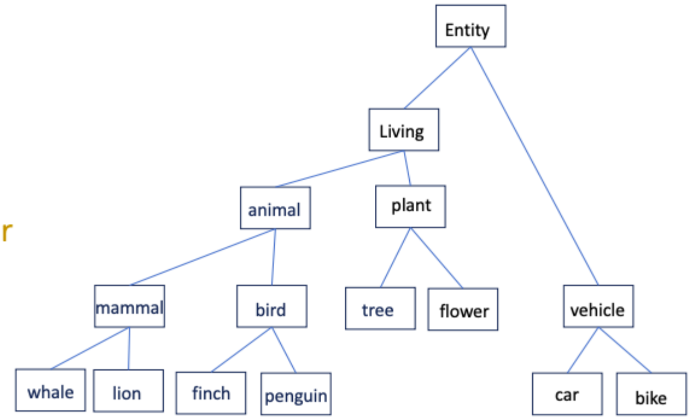
**Synsets**

WordNet has a tree structure and is organized in synsets (near-synonym sets) connected by hypernymy-hyponymy (is-a) relations.

**Path length similarity**

Path length (maximum similarity): two words are more similar the shorter the path that connects them.

r



**Resnik similarity**
Information-content weighed path length: two words are more similar the higher the information content of their lowest common subsume.

**Lesk similarity**
Overlap of glosses: two words are more similar the higher the LexiCap overlap of their glosses.
*Cat: feline mammal usually having thick soft fur and no ability to roar*
*Lion: large gregarious predatory feline of Africa and India having a tawny coat with shaggy mane in the male*
*Car: a motor vehicle with four wheels, usually propelled by an internal combustion engine*

## KNOWLEDGE CLIPS 2 – Naïve Classifiers

**Naïve Bayes Classifiers**
Classify texts by learning how to produce them

### Classification

**What is it**
Given an unseen data point, assign it to one of the C classes based on some representation of the data point and some evidence of how features relate to classes

**Ubiquitous in NLP** (natural language processing)

- Language identification
- Author profiling
- Spam filters
- E-mail routing
- Topic detection
- Sentiment analysis
- Fake news detection
- …

**A crude solution**
Identify discriminative patterns in what you want to classify and write a long series of if-elif-else rules to cover all relevant patterns.
Offers fine-grained control on classification outcomes and we always know why something happened!

**Rule systems**

- Robust: you can write a rule for a rare event and it will be applied anytime that event occurs
- Good way to incorporate intuitions and domain knowledge into a system
- No necessity of large datasets
- Expensive to write
- Require domain knowledge
- Rigid when dealing with ambiguity

**Supervised learning**
The counterpart to rule-based systems. Provide examples to an algorithm, tell it when it's doing good and then have it perform the same task on new data.
No need to hand-write rules, but very much subjected to data problems.

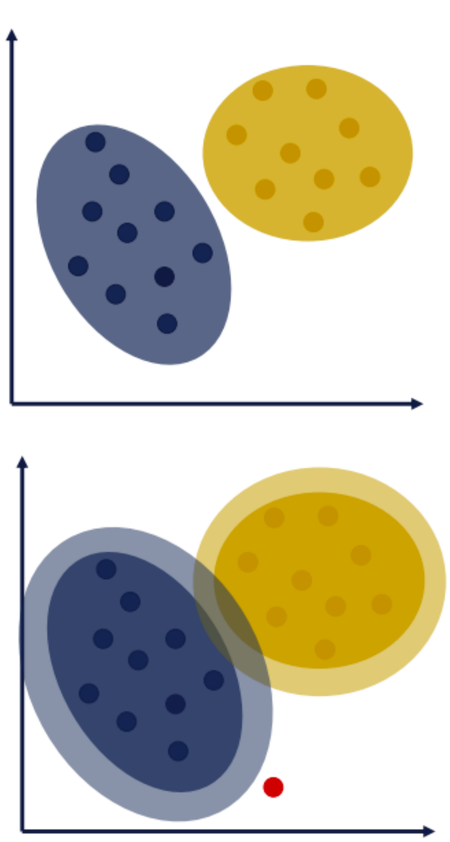
**What is learning?**
Learning =/=memorizing
Learning = generalize, infer, analogize, adapt
Learning is an experience-driven, long term, relational process which results in a change in how we interact with the environment due to the information we have come into contact with.

**What is machine learning?**
The study of techniques to enable machines to learn from data and experience in an automate way.

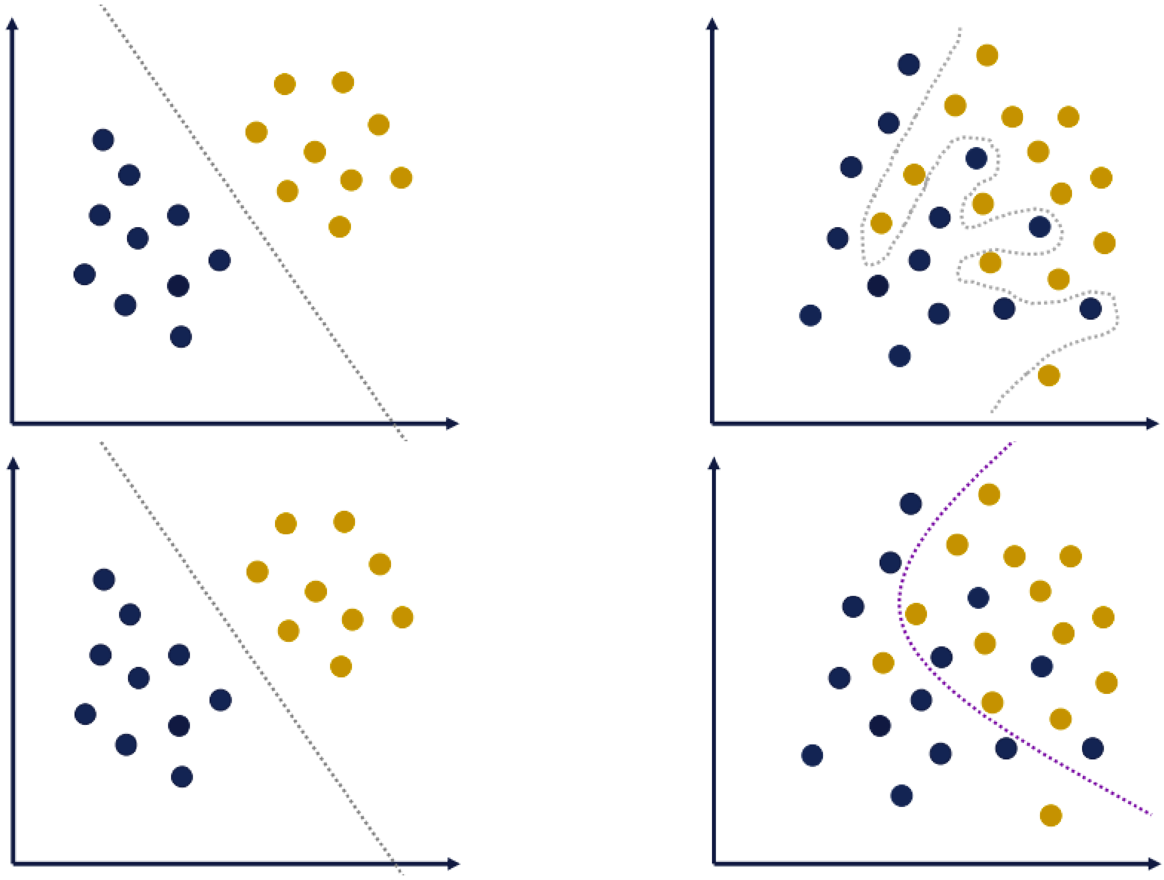
**Why do we care?**
Languages change and differ: having a system which can be trained is more convenient than having to come up with new rules.
Many problems can be cast in terms of learning problems and machine learning offers principled and efficient techniques to address them.

**A core distinction**
*Generative classifiers* learn a model of how the data are generated and could in principle generate new data. Classification happens by choosing the class most likely to have generated the data.
*Discriminative classifiers* learn which features best predict a certain class.

## Generative vs. discriminative



## Linear vs. non-linear

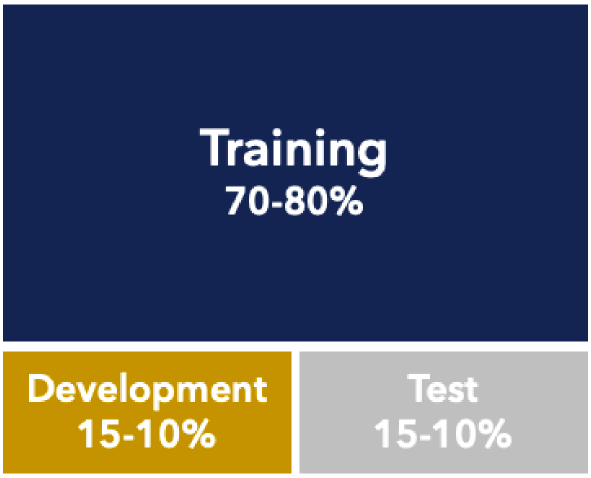


### Overfitting

Learning a function which is too specific *wrt* the training data and doesn't generalise to unseen data.

### Train-dev-test

We try to prevent overfitting by fine-tuning our models (feature representation, algorithm, decision function, optimization) on a different portion of data than the one we test on.



### K-fold cross-validation

Split the data in k portions of the same size, then iteratively train k-1 sub-sets and test on the remaining sub-set. Average scores over the k runs.
K usually set to 5 or 10.

## Evaluation

### Who's better?

How do you determine whether a system performs better than another?

### Intrinsic vs. extrinsic evaluation

- *Intrinsic*: define a metric and check which systems does best.
- *Extrinsic*: embed a tool in a larger system and check how much performance on a downstream task improves given the output of two different versions of your tool.

### The ideal world

Identify all and only the correct data points (examples from the world beyond text?).
Often not possible: then choose whether to miss some you should spot or spot some you should miss.

### Accuracy

The proportion of data points that were assigned to the correct class out of all the data points.

$$Acc=\frac{TP+TN}{¿data}$$

Not very good with imbalanced datasets.

**Precision**

The proportion of correctly classified data points out of the data points you guessed belonged to that class.

$$P_K = \frac{TP_K}{TP_K + FP_K}$$

It doesn't matter if you missed some, as long as you didn't make mistakes when you guessed K (but at least guessed it once, why?).

**Recall**

The proportion of correctly classified data points out of the data points which belonged to that class.

$$R_K = \frac{TP_K}{TP_K + FN_K}$$

It doesn't matter how often you wrongly guessed K as long as you got all data points which belonged to class K.

**F-measure**

The harmonic mean between precision and recall (more conservative than the arithmetic mean)

$$F_K = \frac{(\beta^2 + 1) * P_K * R_K}{\beta^2 * P_K + R_K}$$

The beta is a free parameter which weighs the two terms (favouring recall when > 1). Typically set to 1 (hence $F_1$).

**Macro-averaging**

When there are more than two classes, we compute the F-measure for all the classes and average them assigning equal importance.

Useful when good performance is necessary in all classes, regardless of their frequency.

**Micro-averaging**

We collect the decisions for all classes in a single contingency table and then compute precision and recall from that table.

Useful when good performance is more important for the more frequent classes.

**Who's *reliably* better?**

Can't use t-test because most performance metrics are not normally distributed – which violates the assumption of the test.

Use non-parametric tests, bootstrapping, or approximate randomization testing.

**Bootstrapping**

Artificially increase the number of test sets: draw many (many!!) samples from a given test set with replacement, perform your task, record the score, factor in the performance on the whole test set, then simply check the percentage of runs in which a system beats the other.

## Bayes rule

$$p(c_i|d) = \frac{p(d|c_i) * p(c_i)}{p(d)}$$

### Prior: $p(c_i)$

How often does each class $c_i$ occur in the input?
Encodes the expectations of finding each class in the world.

### Likelihood: $p(d|c_i)$

How likely is it that each feature is observed if the instance is generated by class $c_i$?

$$p(f_1|c) * p(f_2|c) * \ldots * p(f_n \vee c)$$

Encodes the evidence that an instance has been generated by a given class $c_i$.

### Posterior: $p(c_i|d)$

How likely is it that I have an example of class $c_i$ given the instance I'm seeing (i.e. given the feature representation I'm seeing)?

$$C_{NB} = argmax_{c \in C}\, p(c) \prod_{f \in F} p(f|c) = argmax_{c \in C} \log(p(c)) + \sum_{f \in F} \log\, (p(f|c))$$

### Maximum-A-Posteriori

Refers to classifiers which pick the class which maximizes the posterior probability given the current data point.
Often shortened as MAP.

## Naïve Bayes Classifiers

The simplest linear generative probabilistic classifier.
Given a document *d*, it returns the class c with the highest posterior probability given the document:

$$c = argmax_{c \in C}\, p(c \vee d)$$

### Naïve because

- It treats text documents as bag-of-words, discarding all positional and sequential information
- It assumes conditional independence across features (doesn't keep track of feature co-presence)

### Maximum Likelihood Estimates

Prior probabilities for each class are approximated as $p(c) = N_c/N_{documents}$
Likelihoods are approximated by $p(f_i|c)/\sum_{f \in F} count(f,c)$ where F is the set of features appearing in all classes.

### Problems

What if a feature never occurs with a class during training?
If the likelihood of a feature given a class is 0, then the likelihood of the document given the class is 0 and posterior is 0. All because of one missing feature.

**Smoothing**

Very simple: add 1 to all counts so to avoid 0s

$$p(f_i|c)=\frac{count(f_i|c)+1}{\sum_{f\in F}(count(f,c)+1)}=\frac{count(f_i|c)+1}{¿¿}$$

It is paramount that F is the entire set of features, not just those that co-occur with class c. Why?

**Out-Of-Vocabulary words**

What do, e.g., with words that appear at test but not in the vocabulary collected training? Ignore them or make a dedicated feature in training as well (perhaps one class has more than another).

They’re probably rare enough that we don’t care, If there are too many OOV words, check that your sample is representative!

**Stop words**

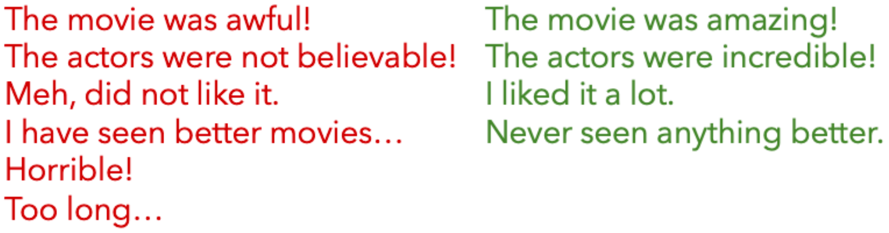
Very frequent, usually uninformative words (the, I, you, of, …): there are lists which can be used to filter them out.

What is the relation between frequency and informativeness? Where did we see it already?

Can you think of applications where you really want to keep stop words around?

Worked-out example

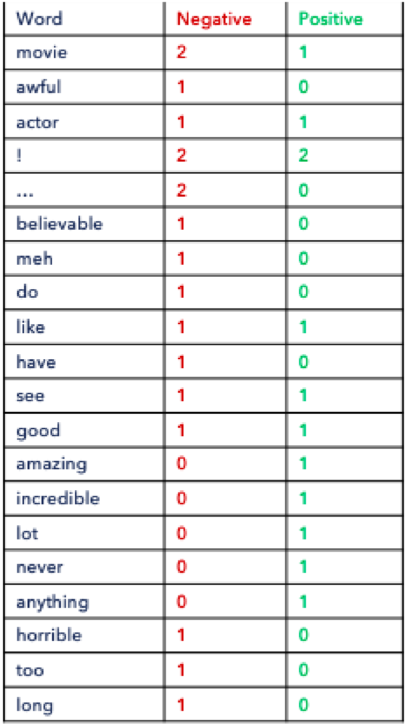
**Some texts**



The movie was awful!
The actors were not believable!
Meh, did not like it.
I have seen better movies…
Horrible!
Too long…

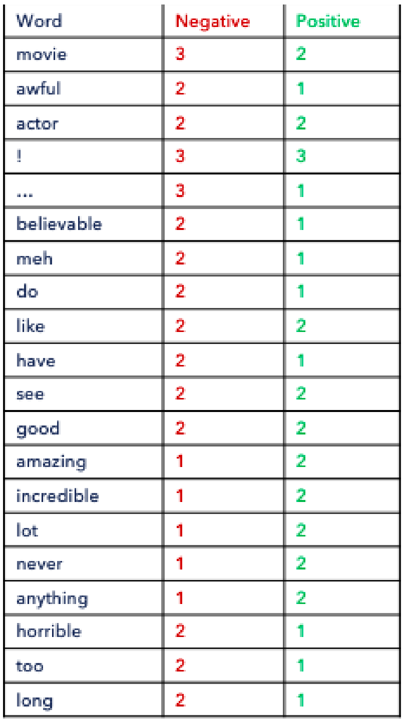
The movie was amazing!
The actors were incredible!
I liked it a lot.
Never seen anything better.

## Corresponding counts:



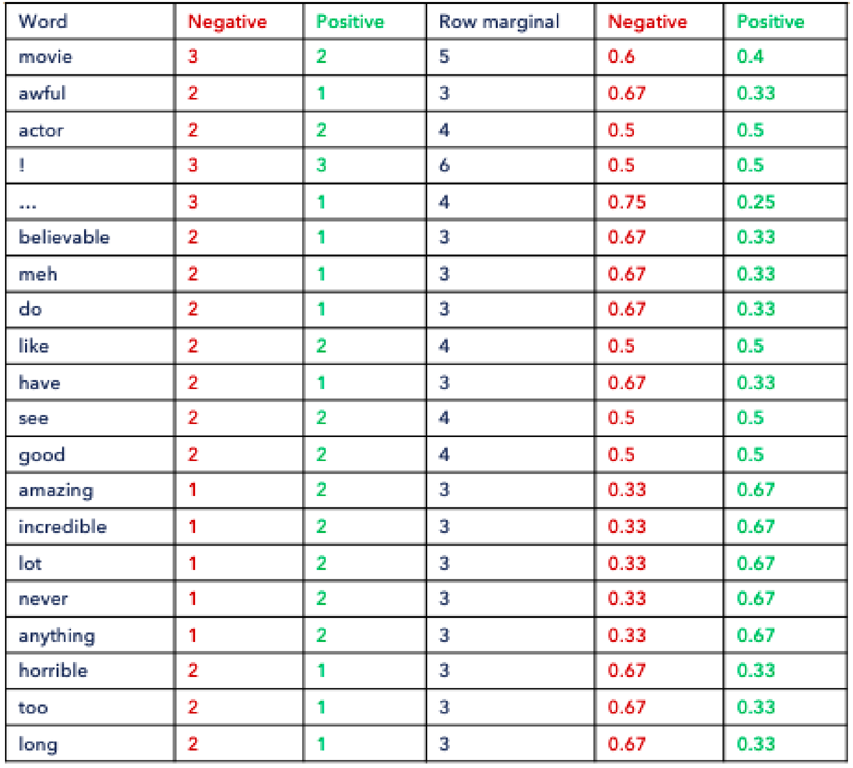
| Word | Negative | Positive |
|---|---|---|
| movie | 2 | 1 |
| awful | 1 | 0 |
| actor | 1 | 1 |
| ! | 2 | 2 |
| ... | 2 | 0 |
| believable | 1 | 0 |
| meh | 1 | 0 |
| do | 1 | 0 |
| like | 1 | 1 |
| have | 1 | 0 |
| see | 1 | 1 |
| good | 1 | 1 |
| amazing | 0 | 1 |
| incredible | 0 | 1 |
| lot | 0 | 1 |
| never | 0 | 1 |
| anything | 0 | 1 |
| horrible | 1 | 0 |
| too | 1 | 0 |
| long | 1 | 0 |

## Smoothed counts: add 1



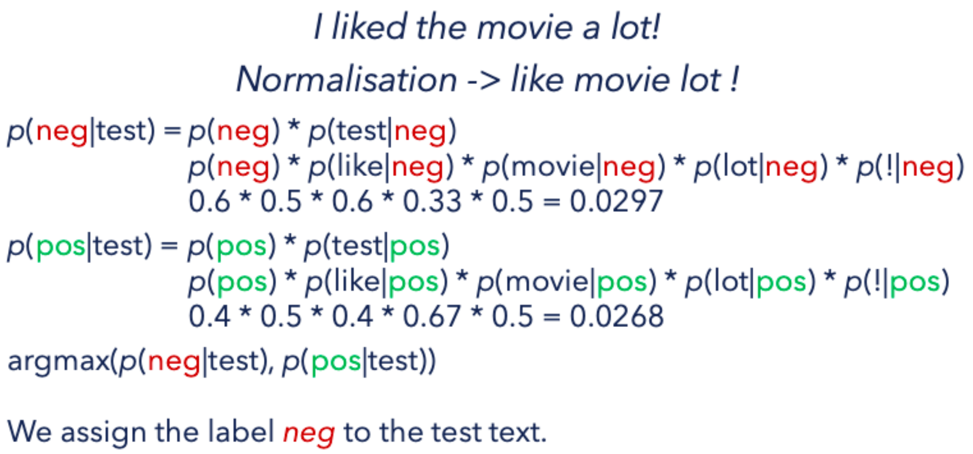
| Word | Negative | Positive |
|---|---|---|
| movie | 3 | 2 |
| awful | 2 | 1 |
| actor | 2 | 2 |
| ! | 3 | 3 |
| ... | 3 | 1 |
| believable | 2 | 1 |
| meh | 2 | 1 |
| do | 2 | 1 |
| like | 2 | 2 |
| have | 2 | 1 |
| see | 2 | 2 |
| good | 2 | 2 |
| amazing | 1 | 2 |
| incredible | 1 | 2 |
| lot | 1 | 2 |
| never | 1 | 2 |
| anything | 1 | 2 |
| horrible | 2 | 1 |
| too | 2 | 1 |
| long | 2 | 1 |

## Conditional probabilities



| Word | Negative | Positive | Row marginal | Negative | Positive |
|---|---|---|---|---|---|
| movie | 3 | 2 | 5 | 0.6 | 0.4 |
| awful | 2 | 1 | 3 | 0.67 | 0.33 |
| actor | 2 | 2 | 4 | 0.5 | 0.5 |
| ! | 3 | 3 | 6 | 0.5 | 0.5 |
| ... | 3 | 1 | 4 | 0.75 | 0.25 |
| believable | 2 | 1 | 3 | 0.67 | 0.33 |
| meh | 2 | 1 | 3 | 0.67 | 0.33 |
| do | 2 | 1 | 3 | 0.67 | 0.33 |
| like | 2 | 2 | 4 | 0.5 | 0.5 |
| have | 2 | 1 | 3 | 0.67 | 0.33 |
| see | 2 | 2 | 4 | 0.5 | 0.5 |
| good | 2 | 2 | 4 | 0.5 | 0.5 |
| amazing | 1 | 2 | 3 | 0.33 | 0.67 |
| incredible | 1 | 2 | 3 | 0.33 | 0.67 |
| lot | 1 | 2 | 3 | 0.33 | 0.67 |
| never | 1 | 2 | 3 | 0.33 | 0.67 |
| anything | 1 | 2 | 3 | 0.33 | 0.67 |
| horrible | 2 | 1 | 3 | 0.67 | 0.33 |
| too | 2 | 1 | 3 | 0.67 | 0.33 |
| long | 2 | 1 | 3 | 0.67 | 0.33 |

**Test**



*I liked the movie a lot!*

*Normalisation -> like movie lot !*

$p$(neg|test) = $p$(neg) * $p$(test|neg)
$p$(neg) * $p$(like|neg) * $p$(movie|neg) * $p$(lot|neg) * $p$(!|neg)
0.6 * 0.5 * 0.6 * 0.33 * 0.5 = 0.0297

$p$(pos|test) = $p$(pos) * $p$(test|pos)
$p$(pos) * $p$(like|pos) * $p$(movie|pos) * $p$(lot|pos) * $p$(!|pos)
0.4 * 0.5 * 0.4 * 0.67 * 0.5 = 0.0268

argmax($p$(neg|test), $p$(pos|test))

We assign the label *neg* to the test text.

KNOWLEDGE CLIPS 3 – Pre-processing

**Text normalization**

How to find the signal amidst the noise.

The basics

**Tokens, types, lemmas**

A *token* is every word in a corpus.
A *type* is every distinct word in a corpus.
A *lemma* is the dictionary entry of a word.

**A practical example**

I will eat what you eat, even if I have never eaten it before.

- 14 tokens (16 with punctuation)
- 12 types (eat and eat are one type, as the two Is)
- 11 lemmas (eat and eaten are the same lemma)

**Lemmatization**

The process of reducing each word-form found in a corpus to its corresponding lemma.

- Is → be
- Went → go
- Geese → goose
- Best → good
- Talked → talk
- …

**Stems and affixes**

Morphemes are the smallest meaning-bearing unit in languages*.
Stems are the main meaning bearing unit in words.
Affixes are strings that modify stems in some wat.
talked = talk(the act of speaking)+ -ed(past)

**Stemming**

The process of reducing word-forms to their stem stripping away all affixes. Related, but not the same as lemmatization (consider irregular cases).
Lemmatization → lemma
-ation is an affix, -(t)iz(e) is an affix and when we stem we strip affixes even though lemmatization is a lemma itself!

**Derivational vs. inflectional**

Inflectional affixes don't create new words and encode numbers, tense, aspect, gender, case. Are fully productive.
talk + -ed[PAST] → talked; sheep + Ø[PL] → sheep
Derivational affixes create new words by changing grammatical category or meaning, and encode things like negation, normalization, reiteration, …
happy + un-[NOT] → unhappy
happy + -ness[NOUN] → happiness
(happy + -ness[NOUN] + un-[NOT] → unhappiness

**Compounds**

Stems can be combined together to produce new meaning (e.g. fire+fighter, tele+vision, motor+bike,…): such words are called compounds.
Their meaning can be very transparent (easy to guess given the components) to very opaque (the meaning of the compound doesn't follow that of the constituents).

**Morphological complexity**

Words can be:

- Monomorphemic when they only consist of a single morpheme (plus possible inflectional morphemes – sings is still monomorphemic, technically)
- Polymorphemic when they include derivational affixes or are compounds

## Normalization

**What does it entail?**

- Segmenting sentences from running text
- Segmenting individual words from running text (tokenization)
- Normalizing word formats (spelling correction, lemmatization, case folding)

**Tokenization**

hyphens: gold-digger or gold – digger?
clitics (I'm or I ' m or I am)?
dates?
abbreviations (m.p.h. or m . p . h .)?
URLs, hashtags, email addresses?
named entities (New_York or New York)?
Often task specific?

**One standard (LDC)**

Hyphenated words stay together

Named Entities stay apart (New York)
Doesn't → does n't
Punctuation is separated
$10 → $ 10

**A few cases**
God and god most likely refer to the same thing and should be treated as the same word (but US and us!)
so much and soooooooooo much are the same word
snowball and snowbakl are probably the same word, the second is just mistyped

**Normalization is always a loss…**
… but there are things you'd rather not know. There can be too much information!
Normalizing entails discarding variability, which is informative, but may be dispersive: find your balance depending on what you want to achieve!

**Variation can be good, though**
When could we want to treat God and god as two different words?
When are so much and soooooooooo much different words? Would soooooooooo much and sooooo much be?
When do we want snowball and snowbakl to be different?

## Regular Expressions

**A flexible tool to search texts**
What if we want to match both god and God? Or so much, sooooo much, and soooooooooo much? Or all inflected forms of a certain lemma? Or find e-mail addresses, URLs, dates in text to replace all of them with the same character?

**Disjunctions**
In standard RegExp* syntax – which comes from Perl and is used in Python as well – we can say we want strings containing either one character or the other, to match god and God, like this.
/[gG]od/
* RegExp (regular expression) – a sequence of characters that specifies a search pattern in text. Usually such patterns are used by string-searching algorithms for "find" or "find and replace" operations on strings, or for input validation.

**Ranges**
When the disjunction can involve more symbols which need to be matched if occurring in the same position, we can use ranges:

- [a-z]: any lower case letter
- [A-Z]: any upper case letter
- [0-9]: any digit

We can use partial ranges too, e.g. [0-5] or [a-t]

**Wildcard**
We can be more liberal and match really anything at all: to achieve this we use a dot (.)

Beware, anything means literally anything.

**Negation**
We can tell a RegExp to match anything but something, like so [^…].
We can negate range, so [^0-9] matches any character except for a digit.

**Counters**
We can say that we want strings containing the same character multiple times in the following ways:

- /so? much/: o once or not at all, so s much is good
- /so+ much/: o at least once, so s much isn't good
- /so* much/: o an arbitrary number of times, including not at all, so s much is good
- /so{n,m} much/: o at least n times and at most m times

**Anchors**
A RegExp matches anything happening anywhere in a string. But we can restrict the match to the beginning (^) or the end ($) of the string.
/^god$/ will match the string "god", but not "gods" or "pagoda".

**Escaping**
In RegExps, some symbols have special meanings: ^means not or start of string, $ means end of string, . means anything, +, ?, *,{,} are counters. If we want to match the dollar sign, the caret, the plus symbol, we escape them by using a slash: \. means a dot.
Other symbols may be forced to have special meanings: b matches the letter, but \b means a word boundary (not a letter, digit or underscore); its negation is \B.

**Grouping**
Sometimes we want to apply a counter not to a single character but to a sequence, or we want to consider disjunctions between sequences. To tell a RegExp that some characters go together as a unit we use brackets (…).
(ah)+ matches ahahahahah
B(eh|he) matches both beh and bhe

**Replacing**

*s/ +/\t/g*

replaces any white space or sequence thereof with a single tab.
*s* indicates to replace the first block with what appears in the second block
*g* indicates to do this for every occurrence of the target pattern in the string, not just the first

**Capture groups**

*s/ ?([^0-9a-zA-Z]+) ?/_\1_/g*

encloses between underscores all non-alpha numerical symbols (and sequences thereof) in between spaces or not (mr. becomes mt_._).
The use of parentheses to store a pattern in memory is called a capture group: it is stored in a numbered register (so \1 points to the first captured pattern, \2 to the second, if present, and so on…)

**Normalize all e-mails**

*s/((^|\b)[A-z0-9\._%\+\-]+FSAs@[A-z0-9\.\-]+\.[A-z]{2,}($|\b))/EMAIL/g*

matches most e-mail addresses and replaces them with the token EMAIL so we know that was originally an email without caring for its specificity.

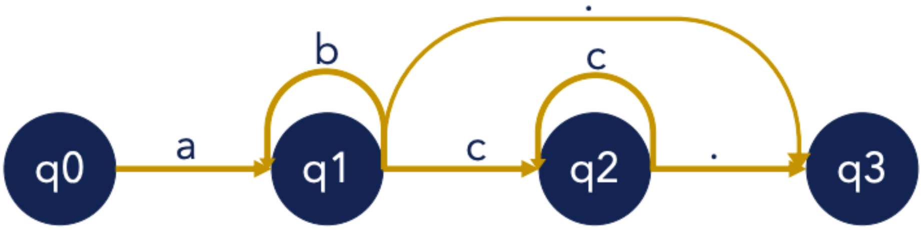
## Finite State Automata

### A formalism to interpret REs

Any regular expression (RE) can be implemented using an FSA. Any FSA can be written as a RE.

Represented using directed graphs: a finite set of nodes, or states, (O) connected by directed edges (→).

### Directed graphs



### Start state and accepting state

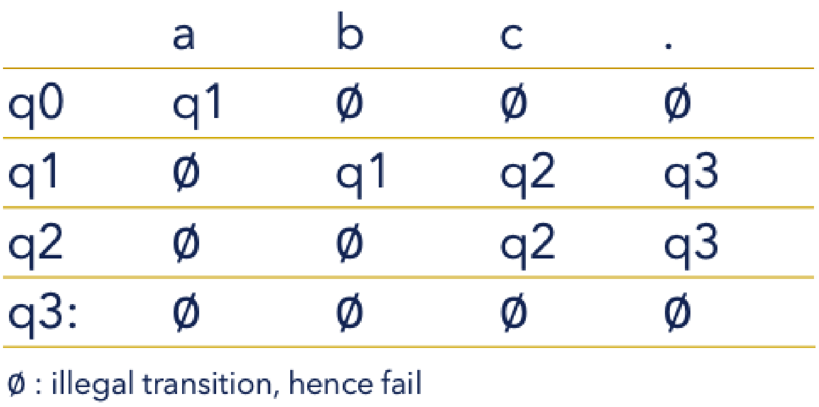
All FSAs have a starting state and an accepting state: if the atomization reaches the accepting state, the string is recognized.

If the automaton gets stuck, i.e. the next symbol in the input doesn't match any states which can be reached from the current state, or has no more input left to process (without being in the accepting state), the string is rejected.

### FSA components

1. Q : a finite set of states
2. $\Sigma$ : a finite alphabet of symbols
3. $q0$ : the initial state
4. F : the set of final states (contained in Q)
5. $\delta(q,i)$ : the transition function. Given a state q from Q and an input I from $\Sigma$, $\delta(q,i)$ returns a new state q'.

### Transition function



| | a | b | c | . |
|---|---|---|---|---|
| q0 | q1 | ∅ | ∅ | ∅ |
| q1 | ∅ | q1 | q2 | q3 |
| q2 | ∅ | ∅ | q2 | q3 |
| q3: | ∅ | ∅ | ∅ | ∅ |

∅ : illegal transition, hence fail

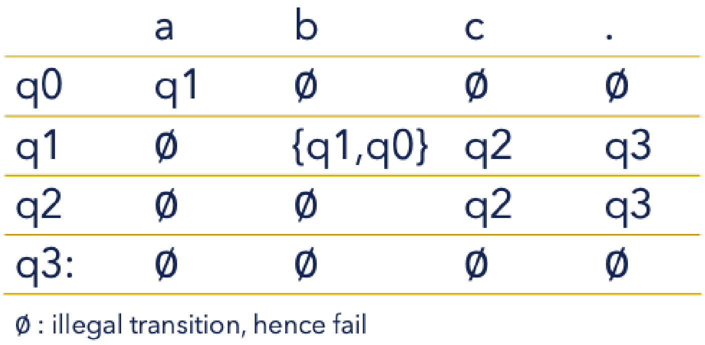
Let's try a few strings:

- a.
- ac.
- aa
- abbbbb
- abc.
- abbbbbc.
- aba.
- abccc
- abccccccccccc.

**Non-determinism**

Rather than having the next state fully determined by current state and the next symbol, we can take multiple paths.
How do we know whether we took the right one?

**Non-deterministic transition function**



| | a | b | c | . |
|---|---|---|---|---|
| q0 | q1 | ∅ | ∅ | ∅ |
| q1 | ∅ | {q1,q0} | q2 | q3 |
| q2 | ∅ | ∅ | q2 | q3 |
| q3: | ∅ | ∅ | ∅ | ∅ |

∅ : illegal transition, hence fail

Let's take abbc.

$\delta(q0,a)$ → q1
$\delta(q1,b)$ → q1|q0

What do we do?

**Option 1: backup**

Backup: mark choice-points to go back if you take a dead end.

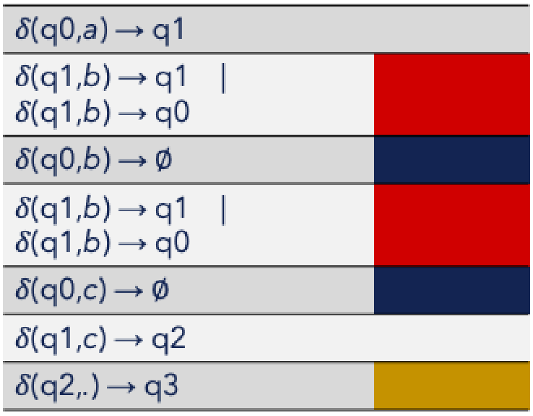
**Option 2: Look-ahead**

Look-ahead: take a look at what's next to help you chose in which of the possible states to move

**Option 3: parallelize**

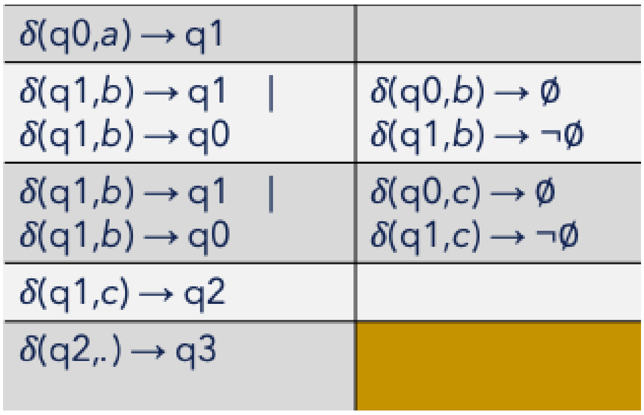
Parallelize: explore every alternative in parallel

OPTION 1



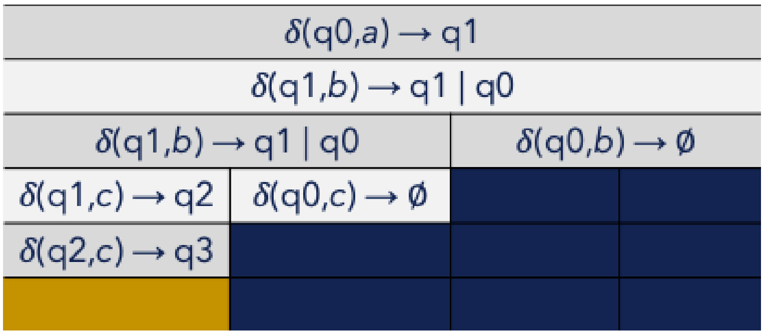
| | |
|---|---|
| $\delta(q0,a) \rightarrow q1$ | |
| $\delta(q1,b) \rightarrow q1$ \|<br>$\delta(q1,b) \rightarrow q0$ | 🟥 |
| $\delta(q0,b) \rightarrow \emptyset$ | 🟦 |
| $\delta(q1,b) \rightarrow q1$ \|<br>$\delta(q1,b) \rightarrow q0$ | 🟥 |
| $\delta(q0,c) \rightarrow \emptyset$ | 🟦 |
| $\delta(q1,c) \rightarrow q2$ | |
| $\delta(q2,.) \rightarrow q3$ | 🟨 |

OPTION 2



| | |
|---|---|
| $\delta(q0,a) \rightarrow q1$ | |
| $\delta(q1,b) \rightarrow q1$ \|<br>$\delta(q1,b) \rightarrow q0$ | $\delta(q0,b) \rightarrow \emptyset$<br>$\delta(q1,b) \rightarrow \neg\emptyset$ |
| $\delta(q1,b) \rightarrow q1$ \|<br>$\delta(q1,b) \rightarrow q0$ | $\delta(q0,c) \rightarrow \emptyset$<br>$\delta(q1,c) \rightarrow \neg\emptyset$ |
| $\delta(q1,c) \rightarrow q2$ | |
| $\delta(q2,.) \rightarrow q3$ | 🟨 |

OPTION 3



| | | | |
|---|---|---|---|
| $\delta(q0,a) \rightarrow q1$ | | | |
| $\delta(q1,b) \rightarrow q1 \mid q0$ | | | |
| $\delta(q1,b) \rightarrow q1 \mid q0$ | | $\delta(q0,b) \rightarrow \emptyset$ | |
| $\delta(q1,c) \rightarrow q2$ | $\delta(q0,c) \rightarrow \emptyset$ | | |
| $\delta(q2,c) \rightarrow q3$ | | | |
| | | | |

**Search-state**

For every choice-point, we need to record the state we can go to and the position in the string we are parsing.

Explore all possible paths before rejecting! This problem is solved using state space search algorithms.

Edit distance

**Spelling correction**

How do we know that snowbakl should be the same as snowball and not, say, snowplow. Intuitively, the first two strings look more alike, but how can we quantify it?

**Insert, delete, substitute**

Measure the number of transformations to go from one string to the other: it takes one step to get from snowball to snowbakl but four to get to snowplow.

You can ban substitutions, which amounts to make them count twice since a substitution equals a delete and an insertion.

**Alignment**

I N T E * N T I O N<br>
* E X E C U T I O N<br>
d s s - i s - - - -

**Alternative**

I N T E * * * N T I O N<br>
* * * E X E C U T I O N<br>
d d d - i i i s - - - -

If substitutions cost double, the edit distance is the same (8) but this alignment requires more operations so it is worse. How do we determine the best? What is best?

**How to compute a minimum edit distance**

A search task: find the shortest sequence of edits to go from a string to another.

The search space is large especially for long strings – like whole sentences – however, many paths end up in the same spot.

**Dynamic programming**

Solve a problem by combining the solutions to smaller problems.

Many different paths in the search space end up in the same place but only one is shortest: remember it because it will necessarily be part of the global shortest path!

### The algorithm

First compute the distance of each string to the empty string. Then:

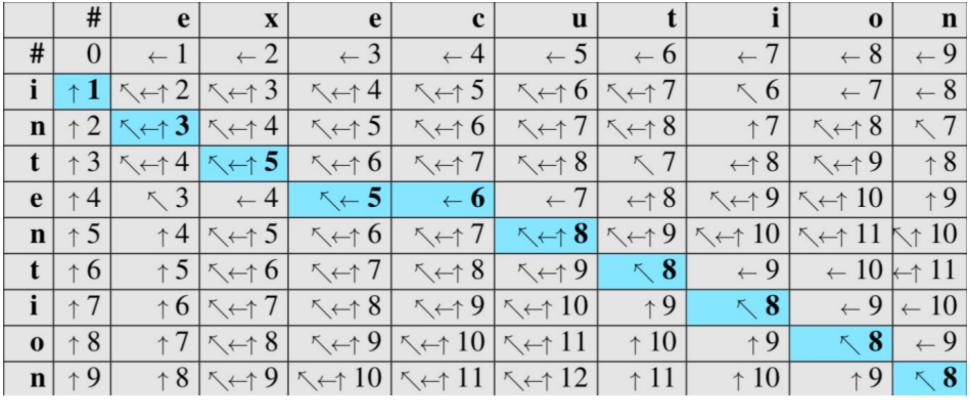
```
For every index i ≤ n in the source:
     for every index j ≤ m in the target:
          if source[i] != target[j]:
               D(i, j) = 1 + argmin(D(i-1, j)
                                    D(i-1, j-1)
                                    D(i, j-1))
          else:
               D(i, j) = D(i-1, j-1)
return D(n, m)
```

### Provide the best alignment too

The algorithm we saw only provides the distance, but doesn't provide the best alignment. To do this, we also store back-pointers in each cell, so we know where we came from when we reached a transformation.

Then, we trace back our steps, favouring substitutions every time we can.

### Graphically



| | # | e | x | e | c | u | t | i | o | n |
|---|---|---|---|---|---|---|---|---|---|---|
| # | 0 | ← 1 | ← 2 | ← 3 | ← 4 | ← 5 | ← 6 | ← 7 | ← 8 | ← 9 |
| i | ↑ **1** | ↖←↑ 2 | ↖←↑ 3 | ↖←↑ 4 | ↖←↑ 5 | ↖←↑ 6 | ↖←↑ 7 | ↖ 6 | ← 7 | ← 8 |
| n | ↑ 2 | ↖←↑ **3** | ↖←↑ 4 | ↖←↑ 5 | ↖←↑ 6 | ↖←↑ 7 | ↖←↑ 8 | ↑ 7 | ↖←↑ 8 | ↖ 7 |
| t | ↑ 3 | ↖←↑ 4 | ↖←↑ **5** | ↖←↑ 6 | ↖←↑ 7 | ↖←↑ 8 | ↖ 7 | ←↑ 8 | ↖←↑ 9 | ↑ 8 |
| e | ↑ 4 | ↖ 3 | ← 4 | ↖← **5** | ← **6** | ← 7 | ←↑ 8 | ↖←↑ 9 | ↖←↑ 10 | ↑ 9 |
| n | ↑ 5 | ↑ 4 | ↖←↑ 5 | ↖←↑ 6 | ↖←↑ 7 | ↖←↑ **8** | ↖←↑ 9 | ↖←↑ 10 | ↖←↑ 11 | ↖↑ 10 |
| t | ↑ 6 | ↑ 5 | ↖←↑ 6 | ↖←↑ 7 | ↖←↑ 8 | ↖←↑ 9 | ↖ **8** | ← 9 | ← 10 | ←↑ 11 |
| i | ↑ 7 | ↑ 6 | ↖←↑ 7 | ↖←↑ 8 | ↖←↑ 9 | ↖←↑ 10 | ↑ 9 | ↖ **8** | ← 9 | ← 10 |
| o | ↑ 8 | ↑ 7 | ↖←↑ 8 | ↖←↑ 9 | ↖←↑ 10 | ↖←↑ 11 | ↑ 10 | ↑ 9 | ↖ **8** | ← 9 |
| n | ↑ 9 | ↑ 8 | ↖←↑ 9 | ↖←↑ 10 | ↖←↑ 11 | ↖←↑ 12 | ↑ 11 | ↑ 10 | ↑ 9 | ↖ **8** |

### Try it yourself!

Compute the minimum edit distance between plant and mantle.

| | # | p | l | a | n | t |
|---|---|---|---|---|---|---|
| # | 0 | 1 | 2 | 3 | 4 | 5 |
| m | 1 | 2 | 3 | 4 | 5 | 6 |
| a | 2 | 3 | 4 | 3 | 4 | 5 |
| n | 3 | 4 | 5 | 4 | 3 | 4 |
| t | 4 | 5 | 6 | 5 | 4 | 3 |
| l | 5 | 6 | 5 | 6 | 5 | 4 |
| e | 6 | 7 | 6 | 5 | 6 | 5 |

Common letters

Path

## KNOWLEDGE CLIPS 4 – Language Modeling

How to model sequences of symbols

### Intro

**Fluent**

How natural does a sequence of words/characters sound? Can we guess whether it was produced by a native or a foreigner?
Can we build a model which automatically learns which sequences are likelier in a language and which sound odd for whatever reason?

**Language processing as prediction**

Language is sequential: we constantly anticipate what's coming next and can be surprised when something happens that we didn't expect, which results in confusion and more time to make sense of what we heard/read.
Can we exploit this sequentially to build a model that learns to predict what's next to score a sequence?

**Probabilities**

Language models assign a probability to sequences of symbols by storing a probability distribution over symbol sequences.
googol is a … *p(go | search | company | number | of | ….)*
*p(The earth revolves around the sun) v.*
*p(The sun revolves around the earth).*

**Prediction**

Closely related to classification in language but concerned with anticipating what will come next given what has happened rather than with assigning a class to an instance.
What feature of language makes prediction and classification similar?

**When do we need a language model?**

Language models are a means to an end, rarely the goal.
They are useful to:

- score sentences before choosing the best one
- generate language
- investigate language processing in humans to inform the development of bots that behave like us
- much more…

**Speech/handwriting recognition**

Speech/handwriting recognition: use what we recognized before to help disambiguate the current word.
The kid smiled when he got his /aɪskriːm/.
(ice cream, ~~I scream~~)
When I'm terrorized, /aɪskriːm/ at the top of my lungs.
(~~ice cream~~, I scream)

**Spelling correction**

Spelling correction: more alternatives can have the same lowest edit distance but one fits best.
We listened to a terrible spng.
(p(song), p(~~sing~~), p(~~sang~~), p(~~sung~~))
I'm not going away until you spng.
(p(~~song~~), p(sing) p(~~sang~~), p(~~sung~~), )

**Machine translation**
Translation: equally plausible translations can be more or less fluid in the target language.
Source: Ik ga naar huis.
Ambiguity: huis → home | house
p(I'm going home.) v. p(I'm going ~~house~~)

**Generation**
If we have a model which encodes the probability of all possible continuations given a word, then we can have the model generate words given a cue.
How do we deal with the fact that if we always pick the word with the highest probability we always generate the same sentence? What is this sentence?

**Language processing**
How do we know whether a language model is good?
Good = encodes probabilities well and assigns high probability to real, correct sentences and low to wrong sentences.
Good = mimics human performance.

**A generative model**
Language models are inherently generative: they learn the probability distribution across states (words) and approximate the model which generated the observed sequences.

N-grams
**The building rocks**
N-grams are the bricks to build the simplest language models.
A word n-gram is a sequence of n words.

- Uni-gram: dog
- Bi-gram: you are
- Tri-gram: the building blocks
- Tetra-gram: city of blinding lights
- …

**Naïve Bayes as a language model**
We have seen Naïve Bayes as a classification tool. However, it encodes a sentence using n-grams and records the probability of each.
If we put all the probabilities together we can derive the probability of the sentence. Where does this solution falls short though?

**N-gram probabilities**

Predicting the next word: how likely is it that given words $w_1$ to $w_n$, we observe word $w_{n+1}$? e.g., given googol is a, how likely is it the we find number?
Predicting the likelihood of a whole sentence: out of all the sequences of n words, how likely is sequence A? e.g. out of all sequences of 6 words, how likely is The earth revolves around the sun?

**Estimation: prediction**
Take a big corpus, count how many times googol is a occurs (regardless of the continuation) and how many times it is followed by number. Then divide them to get a probability.

$$p(number \vee googol\, is\, a)=\frac{c(googol\, is\, a\, number)}{c(googol\, is\, a)}$$

Why are we guaranteed that it is a probability?

**Estimation sequences**
Count how often The earth revolves around the sun occurs in a big corpus, then count how many 6-word sequences there are, then divide.

$$p(the\, earth\, revolves\, around\, the\, sun)=\frac{c(the\, earth\, revolves\, around\, the\, sun)}{c(sequences\, of\, 6\, words)}$$

But there are a lot of 6 word sequences, way too many to estimate.

**The chain rule of probability**

$$p(w_1,\ldots,w_m)=p(w_1)*p(w_2|w_1)*p(w_3|w_1,w_2)*\ldots*p(w_m|w_1,\ldots,w_{m-1})=\text{¿}$$

$$\text{¿}\prod_{i=1}^{m} p(w_i|w_{1:i-1})$$

We can compute the joint probability of a whole sequence by multiplying conditional probabilities of words given their history, i.e. the preceding contexts

**Problems**
Language is infinite! We can always make up new words, let alone word combinations or sentences.
The larger an n–gram, the higher the chance that we won't find it anywhere in a finite corpus.
How do we solve this?

**The Markov assumption**
Approximate the conditional probability of a word given its entire history by the conditional probability of a word given its local history of n words, i.e. the preceding n-gram.

$$p(w_1,\ldots,w_m)\approx\prod_{i=1}^{m} p(w_i \vee w_{i-n:i-1})$$

**The bigram model**
Only look at the preceding word as history.
p(number | googol is a) ≈ p(number | a)
<u>Pros</u>: easier to estimate transitions, reduce sparsity.
<u>Cons</u>: throw away information since longer sequences are more constraining of following words.

### Maximum likelihood estimates

How do we find the probability that number follows a? Count and normalize!
Find and record the occurrences of a followed by number, keep track of the frequency count, then divide by the frequency count of all the bigrams which start with a, which is the frequency count of a. (Why?)

### Why maximum likelihood?

Using relative frequencies (i.e. normalized counts) entails that the resulting parameters (i.e. the probabilities of finding a word given any preceding n-gram) maximizes the likelihood of the corpus used to collect counts.
Given the model, the input corpus is the likeliest set of sequences we could expect.

### What happens at sequence boundaries?

When dealing with n-gram models we need to augment sequences with a symbol marking the beginning (BoS as in Beginning of Sequence) and the end (EoS) of each sequence.
In a trigram model, we add two BoS symbols and one EoS symbol (why?)

### Log space

When working with chained probabilities it is best to convert all probabilities to logs so that we can sum instead of multiplying and avoid having very little numbers (why do we run this risk?).

$$\log p\left(w_1, \ldots, w_m\right) \approx \sum_{i=1}^{m} \log p\left(w_i \vee w_{i-n:i-1}\right)$$

### Choice of n

In practical applications, trigram (or higher) models are usually preferred to bigram models: the history used to predict the next word consists of more words.
More fine-grained models can be used when we have tons of training data: the higher the n, the more fine-grained the information, the weaker the Markov assumption, and the more severe the sparsity!

## Markov models

### Markov Chain

Markov Models, i.e. any model which makes use of the Markov assumption, are defined by the following components:

- a set of history states Q (q1, q2, q3, …, qN)
- a set of predicted states R (r1, r2, r3, …, rM)
- a transition probability matrix A (N-by-M)
- an initial probability distribution π

### States

In a bigram model, Q and R are the same set. With larger n-gram models they’re not (which is larger?)
Typically Q corresponds to the rows of the A matrix while R to the columns, such that relative frequencies are obtained by row-normalizing counts.

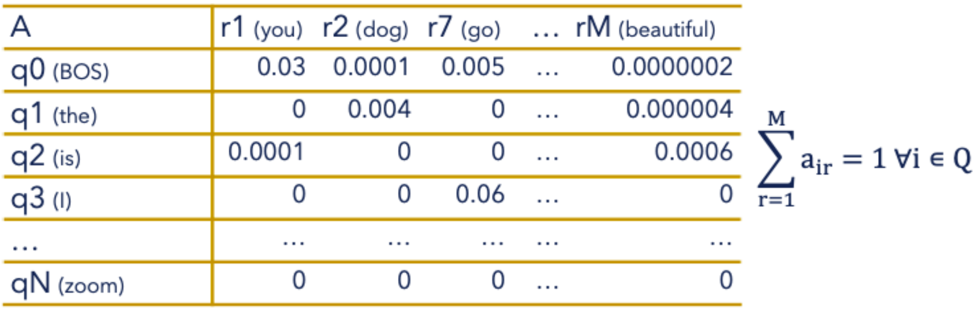
**Transition matrix**

The transition probability matrix A encodes the probability of going from a state q∈Q to a state r∈R, such that a cell $a_{ij}$∈A with i≤|Q| and j≤|R| encodes the probability of finding state j given state i.

**Initial distribution**

Usually embedded in A by augmenting Q with the BOS state.
The initial distribution is given by the transitions between states q which only consist of BoS symbols and the r states.
The initial distribution can be set in advance or estimated

**A possible example**



| A | r1 (you) | r2 (dog) | r7 (go) | … | rM (beautiful) |
|---|---|---|---|---|---|
| q0 (BOS) | 0.03 | 0.0001 | 0.005 | … | 0.0000002 |
| q1 (the) | 0 | 0.004 | 0 | … | 0.000004 |
| q2 (is) | 0.0001 | 0 | 0 | … | 0.0006 |
| q3 (I) | 0 | 0 | 0.06 | … | 0 |
| … | … | … | … | … | … |
| qN (zoom) | 0 | 0 | 0 | … | 0 |

$$\sum_{r=1}^{M} a_{ir} = 1 \; \forall i \in Q$$

Evaluation

**Extrinsic**

We expect our language model to make our speech recognizer more accurate, our spellchecker better able to detect and correct typos, our translator capable of producing more adequate and fluent sentences in the target language, our classifier more accurate, our language generator more human-like…
We expect it to improve a down-stream task.

**Very expensive**

Running end-to-end systems takes time and resources. How do we assess the language model itself to make an educated guess about its potential effect on the downstream task? Beside the practical application it’s embedded in, what do we expect a good language model to do in itself?

**Intrinsic evaluation**

Feed the model some new data and check how well it predicts each token in the sentences and how well it scores sentence probabilities.
A good language model will fit the new data well, i.e. it’ll usually predict the correct type, or, differently put, assign a high probability to the sequences in the new data.

**Perplexity**

The standard way to intrinsically evaluate whether a prediction system is working: essentially computes how surprised the system is of seeing what it actually sees in the light of what it expected to see given what it knows.

**More formally**
At any new token in the data, the language model outputs a probability for every possible type as a continuation given the previously observed history and the transition probability matrix it learned on some other data.
The higher the probability a model assigns to new valid sentences, the better the language model.

**Perplexity and probability**
Perplexity is the inverse probability of the test set under a language model, normalized by the number of tokens (the more tokens there are, the lower the final probability of a sequence).

$$p(W) = 2^{-l}$$

where $l = \frac{1}{¿W \vee ¿ \sum_{i=1}^{¿W \vee ¿ \log p(w_i \vee w_{i-n:i-1}) ¿} ¿¿}$ with $w_i \in W$ where W is a sequence of tokens.

**Best practices**
Estimate the language model (states and transition matrix) on some corpus.
Fine-tune it on some different corpus.
Test it, i.e. check how well it fits new data, a.k.a. how well it generalizes on yet another different corpus

**NEVER, NEVER, NEVER, NEVER…**
…test on data on which the model was trained or fine-tuned: learning cannot only amount to remembering for it to be useful!
Maximum likelihood estimates already maximizes the likelihood of the training set given the probabilities. Fine-tuning may fit specific biases of the corpus.

**Overfitting**
Every language model encodes specific facts about the training corpus. This entails that the longer the n-grams, the better we can fit those specificities and improve the fit.
But that’s irrelevant and deceiving because we want to use the language model on different data!

**Generalization**
Correctly set up your problem: some generalizations cannot be expected of a system, some may be trivial. Make sure that language, genre, domain, … are consistent and appropriate to the task!

**Sequence boundaries**
We need to include the BoS and EoS in the probability, so W becomes the concatenation of BoS, W, and EoS.
The EoS symbol also contributes to the count of tokens in the sequence, hence T = T + 1.

**The vocabulary constraint**
Perplexity scores can only be compared if derived on the same test set, with the same vocabulary.

Why is that?

## Smoothing

### The language problem

Language changes on a number of axes (space, time, demographics, culture, education, …). To what extent can we rely on the probabilities we derive from a sample to model language?
What about the things we didn't see?

### Unobserved ==? non-existent

In any language sample we observe a finite number of tokens, types and lemmas. Should we conclude that's everything?
How do we make sure our sample approximates the language we want to model? What do we do with the words we still miss?

### Unobserved ==? ungrammatical

Some sequences we didn't observe are not legit in a language (e.g. you dog or I beautiful run): we want our model to be very surprised by these sequences! (or what'd happen?)
Some other sequences we didn't see may be rare or very domain specific: what if our language model thinks they're ungrammatical or impossible?

### Unknown words, or OOV words

If we find a word we didn't see in training, we can't even start looking for the transition from the preceding n-gram!
The OOV rate is the percentage of words in the test set which never occur in training: if high, we have a representativeness problem.
How do we deal with situations in which the vocabulary is not known in full (open vocabulary)?

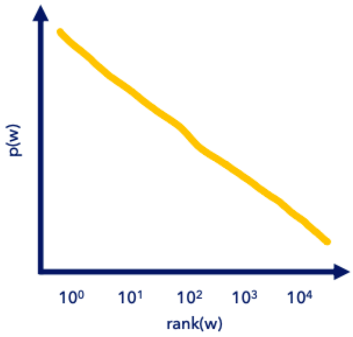
### Make it close

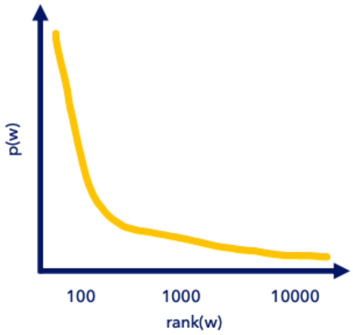
Just pretend an open vocabulary situation is a closed vocabulary problem where we know all the possible words in advance by replacing some words in training (usually the rare ones) with a single token, say , for which you derive ML estimates the usual way.
How does this relate to normalization?

### What the UNK?!

We've made sure that all words we may encounter at test also occur in training. What about transitions? What if we see a new n-gram at test which consists of known words?
Remember: language is infinite with a finite number of symbols!

**Zipfian**





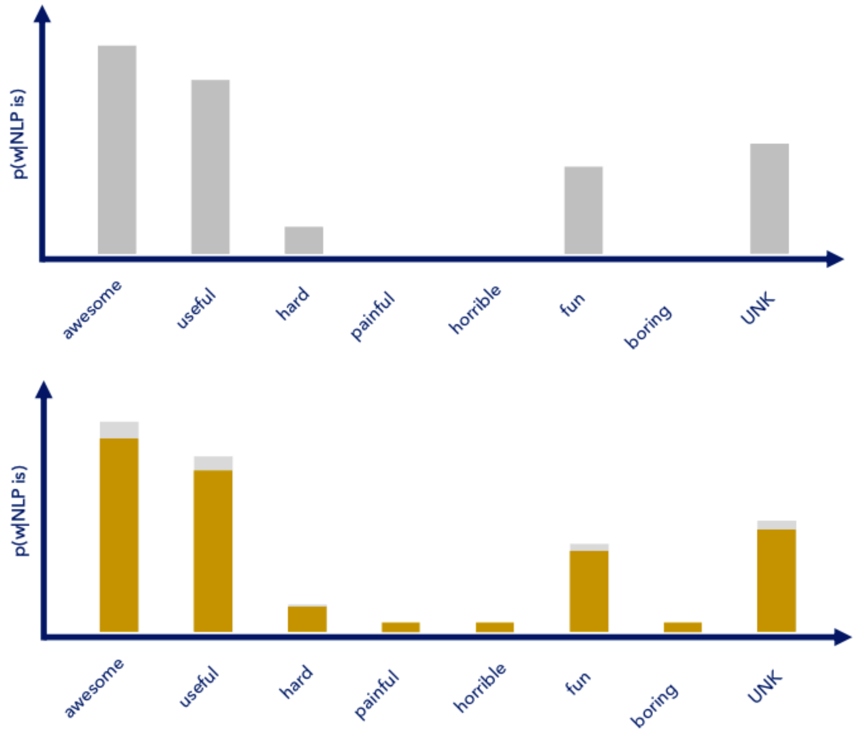
**Few frequent, many rare**
Most words in a corpus occur once, a lot occur twice, many three times, …, very few occur often.
What about things we don't observe in our sample? Likely they're even more!

**Sparsity, nothing but sparsity**
If most words don't occur in a finite corpus, and, of those that do, most occur rarely, the n-grams containing rare words are a lot!
How do we estimate these transitions?

**Can't afford 0s**
The frequency of something that never occurs in training is 0, so its probability is 0. That means we're underestimating the probability of possible continuations.
In a Markov chain, any sequence where even one transition has a 0 probability in the language model is illegal (i.e. has a 0 overall probability)! And we can't even compute perplexity (why?)

**Discounting**
Smoothing means moving probability mass around so that the transition matrix doesn't have 0s anymore.
The probability that is used to augment the 0s comes from non-zero transitions, which become somewhat less probable (smoothed or discounted).

**Increased entropy**



**Laplace in theory**

Add 1 to all frequency counts before normalization.

Mind the denominator in MLE estimates! If we add 1 to all cells in a row, we have to add the number of columns (i.e. the vocabulary, or the number of types in the corpus, i.e. |R|) to the denominator or we won't have a probability distribution.

**Laplace in action**

Transitions with a frequency of 0 get a frequency of 1, transitions with a frequency of 1 get a frequency of 2, …

$$c^{¿}=(c+1)\frac{N}{N+V}$$

Where N is the number of tokens, V is the number of types, c is a count.

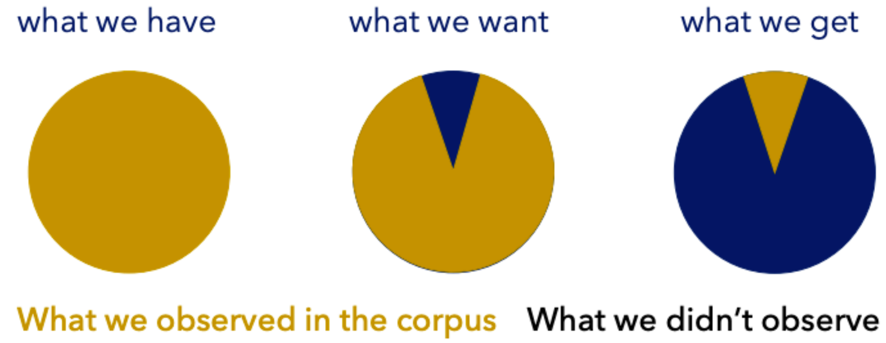
$$p^{¿}=\frac{c^{¿}}{N}$$

**Quick and dirty**

With large vocabularies and not-so-high frequencies, smoothed probabilities are too different from the nonsmoothed one.

If we give a little probability to unobserved transitions, but unobserved transitions are a lot (remember Zipf!), then we give a lot of probability to unobserved events collectively!

**A sad, sad reality**



**Add k**

Decrease the amount of probability mass which gets moved around by adding less than 1 to each count.

$$p^{¿}(w_n|w_{n-1}) = \frac{c(w_{n-1}w_n)+k}{c(w_{n-1}+kV)}$$

**Improve our solutions**

Even add k is no good: estimates have too little variance and are still off.

But we know that smaller n-grams are less sparse, so maybe we can find information there! Sometimes less is more.

**Interpolation**

We always combine evidence at different granularities, assigning a weight to each source of knowledge (typically, more to the most reliable source).

The more constraining source may be too specific or overfitting, so we always consider a more general possibility.

**Linear combinations**

Always consider transitions for smaller n-grams using a weighted linear combination of the probabilities.

$$p(w_i|w_{n-2}, w_{n-1}) = \lambda_1(w_i|w_{n-2}, w_{n-1}) + \lambda_2(w_i|w_{n-1}) + \lambda_3(w_i)$$

where $\lambda_1+\lambda_2+\lambda_3=1$, so p is a legal probability.

**Backoff**

Anytime a transition from an n-gram to a state found in the test set didn't occur in training, recursively fall back on to the smaller n-gram (from why did you to did you, for example) until a non-zero transition is found.

If the test transition is found in training, all good.

**Discounting**

When using back-off we need to discount ML estimates: if we don't, anytime we use, e.g., a bigram probability instead of a trigram probability we'd be adding mass (why is that?).

Discounting typically relies on observed transitions to estimate unobserved ones (Good-Turing algorithm).

## Fine tuning

**Choose your options**

How to decide the n-gram size? How to set the $\lambda$ s? How to decide which discounting method works best for a back-off smoothing? How to set k in the add-k smoothing?
Use the dev set! Estimate the transition matrix (or matrices, with interpolation and back-off) and find the parameters which minimize perplexity on the dev set.

**A trade-off**

Higher n-grams are more constraining and precise, but more sparse.
If we have 20K types, then there are

- $20K^2$ bigrams = $4 * 10^8$
- $20K^3$ trigrams = $8 * 10^{12}$
- $20K^4$ tetragrams = $1.6 * 10^{17}$

These are the parameters we need to estimate!

**More parameters, more data**

If our corpus is small, we won't be able to reliably estimate many parameters: in order to afford large n we need a large corpus!
A too complex model given the data means higher sparsity, higher OOV rate, brittleness, unreliability in estimated probabilities.

**Reduce parameters**

- Normalize (stemming, lemmatization, case folding, the whole toolbox)
- Use semantic classes (DoW instead of seven types)
- Consider words seen once as never seen

**In practice**: don't reinvent the wheel! Use existing and optimized libraries to do statistical language modeling, e.g., KenLM.

# KNOWLEDGE CLIPS 5 – Part-of-Speech Tagging

Assign tags to word using Hidden Markov Models

## Parts-of-Speech

**Word classes, or syntactic categories**

Parts-of-Speech (PoS) are clusters of words which have a similar behavior in a language:

- They have similar contexts of occurrence
- They can be combined with the same set of morphological affixes

Although other regularities exist (phonological and semantic, primarily), distributions are the defining property of PoS tags.

**Different degrees of specificity**

Nouns

count, mass, common, proper, masculine, feminine

Verbs

auxiliaries, participle, past, passive, gerund

Adjectives

comparative, superlatives

Pronouns

personal, possessive

…

**Open class words**
Nouns, verbs, adjectives, and adverbs
Are productive: new words are created all the time and change fast.
Change across corpora of a same language.
Typically content words (nouns, verbs, adjectives, adverbs).

**Closed class words**
Conjunctions, determiners, pronouns, prepositions, …
Are not productive: the set is pretty much fixed, change is slow.
Usually completely represented in any sizable corpus.
Typically function words with grammatical roles (determiners, pronouns, prepositions, conjunctions, …)

**Cross-linguistic validity**
Different languages have different PoS tags.
Differences are more pronounced for closed class words, but not all languages have all 4 major open classes.

**The Penn Treebank tagset**
English-specific, includes 45 tags, typically appended to a word using a slash /.
Doesn't include syntactic information (e.g. is a word the subject or the object?)
Assumes tokenized input (with multiword expressions tokenized at the white space).

**Tagged corpora**
Some corpora come with information about the PoS of every word in it:
Brown corpus (1million words, balanced)
WSJ (1million words, news articles)
Switchboard (2million words, transcribed phone conversations).
Annotations were hand corrections of an automatic system which assigned PoS to words.

## Tagging

**PoS tagging**
The process of automatically assigning PoS tags to words in a corpus. Takes a tokenized corpus as input and outputs a sequence of tags, one for each input token.

**Why do we care?**
Words in dictionaries have an indication of their syntactic category, why don't we just use that?
Ambiguity! Many surface forms can have different PoS tags and functions in language. PoS tagging aims at resolving the ambiguity in natural language.

Useful for parsing, Named Entity Recognition and coreference resolution among other things

**Type vs. tokens**
Types tend to be unambiguous (more than 80% are) but the ambiguous ones occur a lot (almost two thirds of the tokens in a corpus are ambiguous!) A reliable automatic system to disambiguate has a huge pay-off.

**Stupid baseline**
Most types in a corpus are nouns, so just assume everything is a noun: you'll get an accuracy of around 60% on types, but much much lower on tokens (<20%)
How good are these figures?

**Most frequent class**
Some words are ambiguous but have very skewed distributions: the vast majority of their occurrences correspond to a same PoS tag (cat can be a verb, but most of the times it's a noun).
So, every time an ambiguous word is found, assign its most frequent PoS tag. Accuracy gets to ~92% for tokens.

**Disambiguate**
A raft of ducks were swimming on the lake.
The boxer ducks every punch from the opponent.
ducks can be a plural noun or a third person singular: with the most frequent usage, one token above is misclassified. But context is informative!

**Which context?**
Preceding words give away a lot of information about the next word: where did we find this intuition?
Successive contexts are also informative but not always available: when is this the case?

**The problem specification**
Find some algorithm which can make use of preceding contexts to constrain the interpretation of the word being tagged and select the correct PoS among the possible ones.

## Hidden Markov Models

**Hidden states**
In PoS tagging we don't observe the states we want to predict as we do in language modeling: the PoS tags are hidden. We observe a sequence of words and want to find the best sequence of tags for that particular sequence of words out of all possible sequences of tags.

**Extend the Markov Chain**
A Hidden Markov Model (HMM) consists of the following components:

- Q: a finite set of N states
- A: a state transition probability matrix
- π: an initial probability distribution

- O: a sequence of T observations
- B: an observation likelihoods matrix

**Components: Q**

Rather than consisting of words, Q consists of PoS tags.
So, Q contains the states of the HMM. What about BoS and EoS?

**Components: A**

A is a |Q|-by-|Q| matrix, where each cell $A_{ij}$ indicates the probability of moving from state i∈Q to state j∈Q.
Hence, to compute A we need some corpus where we can observe PoS tags: we need annotated data!

**Components: π**

Similarly to the Markov Chain, encodes the probability that each state q∈Q follows the BoS symbol.
As before, we can fix it in advance if we want to embed constraints, or we can estimate it from a corpus (augmenting each PoS tag sequence with a BoS symbol).

**Components: O**

The set of observed events: in PoS tagging, O contains the words.
This set must be finite: why?
How does O relate to Q? Can there be an event oi whose corresponding tag q∉Q?

**Components: B**

B is a |Q|-by-|O| matrix, where each cell bqo indicates the probability that a word o∈O is generated by a state q∈Q.
To compute B, we need to find out how often each word occurs tagged with a particular PoS tag in a corpus. We need annotated data!

**A = p($t_i$|$t_{i-n:i-1}$)**

A encodes the transition probabilities across states, hence across PoS tags (or tag transition probabilities).
We estimate them from an annotated corpus using ML estimates:

$$p\left(t_i \middle| t_{i-n:i-1}\right) = \frac{c\left(t_{i-n:i-1}, t_i\right)}{c\left(t_{i-n:i-1}\right)}$$

How often is the PoS tag n-gram $t_{i-n:i-1}$ followed by the PoS tag $t_i$ ?

B = p($w_i$|$t_i$)

B encodes the probability that a certain word occurs since we observed a certain tag: given that we observed a noun, how likely is it that this noun is exactly dog and not *aardvark*?
Again, we use ML estimates in this form:

$$p\left(w_i \middle| t_i\right) = \frac{c\left(t_i, w_i\right)}{c\left(t_i\right)}$$

### Flip the problem

We want to know the likeliest tag for a word, but we compute the likeliest word after observing a tag. How’s that? What does this remind you of?
We aim for the posterior but we compute the likelihood and the prior, and then estimate the posterior. What is the likelihood, A or B?

### Two key assumptions

Markov assumption: the probability of the next tag is only determined by the local history, not the whole sequence.
Output independence: the probability of a word only depends on the state that produced the corresponding tag, not on any other state or word.

## Decoding

### Pick the best tags

The task of determining the sequence of hidden variables given a sequence of observed events.
The decoding task takes an estimated HMM λ(A, B, π) and a sequence of observations O as input to output the likeliest sequence of states Q to have generated O.

### Formally

$$t_{1:n} = argmax_{1:n} p\left(t_{1:n} \vee w_{1:n}\right)$$

We replace the posterior with the product of likelihood and prior, following Bayes rule:

$$t_{1:n} = argmax_{1:n} p\left(t_{1:n}\right) p\left(t_{1:n} \vee w_{1:n}\right)$$

We plug in the formulas to estimate both terms under the Markov assumption and the output independence hypotheses:

$$t_{1:n} = argmax_{1:n} \prod_{i=1}^{n} p\left(t_i \vee t_{i-n:i-1}\right) p\left(t_i \vee w_i\right)$$

### Transition and emission

The posterior is the product of the transition probability from PoS tag n-grams to PoS tag and the emission probability from PoS tag to word.
Transition probabilities (hence A) captures the prior: how likely is this tag in this context?
Emission probabilities (hence B) capture the likelihood: how likely is it that, given a certain tag, we would observe precisely this word?

## Viterbi algorithm

### All the possible sequences

Given a sequence of 4 words and the Penn tag-set of 45 tags there are $45^4$ possible hidden sequences, i.e. 4,100,625. With a sequence of 5 words, they become 184,528,125. Try imagine with a sentence of normal length.

### Dynamic programming to the rescue

The dynamic programming solution to the decoding step in HMMs is called Viterbi algorithm.
The inventor, bless him, did not patent it. Nowadays it’s used for pretty much every application of HMMs

### Basic principle

Recursively compute the best path that could lead us to a certain point given:

- The HMM $\lambda$=(A,B,π)
- The observations up to that point
- The most probable state sequence to have generated the observations given the HMM $\lambda$

### Recursion

When we are in a state, we have already computed the best path that led us there, so there’s no need to recompute it.
If we move from that cell, the best path through the states which brought us there will necessarily be part of the total best path (under our model).

### Cutting complexity

Estimating the total number of tag sequences given a word sequence is exponential on the length T of the word sequence: $O(Q^{|T|})$
Using the Viterbi algorithm, the complexity becomes linear in T, since at every new event we have to compute the probabilities from the previous states to the current: $O(Q^nT)$ where n is the order of the model (2: bigram, 3: trigram, …)

### In practice

Create a matrix M with as many rows as there are states $q \in Q$ and as many columns as there are events $o \in T$ to be decoded plus the BoS and EoS symbols.
The value of each cell $M_{qo}$ is computed as follows:

$$p\left(t_o \middle| t_{o-n:o-1}\right) p\left(w_o \vee t_o\right)$$

Hence each cell contains the posterior probability of finding each tag given the current word.

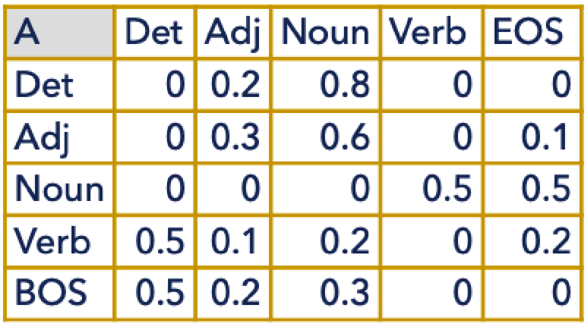
### The dynamic side of things

Each posterior depends on the emission and the local transition: at each word $w_t$ with $t \in T$, compute the posteriors considering all the emission probabilities and the local history.
Then take the cell with highest posterior and keep track of which transition brought you there.
When you reach the EoS word, back track your steps through the maximum a posteriori sequence of tags.

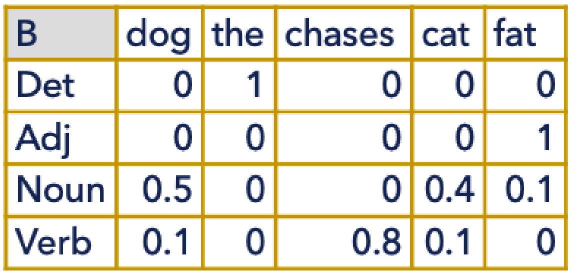
## A worked-out example

### An estimated HMM

$A_{ij}$ encodes the probability that the tag in column j occurs given that the tag in row i has. $B_{ik}$ encodes the probability that word k is observed given that tag i was (rows sum to 1).



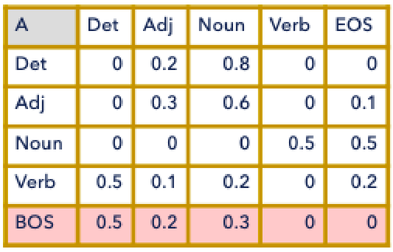
| A | Det | Adj | Noun | Verb | EOS |
|---|---|---|---|---|---|
| Det | 0 | 0.2 | 0.8 | 0 | 0 |
| Adj | 0 | 0.3 | 0.6 | 0 | 0.1 |
| Noun | 0 | 0 | 0 | 0.5 | 0.5 |
| Verb | 0.5 | 0.1 | 0.2 | 0 | 0.2 |
| BOS | 0.5 | 0.2 | 0.3 | 0 | 0 |



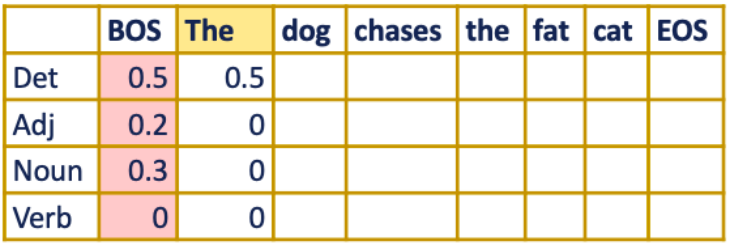
| B | dog | the | chases | cat | fat |
|---|---|---|---|---|---|
| Det | 0 | 1 | 0 | 0 | 0 |
| Adj | 0 | 0 | 0 | 0 | 1 |
| Noun | 0.5 | 0 | 0 | 0.4 | 0.1 |
| Verb | 0.1 | 0 | 0.8 | 0.1 | 0 |

**Start**

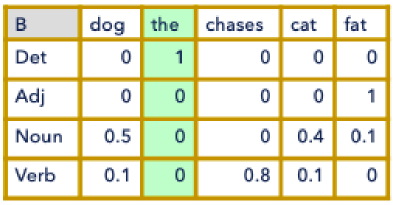
Multiply the transition probability and the emission probability, then store the result in a trellis v, indicating the probability of the previous path and store the partial path.



| A | Det | Adj | Noun | Verb | EOS |
|---|---|---|---|---|---|
| Det | 0 | 0.2 | 0.8 | 0 | 0 |
| Adj | 0 | 0.3 | 0.6 | 0 | 0.1 |
| Noun | 0 | 0 | 0 | 0.5 | 0.5 |
| Verb | 0.5 | 0.1 | 0.2 | 0 | 0.2 |
| BOS | 0.5 | 0.2 | 0.3 | 0 | 0 |



| | BOS | The | dog | chases | the | fat | cat | EOS |
|---|---|---|---|---|---|---|---|---|
| Det | 0.5 | 0.5 | | | | | | |
| Adj | 0.2 | 0 | | | | | | |
| Noun | 0.3 | 0 | | | | | | |
| Verb | 0 | 0 | | | | | | |



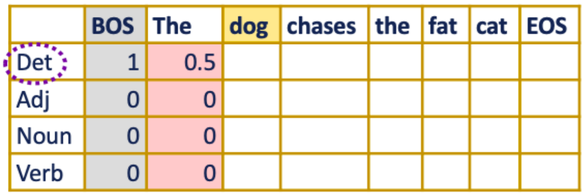
| B | dog | the | chases | cat | fat |
|---|---|---|---|---|---|
| Det | 0 | 1 | 0 | 0 | 0 |
| Adj | 0 | 0 | 0 | 0 | 1 |
| Noun | 0.5 | 0 | 0 | 0.4 | 0.1 |
| Verb | 0.1 | 0 | 0.8 | 0.1 | 0 |

**The trellis**

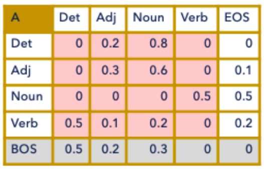
Represents the probability that the HMM is in state q∈Q after seeing the previous events $o_{1,\ldots,t} \in O$ and passing through the most probable sequence of states.

The value of each cell at successive states is computed by multiplying the current transition and emission probabilities by the most probable sequence which could lead to the cell.

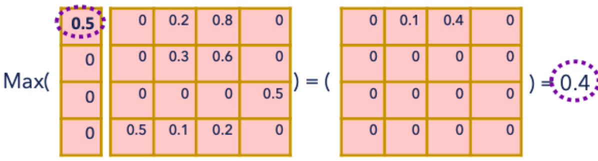
**The dynamic part: transitions**



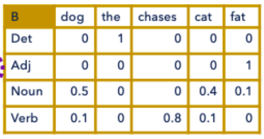
| | BOS | The | dog | chases | the | fat | cat | EOS |
|---|---|---|---|---|---|---|---|---|
| Det | 1 | 0.5 | | | | | | |
| Adj | 0 | 0 | | | | | | |
| Noun | 0 | 0 | | | | | | |
| Verb | 0 | 0 | | | | | | |



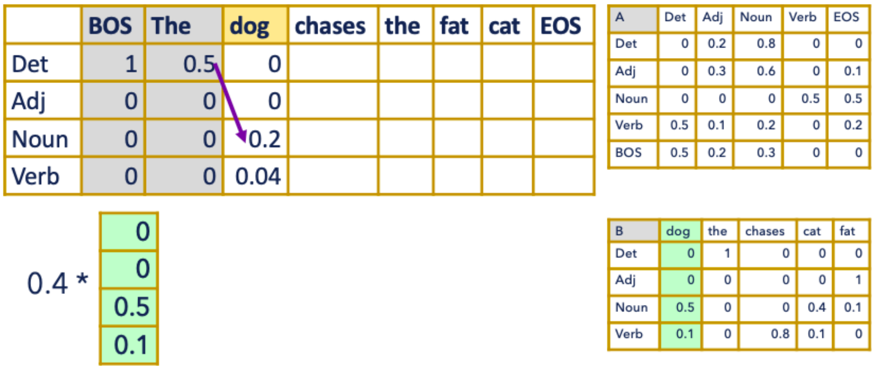
| A | Det | Adj | Noun | Verb | EOS |
|---|---|---|---|---|---|
| Det | 0 | 0.2 | 0.8 | 0 | 0 |
| Adj | 0 | 0.3 | 0.6 | 0 | 0.1 |
| Noun | 0 | 0 | 0 | 0.5 | 0.5 |
| Verb | 0.5 | 0.1 | 0.2 | 0 | 0.2 |
| BOS | 0.5 | 0.2 | 0.3 | 0 | 0 |



$$\text{Max}\left(\begin{array}{|c|}\hline 0.5 \\ \hline 0 \\ \hline 0 \\ \hline 0 \\ \hline\end{array}\;\begin{array}{|c|c|c|c|}\hline 0 & 0.2 & 0.8 & 0 \\ \hline 0 & 0.3 & 0.6 & 0 \\ \hline 0 & 0 & 0 & 0.5 \\ \hline 0.5 & 0.1 & 0.2 & 0 \\ \hline\end{array}\right) = \left(\begin{array}{|c|c|c|c|}\hline 0 & 0.1 & 0.4 & 0 \\ \hline 0 & 0 & 0 & 0 \\ \hline 0 & 0 & 0 & 0 \\ \hline 0 & 0 & 0 & 0 \\ \hline\end{array}\right) = 0.4$$



| B | dog | the | chases | cat | fat |
|---|---|---|---|---|---|
| Det | 0 | 1 | 0 | 0 | 0 |
| Adj | 0 | 0 | 0 | 0 | 1 |
| Noun | 0.5 | 0 | 0 | 0.4 | 0.1 |
| Verb | 0.1 | 0 | 0.8 | 0.1 | 0 |



| | BOS | The | dog | chases | the | fat | cat | EOS |
|---|---|---|---|---|---|---|---|---|
| Det | 1 | 0.5 | 0 | | | | | |
| Adj | 0 | 0 | 0 | | | | | |
| Noun | 0 | 0 | 0.2 | | | | | |
| Verb | 0 | 0 | 0.04 | | | | | |

$$0.4 * \begin{pmatrix}0\\0\\0.5\\0.1\end{pmatrix}$$

| A | Det | Adj | Noun | Verb | EOS |
|---|---|---|---|---|---|
| Det | 0 | 0.2 | 0.8 | 0 | 0 |
| Adj | 0 | 0.3 | 0.6 | 0 | 0.1 |
| Noun | 0 | 0 | 0 | 0.5 | 0.5 |
| Verb | 0.5 | 0.1 | 0.2 | 0 | 0.2 |
| BOS | 0.5 | 0.2 | 0.3 | 0 | 0 |

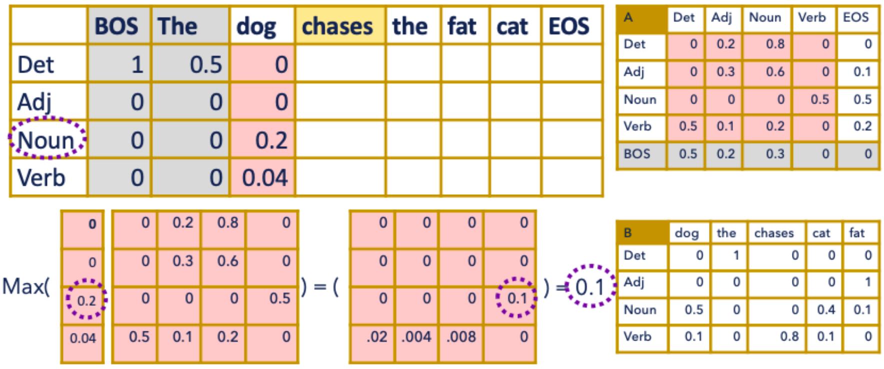
| B | dog | the | chases | cat | fat |
|---|---|---|---|---|---|
| Det | 0 | 1 | 0 | 0 | 0 |
| Adj | 0 | 0 | 0 | 0 | 1 |
| Noun | 0.5 | 0 | 0 | 0.4 | 0.1 |
| Verb | 0.1 | 0 | 0.8 | 0.1 | 0 |

**The Viterbi probability**

In order to update the posterior probability of observing each tag given the sequence up to that observed event, we need three pieces of information:

- the posterior probability up to the previous word (the trellis)
- the transition probabilities from state $q_i$ to state $q_j$
- the emission probabilities for observation $o_j$ given state $q_j$

**One more step: transitions**



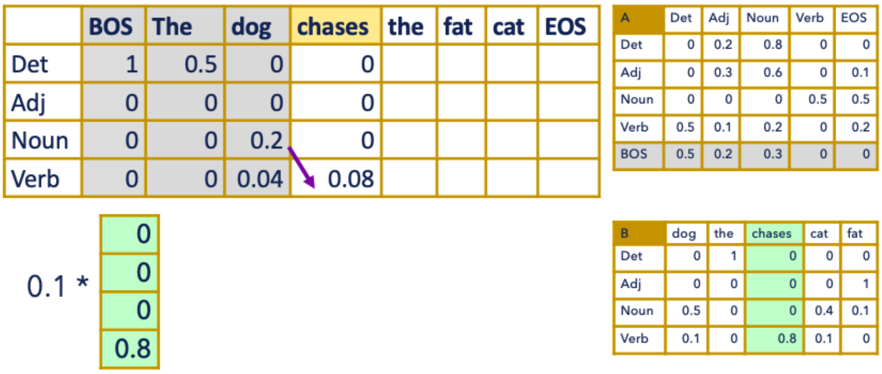
| | BOS | The | dog | chases | the | fat | cat | EOS |
|---|---|---|---|---|---|---|---|---|
| Det | 1 | 0.5 | 0 | | | | | |
| Adj | 0 | 0 | 0 | | | | | |
| Noun | 0 | 0 | 0.2 | | | | | |
| Verb | 0 | 0 | 0.04 | | | | | |

$$\mathrm{Max}( \begin{pmatrix}0\\0\\0.2\\0.04\end{pmatrix} \begin{pmatrix}0 & 0.2 & 0.8 & 0\\0 & 0.3 & 0.6 & 0\\0 & 0 & 0 & 0.5\\0.5 & 0.1 & 0.2 & 0\end{pmatrix} ) = ( \begin{pmatrix}0 & 0 & 0 & 0\\0 & 0 & 0 & 0\\0 & 0 & 0 & 0.1\\.02 & .004 & .008 & 0\end{pmatrix} ) = 0.1$$

| A | Det | Adj | Noun | Verb | EOS |
|---|---|---|---|---|---|
| Det | 0 | 0.2 | 0.8 | 0 | 0 |
| Adj | 0 | 0.3 | 0.6 | 0 | 0.1 |
| Noun | 0 | 0 | 0 | 0.5 | 0.5 |
| Verb | 0.5 | 0.1 | 0.2 | 0 | 0.2 |
| BOS | 0.5 | 0.2 | 0.3 | 0 | 0 |

| B | dog | the | chases | cat | fat |
|---|---|---|---|---|---|
| Det | 0 | 1 | 0 | 0 | 0 |
| Adj | 0 | 0 | 0 | 0 | 1 |
| Noun | 0.5 | 0 | 0 | 0.4 | 0.1 |
| Verb | 0.1 | 0 | 0.8 | 0.1 | 0 |

**One more step: emissions**



| | BOS | The | dog | chases | the | fat | cat | EOS |
|---|---|---|---|---|---|---|---|---|
| Det | 1 | 0.5 | 0 | 0 | | | | |
| Adj | 0 | 0 | 0 | 0 | | | | |
| Noun | 0 | 0 | 0.2 | 0 | | | | |
| Verb | 0 | 0 | 0.04 | 0.08 | | | | |

$$0.1 * \begin{pmatrix}0\\0\\0\\0.8\end{pmatrix}$$

| A | Det | Adj | Noun | Verb | EOS |
|---|---|---|---|---|---|
| Det | 0 | 0.2 | 0.8 | 0 | 0 |
| Adj | 0 | 0.3 | 0.6 | 0 | 0.1 |
| Noun | 0 | 0 | 0 | 0.5 | 0.5 |
| Verb | 0.5 | 0.1 | 0.2 | 0 | 0.2 |
| BOS | 0.5 | 0.2 | 0.3 | 0 | 0 |

| B | dog | the | chases | cat | fat |
|---|---|---|---|---|---|
| Det | 0 | 1 | 0 | 0 | 0 |
| Adj | 0 | 0 | 0 | 0 | 1 |
| Noun | 0.5 | 0 | 0 | 0.4 | 0.1 |
| Verb | 0.1 | 0 | 0.8 | 0.1 | 0 |

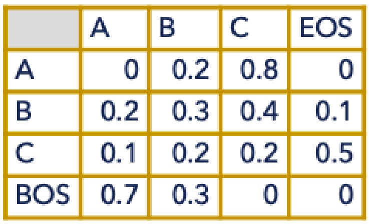
**Tom Thumb**

At each event, record which state q∊Q has the highest posterior probability (computed using the transition probabilities, the emission probabilities AND THE PREVIOUS POSTERIOR PROBABILITY) and how you got there.

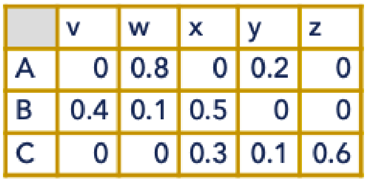
When you hit the EoS word, do like Tom Thumb and follow your steps back, collecting the crumbs you left behind.

**Try it yourself!**

Given the HMM in the tables, find the best sequence of tags for the string **w v y**. Fill out the trellis below:



| | A | B | C | EOS |
|---|---|---|---|---|
| A | 0 | 0.2 | 0.8 | 0 |
| B | 0.2 | 0.3 | 0.4 | 0.1 |
| C | 0.1 | 0.2 | 0.2 | 0.5 |
| BOS | 0.7 | 0.3 | 0 | 0 |



| | v | w | x | y | z |
|---|---|---|---|---|---|
| A | 0 | 0.8 | 0 | 0.2 | 0 |
| B | 0.4 | 0.1 | 0.5 | 0 | 0 |
| C | 0 | 0 | 0.3 | 0.1 | 0.6 |

| | BOS | w | v | y | EOS |
|---|---|---|---|---|---|
| A | 0.7 | 0.7*0.8=0.56 | 0 | 0.014 | 0 |
| B | 0.3 | 0.3*0.1=0.03 | 0.1792 | 0 | 0.1 |
| C | 0 | 0 | 0 | 0.007 | 0.5 |

0.448 (A)

0.07(B)

0.0112 (A)

Extras

**More context**

2grams are not very constraining in terms of context, 3grams are better (4grams would be even better, what problems may there be?).

We can easily extend Viterbi: rather than taking the maximum over transitions from each posterior in the previous column, we do it over posteriors from two (or n) previous columns.

**Brings sparsity**

It's already impossible to estimate all possible state transitions in a trigram model: in our example we worked with 0s, but in real life we don't want to. 0s may signal an impossible transitions as well as one we didn't have enough data to observe!

We don't want 0 transitions to hijack the computation…

**Interpolation**

Same as in language modeling: the transition probability is estimated by linearly interpolating using smaller n-grams, until the unigram. When we can't have everything, we still can use something.

And something is way better than nothing.

**Unknown words**

Leverage (pseudo)-morphology: words consist of smaller units, which influence their PoS tag!

Hence, also compute emission probabilities for partwords (typically suffixes of different lengths, down to the single character): whenever you hit an unknown word, use the emission probability for the suffixes! Something's better than nothing.

**The forward algorithm**

Similar to the Viterbi algorithm, but used to compute the likelihood of a sequence of observed events given a HMM $\lambda(A,B,\pi)$.

Replace the argmax by the sum: you want to take into account the probability of all possible paths through the hidden states, not find the likeliest path. Also, there are no back-pointers.

## KNOWLEDGE CLIPS 6 – Formal languages and syntax

Describe the latent structure

**What is a formal language?**

A formal language consists of a possibly infinite set of strings (words) whose constituting symbols (letters) are drawn from an alphabet and are well-formed according to a given set of rules.

**Rules**

Formal languages are characterized by grammars.
A grammar consists of

- symbols
- rules describing how a certain symbol can be legitimately rewritten in a different form.

**Terminals and non-terminals**

Rules have the form LHS → RHS

LHS is always a non-terminal symbol and RHS can be different things depending on how much expressive power the grammar in use has. The arrow means 'can be rewritten as'.

**The components of a grammar**

1. N: a finite set of non-terminal symbols (states)
2. $\Sigma$: a finite set of terminal symbols, disjoint from N
3. $P$: a finite set of production rules indicating how to rewrite non-terminals as combinations of terminals and non-terminals
4. $S \in N$: a distinguished start non-terminal state from N

**Generate**

Grammars can be used to generate symbols from non-terminals, in which case each rule specifies how to transform each non-terminal into symbols. To generate, we read rules left to right.

**Recognize**

Grammars can also be used to recognize a string and assign a structure to it, if possible given the specified rules. In this case we read rules right to left.

**Constituency and abstractions**

Grammars and rewrite rules can be seen as abstractions that move from specific symbols to compact units (and vice versa), which crucially behave in the same way regardless of where they are. These abstractions are called constituents.

**How do you put things together?**

The same type of constituent, regardless of its constituent symbols, can occur with similar constituents, even though it can be the case that the constituent symbols could not occur there alone. Constituents are abstract building blocks which disregard specificities of elements.

**Structure**

Grammars characterize the structure of a language, with no concern whatsoever for their semantic: strings don't have truth values within a given world, they can simply be well-formed or not as determined by the grammar we are using to analyze them.

**Expressive power**

Depending on their underlying formalism, formal languages can express different sets of ideas, of varying complexity.

The more expressive power a formal language has, the harder it is to characterize it computationally.

**Chomsky's Normal Form**

A grammar is in CNF if

- there are no rules generating the empty string
- rules have either terminals or sequences of two non-terminals as the right-hand side.

Grammars in CNF are binary branching: each node has two siblings, unless it generates a terminal symbol.

**Context Free Grammars**

In CFGs, rules can be of any form which belongs to the infinite set

$$N \rightarrow (N \cup \Sigma)^*:$$

the RHS can consist of any ordered combination of terminals and non-terminals, of any length. The LHS always consists of one non-terminal.

**Mind the finiteness**

P is a finite set of production rules even if rules can come from the infinite set $(N \cup \Sigma)^*$.

If we had an infinite set of production rules, it would be impossible to either generate sequences of terminals from the start state or to assign a structure of states given a sequence of terminals.

**Derivation**

One string A derives another string B if it can be rewritten as B through the application of a series of rules in the grammar.

We can define a language $\mathcal{L}$ generated by a grammar G as the set of strings composed of terminals in $\langle N,\Sigma,P,S\rangle$ that can be derived from the start symbol S through the application of rules in P defined over $(N \cup \Sigma)^*$.

## Syntax

**Arrangement**

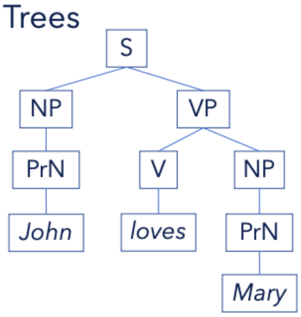
Syntax refers to the way words in natural language are arranged together following rules.

Syntax is expressed using formal languages, since it's not concerned with semantics and truth values but just with determining the structure of a set of strings and whether it's part of the specified grammar.

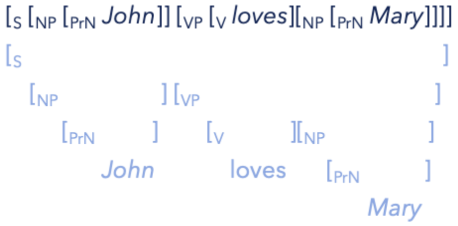
Trees and brackets

Derivations are presented in two ways:

Trees



Brackets


$[_S\,[_{NP}\,[_{PrN}\,\textit{John}]]\,[_{VP}\,[_V\,\textit{loves}][_{NP}\,[_{PrN}\,\textit{Mary}]]]]$

**Treebanks**

Treebanks store syntactic trees.
Treebanks provide corpora consisting of syntactically annotated sentences: the Penn Treebank has annotated the Brown, Switchboard, ATIS, and WSJ corpora (and some in Arabic and Chinese). More exist for other languages.

**Features of the Penn Treebank**

The Penn Treebank provides sentences using bracketing to mark constituents.
Some phrases come with annotations about their grammatical function and semantic function.

**Treebanks as grammars**

The syntactic annotations implicitly constitute a grammar G consisting of the set of non-terminals, the set of words, the rules, and the start state. So we could build a grammar from the text: how can this help us?
The Penn Treebank has a relatively flat grammar, with many long rules (not very CNF compliant).

**What are S, NP, VP, PrN, V?**

Non-terminals can be of two types:

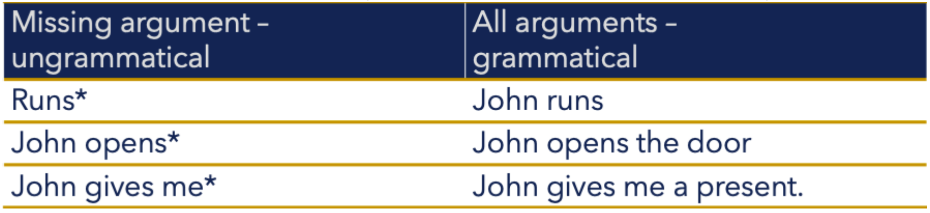
- pre-terminals like PrN and V are Parts-of-Speech (or atomic non-terminals) and production rules indicate which words they can generate.
- Constituents (NP and VP): abstract units which absolve complex syntactic functions (e.g. being a subject or an object). S, for sentence, is our start state.

**Clauses**

S can also occur as the RHS of a rule: what characterizes S structures then?
They're self-sufficient, they form a complete thought. More formally, S is a node of the parse tree below which the main verb of S has all its arguments.

**Arguments**

The NPs which are necessary for a verb to be used correctly.



| Missing argument – ungrammatical | All arguments – grammatical |
|---|---|
| Runs* | John runs |
| John opens* | John opens the door |
| John gives me* | John gives me a present. |

**Examples of NPs**

*a stop*
*John's mom's coat*
*flights*
*44 cats*
*the other tasty croissant*
*many thanks*

*all students from Italy*
*my paintings stored at home*
*a chair that spins*
*the last man to stand*
*the best man*
*…*

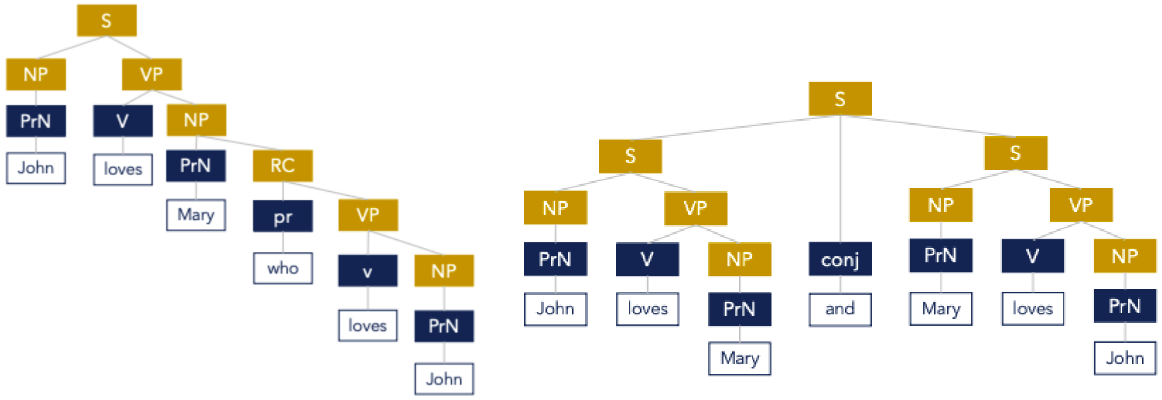
**VPs**

The verb phrase consists of the head verb and a number of optional other constituents, depending primarily on the argument structure of the head verb.
VPs can consist of a verb and an NP; a verb and a sentential complement, i.e. an entire embedded sentence; a verb and another VP; and other structures.

**Coordination**

NP → NP and NP
A coordinate phrase is the conjunction of two phrases of the same type, to yield a phrase of the same type.
Often used as a test of constituency: if two phrases can be coordinated without violating any rule, then they are constituents.

**Complexity**



Length is a good cue for complexity but a shallow one: tree depth can be a more fine-grained indicator.

## Parsing

**A definition**

Syntactic parsing, often parsing for short, is the problem of mapping from a string of words to its parse tree given a grammar G.
CFGs are purely declarative: they don't say how to to compute the parse tree for a given sentence. How do we do it?

**Parsing as recognition**

Parsing algorithms check whether a given string is grammatical under a specified grammar, without actually assigning a specific parse tree to the string.
How does this relate to the forward algorithm in PoS tagging?

**Brute force won't do it**

You can generate all possible parses and see whether one does the job, but generating all possible parses under a grammar can quickly become infeasible (exponential growth with the elements in the input string).

**Context free**

Since the grammar is context-free, we can generate parses for sub-strings and if parses are possible, we're good. Context can't change our parse of the string: if a constituent had a parse in the grammar, it'll always have one.
We know how to solve a big problem by combining solutions to sub-problems.

CKY

**A string and a grammar**

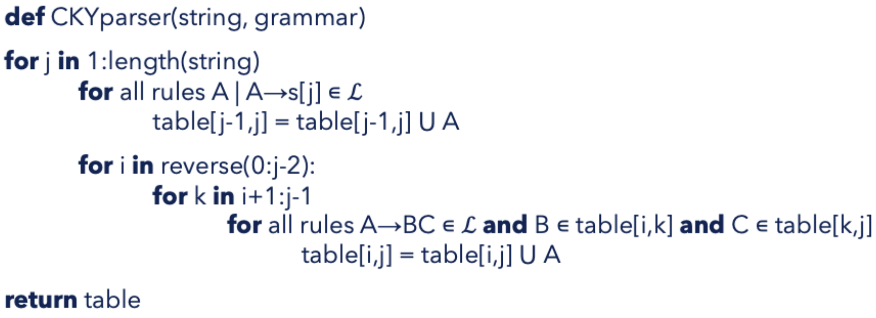
$\mathcal{L}$:
S → AB | BC
A → BA | a
B → CC | b
C → AB | a
s: abba

The grammar needs to be in Chomsky's Normal Form (CNF). In a CNF grammar rules are restricted to the forms:

- A → BC
- A → a

where capital letters indicate non-terminals and lowercase letters indicate terminals.
Remember: any CFG can be rewritten in CNF format!

**Pseudo-code**

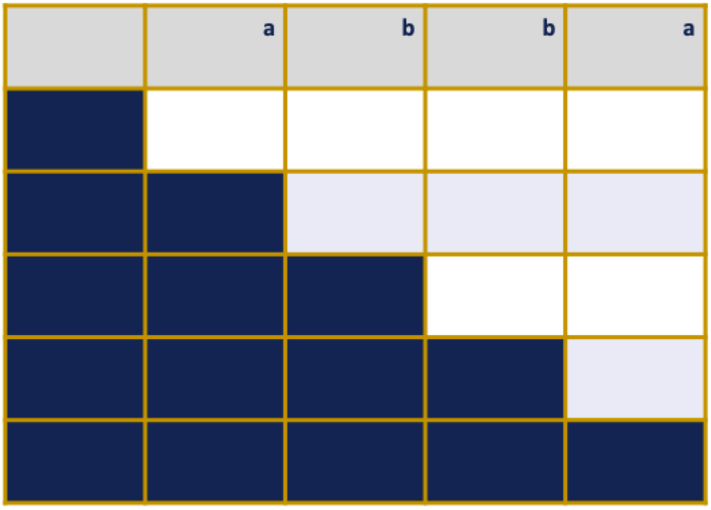


```
def CKYparser(string, grammar)

for j in 1:length(string)
        for all rules A | A→s[j] ∈ ℒ
                table[j-1,j] = table[j-1,j] U A

        for i in reverse(0:j-2):
                for k in i+1:j-1
                        for all rules A→BC ∈ ℒ and B ∈ table[i,k] and C ∈ table[k,j]
                                table[i,j] = table[i,j] U A

return table
```

**A 2 dimensional table**

$\mathcal{L}$:
S → AB | BC
A → BA | a
B → CC | b
C → AB | a

|  | y | y | x | z | x |
|---|---|---|---|---|---|
|  | B, C | S, A | A, C, S | S, A, C | A,**S**,B,C |
|  |  | B, C | A | S, B, C | A, C |
|  |  |  | A, B | S, C | S, A |
|  |  |  |  | B | C |
|  |  |  |  |  | A, B |
|  |  |  |  |  |  |



|  | a | b | b | a |
|---|---|---|---|---|
|  |  |  |  |  |
|  |  |  |  |  |
|  |  |  |  |  |
|  |  |  |  |  |
|  |  |  |  |  |

ℒ:
S → AB | BC | CC
A → BA | CB | w | x
B → CC | x | y | z
C → AB | BA | AA | y

Target string: *y y x z x*

- **Long example in the slides**

**Try it yourself!**

The string belongs to the grammar.

BLOCK 2

KNOWLEDGE CLIPS 8 – Parsing

Probabilities, chunks, dependencies

PCFGs

**Structural ambiguity**

Occurs when more than one parse tree can be assigned to a sentence based on the rules in the grammar. Examples are attachment ambiguity and coordination ambiguity.

*I shot an elephant in my pajamas.*

*Tasty sandwiches and flowers make my grandma happy*

**Parsing as recognition**

The CKY only tells us if a sentence is well-formed given a grammar, not which parse best captures the sentence structure.

Sometimes only one parse is possible, but how do we choose if two or more are valid?

**Resolving ambiguity**

Rules in CFGs can be augmented with probabilities indicating how likely each right-hand component is given a LHS in order to compute the likelihood of each parse tree given the grammar.

We call Probabilistic Context Free Grammars (PCFGs) the CFGs whose rules are augmented with probabilities

**How to extend a CFG to make a PCFG**

In a (CNF) CFG, each rule has the form X → YZ or X → y.

In a (CNF) PCFG, each rule is augmented with a probability indicating how likely the RHS is given the LHS:

X → YZ[p(YZ|X)] or X → y[p(y|X)]

**The probability of a parse tree T**

The product of the probabilities of each rule used to expand the non-terminals in the parse tree T.

Given rules of the form LHS → RHS, the probability of any parse tree over a sentence s is defined as

$$p(T,s)=\prod_{i=1}^{¿T\vee ¿\, p(RHS_i \vee LHS_i)¿} ¿$$

**Simplify a bit**

p($T_i$,s) represents two things:

- the joint probability of sentence s and parse tree i
- the probability of the parse tree i (what we want!), since

p($T_i$,s) = p(s|$T_i$) p($T_i$)

boils down to

p($T_i$,s) = p($T_i$)

as p(s|$T_i$) is 1, because a parse tree includes all words in s.

**Argmax**

Out of all the parse trees that can parse s (s is called the yield of a parse tree T), the disambiguation algorithm picks the parse tree $\hat{T}$ with the highest probability given s:

$$\hat{T}(s)=argmax\, p(T|s)=argmax\frac{p(T,s)}{p(s)}=argmax\, p(T,s)=argmax\, p(T)$$

with T such that yield(T)=S, since <u>we don't care about parse trees with a different yield than the sentence we're parsing</u>!

**The probability of a sentence**

PCFGs assign probabilities to strings of words, i.e. sentences, just like a language model would.

If the sentence s is unambiguous, its probability under the PCFG is the probability of its only parse tree with yield s.

If s is ambiguous, then its probability is the sum over the probabilities of all the parse trees with yield s.

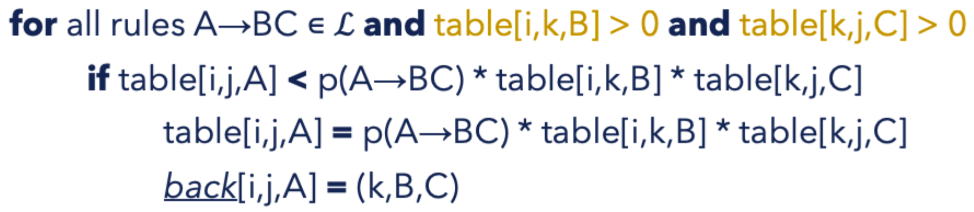
**Probabilistic CKY**

If the input sentence s is of length n, we represent it using a tensor t of dimensionality

(n+1) x (n+1) x V

Where V is the size of the set of non-terminals. Each cell t[*i,j,*A] contains the probability of constituent of type A to span positions from *i* to *j*.

**A crucial modification to the algorithm**



```
for all rules A→BC ∈ ℒ and table[i,k,B] > 0 and table[k,j,C] > 0
    if table[i,j,A] < p(A→BC) * table[i,k,B] * table[k,j,C]
        table[i,j,A] = p(A→BC) * table[i,k,B] * table[k,j,C]
        back[i,j,A] = (k,B,C)
```

**The trace**

Whenever we update table[*i*,*j*,A] because we found a parse from *i* to *j* with a higher probability, we want to record where we were coming from in back[*i,j,*A], so that if and when we find the state S in the final cell, we can recompute the best parse by going through the best sub-parses.

**How do we get the probabilities?**

Treebanks!

We count how many times each expansion of a non-terminal occurs and normalize by the occurrence of the non-terminal, the usual stuff.

Problems of PCFGs

**It's all nice but…**

- Independence assumption: useful for computational tractability, bad for capturing structural dependencies, since probabilities are assumed to be independent of each other
- Lack of lexical information: CFGs don't model syntactic restrictions which depend on specific lexical items, resulting in a poor solution to structural ambiguities.

**Independence: Context freedom**

In CGFs, the expansion of a non-terminal is independent of context, i.e. of neighboring non-terminals in the parse tree.

In PCFGs, the probability of a given rule is independent of the rest of the parse tree.

That's why we can multiply probabilities to get the parse tree probability.

**Context dependence**

In natural languages, the choice of how to expand a node depends very much on neighboring non-terminals in the parse.

Example: NPs in subject position tend to be pronouns, while NPs in object position tend to be non-pronominal.

**Splitting non-terminals**

We could then have two NPs, one for subject positions and one for object position: this way we could estimate a probability distribution over rules for $NP_{subj}$ and a separate probability distribution over rules for $NP_{obj}$.

This requires very fine-grained annotations. Can we leverage the parse tree itself to split non-terminals?

**Parent annotation**

Each node is annotated with the non-terminal which directly dominates it, i.e. the non-terminal of its parent node.

Subject NPs are dominated by S, while object NPs are dominated by VPs, hence we have NP^S and NP^VP.

We can do the same for pre-terminal nodes, as in cat^N v. cat^V.

**Overfitting**

If we split too much to accommodate the training set, we lose generalization power: we have to pick the right granularity for splits of non-terminals and pre-terminals.

Or use algorithms that do this for us, e.g. the split and merge, which aims to find the best set of non-terminals trading likelihood and generalization.

**Remember the two issues!**

- Independence assumption: useful for computational tractability, bad for capturing structural dependencies, since probabilities are assumed to be independent of each other
- Lack of lexical information: CFGs don't model syntactic restrictions which depend on specific lexical items, resulting in a poor solution to structural ambiguities.

**Lack of lexical dependencies**

Words do play a role in PCFGs through the probability that non-terminals are expanded to a particular word (e.g. V → *run [0.00003]*).

But lexical information can also be useful to resolve attachment ambiguities! For example, certain verbs prefer certain constructions, even though others are grammatical.

**Determinism**

Since the grammar is context-free, depending on how the probabilities are set, the grammar will always prefer one attachment over the other, even though there may be information in the sentence to disambiguate:

*The guy looked at the man with the telescope.*

*The guy waved at the man with the telescope.*

**How do we address this?**

We can annotate based on lexical information, although this can quickly become infeasible and overfit the training corpus!

There are more advanced solutions, but they are outside the scope of this course: you're encouraged to read up and ask questions if you're interested!

## Partial parsing

**Partial or shallow**

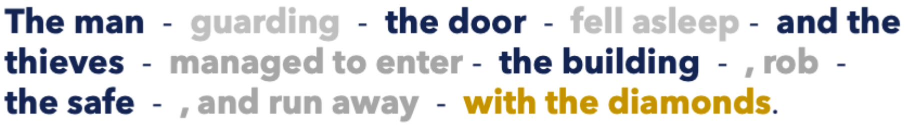
Sometimes we don't need the full parse tree, we just need to know the segments which are akin to constituents at the surface level, without the structure indicating how they combine. Useful for information extraction and information retrieval.

**Chunking**

Is the process of identifying and classifying the flat, nonoverlapping segments of a sentence which constitute the basic non-recursive phrases corresponding to the major content-word PoS tags, i.e. NPs, VPs, APs, and PPs.
There's no hierarchical structure involved, just sequential.

**An example**

The man guarding the door fell asleep and the thieves managed to enter, rob the safe, and run away.



**The man** - guarding - **the door** - fell asleep - **and the thieves** - managed to enter - **the building** - , rob - **the safe** - , and run away - **with the diamonds**.

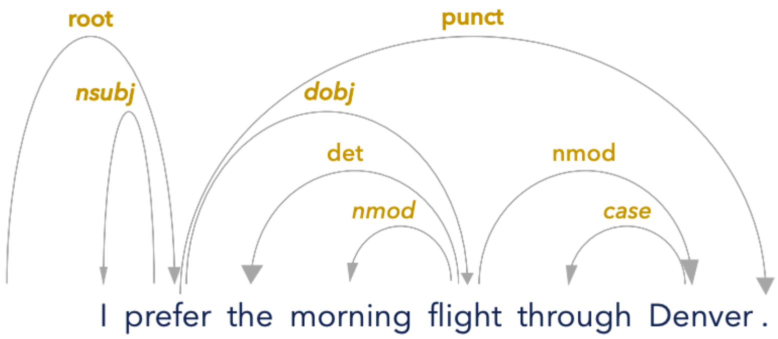
**Non-recursion**

Base phrases of a given type never contains smaller phrases of the same type: chunking boils down to determining the boundaries of the phrases, or chunks.
Typically, phrases include the head with any preceding material (in English) but exclude post-head material to avoid having to resolve attachment ambiguities

**Dependency Parsing**

A different formalism
Syntax is described in terms of tokens (or lemmas) in a sentence and a set of typed directed (from head to dependent) binary grammatical relations among such tokens. No more constituents and phrase structure!

**For example**



**What for?**

Explicitly and directly encode information about relations across words, often buried in constituent based trees!

Deal very well with morphologically complex languages and free word order (not English!) without having ultra-specific rules.
Typed relations approximate semantic relations.

**Roles**
We can represent syntactic roles directly as typed binary relations. These correlate with constituent types and position in the sentence, but the relation is not causal and doesn't hold with free word order languages!
Clausal relations: syntactic roles (subj, dobj, …)
Modifier relations: ways of modifying heads (mod)

**Directed graphs: components**
G=(V,A)
V: a set of vertices (tokens in the vocabulary – can include stems and affixes or punctuation)
A: labeled ordered pairs of vertices, or arcs (the typed binary relations across tokens)

**Rooted trees**
A dependency tree is a directed graph where:

- There is one root node with no incoming arcs
- Every other vertex has exactly one incoming arc
- There is only one path from the root to each vertex in V

Each word has a single head, the structure is connected, there's only one route to each vertex.

**Training data**
Dependency treebanks! Linguists have created corpora with annotated typed binary directed dependency relations that are used to train dependency parsers.
Alternatively, deterministic approaches (head-finding rules) can be applied to constituent-based treebanks to convert them to dependency treebanks.

## Evaluation

**Chunking**
Chunkers are evaluated with precision, recall, and F-measure. A correctly identified chunk has the right span (starts and ends at the right points) and has the right label.
Precision is the ratio between the correct chunks produced by the system and all the chunks produced by the system. Recall weighs the correct chunks over the correct chunks in the text. F-measure combines the two.

**PARSEVAL**
Measure to what extent the constituents in the hypothesized parses look like the actual constituents, defined by linguists (gold standard corpus with annotations required).
Performance is assessed at the constituent level because, alas, parsers will likely make at least one mistake per sentence, if sentences are long or language is non-standard.
Constituent-level evaluation is more fine-grained.

**Correct hypotheses**
A constituent is labeled as correct if there is a constituent in the gold-standard with the exact same *starting point, end point*, and *non-terminal symbol*.
As usual, we rely on recall (correct in hypothesis over correct in gold standard), precision (correct in hypothesis over hypothesized), and F-measure.

**Dependencies**
Labeled and unlabeled attachment accuracy: how many estimated dependencies have the same head and dependent as the gold standard and also the correct type?
Also label score: disregard the head-dependent relation, only check if a word gets the correct typed relation.

## KNOWLEDGE CLIPS 9 – Distributional

Represent meaning with vectors

**Words as symbolic atoms**
So far we've looked at words as strings of letters or indices in a vocabulary. We know edit distance, which only deals with word form.
If we want to compare meaning, we need thesauri. Can we do it with just a corpus?

**Sets to express meanings**
Synonyms are words which can be substituted in some contexts. Hence, two words are more similar when the overlap of their contexts is large. How does this relate to the Lesk algorithm for word sense disambiguation?

**Jaccard's distance**
A measure of set overlap, which ranges between 0 (no overlap) to 1 (complete overlap).
Very flat and coarse: no information about co-occurrence frequency or about context similarity, prone to chance co-occurrences.

$$Jacc(x,y)=\frac{x\cap y}{x\cup y}$$

**From symbols to numbers**
We usually think of words and letters as symbols: each symbol is identical to itself and equally different from all other symbols of the same type.
But when we encode words and letters as numbers (or numerical features) we can exploit a wide range of tools and start to leverage graded relations other than same/different.

**Meaning as use**

> The meaning of a word is its use in the language. – Ludwig Wittgenstein (1953)

**The distributional hypothesis**

> You shall know a word by the company it keeps. – J.R. Firth (1957)

## Words & Vectors

**Connotations**

- Valence: how pleasant the stimulus word is
- Arousal: the intensity of the emotion provoked by the stimulus word
- Dominance: the degree of control exerted by the stimulus word

**N-dimensional spaces**

A word for which we know the valence, arousal, and dominance ratings is akin to a vector in a 3d space, which defines a single point with coordinates [x,y,z].
In this space we can compute finer-grained measures than Jaccard distance between points (i.e. words) and capture relations across words.

**Embeddings**

Continuous, numerical representations of words as points embedded in a particular n-dimensional space (hence the name).
Depending on how the vectors are derived, the same word (the same surface form) can have different embeddings. There is a close relation between the single vector and the vector space.

**Co-occurrence counts**

Vector spaces are typically created by defining a set of target units, a type of context, and incrementing counts of target-context cooccurrences.
Vector spaces are essentially matrices: rows are targets, columns are contexts. Targets and contexts can be virtually anything: vector space models are very flexible!

**Similarity**

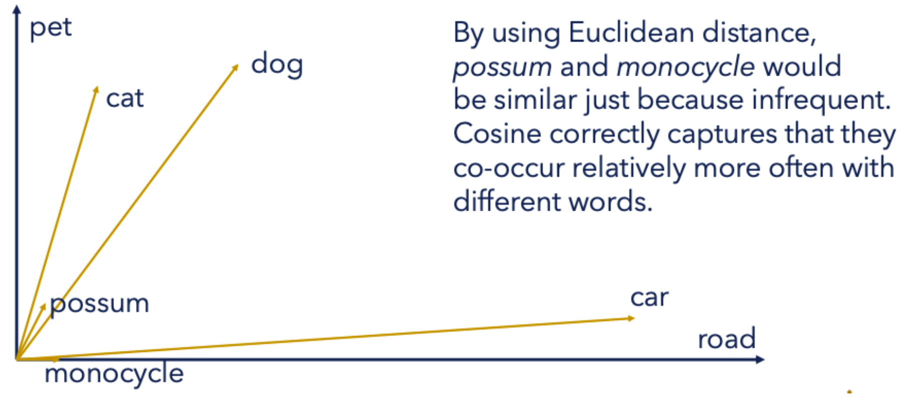
Computed using the dot-product and derived measures, typically cosine (dot product normalized by the L2 vector norm).
Cosine captures the angle between two (hyper)vectors: a cosine of 1 indicates similar words, a cosine of 0 indicates dissimilar words.

$$\cos(\vec{v},\vec{w})=\frac{\vec{v}*\vec{w}}{\text{¿}\vec{v}\vee\text{¿}\vee\vec{w}\vee\text{¿}=\frac{\sum_{i=1}^{N}v_i w_i}{\sqrt{\sum_{i=1}^{N}{v_i}^2}\sqrt{\sum_{i=1}^{N}{w_i}^2}}\text{¿}}$$

**Why not Euclidean distance?**

Usually, we compute distances in n-dimensional spaces using Euclidean distance. However, Euclidean distance is very sensitive to similarity in absolute values.
Cosine gives prominence to similarity in relative values: words are similar if they have similar co-occurrence patterns, rather than similar co-occurrence frequencies.

**Consider this**



By using Euclidean distance, *possum* and *monocycle* would be similar just because infrequent. Cosine correctly captures that they co-occur relatively more often with different words.

**Problems of raw frequency counts**
Even by using a frequency insensitive similarity measure, we are mis-representing word associations because we don’t take into account information provided by context.
Plus, vectors are sparse: they are huge but could be a lot smaller preserving much of their information.

Association weights

**Away to factor information in**
Rather than computing cosine on raw frequency counts, weigh each count by some association measure.
Warp the co-occurrence patterns to weigh the more informative contexts more and the useless ones less (or not at all).

**Pointwise Mutual Information**
Intuition: weigh more the co-occurrences of words that happen more often than expected by chance, since these co-occurrences capture some linguistically relevant phenomenon (like what?)
If language was random, we would have no association weights, but language isn’t, so let’s use this: some co-occurrences carry more information!

**Analytically**

$$PMI(w,c)=\log_2 \frac{p(w,c)}{p(w)\,p(c)} = \text{¿} \log_2 \frac{\frac{count(w,c)}{V}}{\frac{count(w)}{V}\,\frac{count(c)}{V}} \text{¿}$$

Where counts are simple co-occurrence and occurrence counts used to derive ML estimates (V is the number of tokens in the corpus).

**Positive PMI**
Cosine can take values between -1 and 1, where 0 means nothing in common, 1 means same direction and -1 means opposite direction. However, to avoid negative similarity scores (which we can’t really interpret) we force all cells in the vector space to be positive. Hence, if something co-occurs less often than expected by chance, we just say its PMI is 0.

### Biases

If two rare things occur together, they have the highest PPMI, which in general favors low-frequency co-occurrences generated from low-frequency targets and contexts.
Two solutions:

- Including an exponent when computing the context probability such that it increases the probability of rare events
- Local Mutual Information in which PMI is multiplied by the log frequency of the target co-occurrence

### Dimensionality reduction

Reasons to reduce sparsity
Vector spaces typically have 105 dimensions, most of which are 0s. Two targets with 0s on the same dimension don't influence the cosine computation, so no reason to keep them around.
Co-occurrences are harvested using atomic contexts which however can be more or less similar: can we leverage this further piece of information?
Plus, make it easier to use the vectors in a downstream task!

### The simplest approach

Compute the information each context carries (frequency, variance, diversity, …) and only keep the contexts with highest values.
Pros: transparent, quick, the dimensions can be easily interpreted and their properties derived from the corpus.
Cons: doesn't reduce noise and doesn't bring latent meanings to the surface.

### Noise and latent meaning

Noise and brittleness: some co-occurrences may happen by chance or under a very specific, non-conventional use of a word. We may want to give these co-occurrence very low importance.
Discover latent meaning: leverage the fact that some contexts are similar and may represent a broader semantic domain, e.g. cats and dogs co-occur with contexts related to animals, not just with the word *animal*.

### Dimensionality reduction

Project the distributional matrix of dimensionality N*M into a lower-dimensional space N*K where K < M (and K < N) such that the reduced space preserves the maximum amount of variance possible: we only want to get rid of redundant information.
The resulting row vectors are real-valued and dense.

### Dimensionality reduction techniques

- Singular Value Decomposition
- Non-negative Matrix Factorization
- kernel Principal Component Analysis
- Latent Dirichlet Allocation (used in Topic Models)
- …

### Latent Semantic Analysis

One of the most influential papers in distributional semantics: using a vector space model, association measures and dimensionality reduction, Landauer & Dumais [1997] approximated human performance in the TOEFL dataset.
Latent dimensions were shown to encode meanings such as *animal-related, vehicle-related, food-related*, …

**Degrees of freedom**
Choose

- a context (and its size)
- an association measure (or none)
- a dimensionality reduction technique (or none)
- a number of latent dimensions
- a similarity measure

Models that make use of this pipeline falls under the name of Distributional Semantic Models.

DIY
**Notebook**
We'll see how to extract word vectors from some corpus and evaluate them by checking how well they encode human's intuitions about semantic relations.

KNOWLEDGE CLIPS 10 – Word2Vec
Using neural networks to learn word vectors

Logistic regression
**The equation**

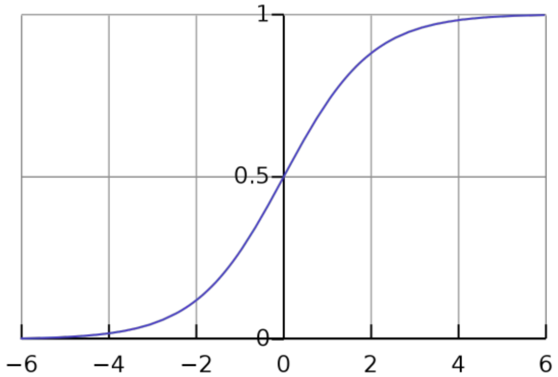
$$z = x * w + b$$

x is the feature representation of one input data point
w is the vector of weights assigned to each feature
b is the bias term (what does it encode?)
z is the weighed sum of the evidence for the class

**Straight to the posterior**
Discriminative classifiers don't make use of likelihood terms but attempt to directly compute the posterior, i.e. p(c|d).
No knowledge of how to generate documents of a class!

**4 components**

- A feature representation: a vector of numeric or symbolic features which encodes each input item
- A classification function: computes the probability of the class given the document
- An objective function: measures how close a model is getting to learning what it's supposed to learn
- An optimizer: updates the model based on the errors in the decision it makes



**Classification function: sigmoid $\sigma$(z)**

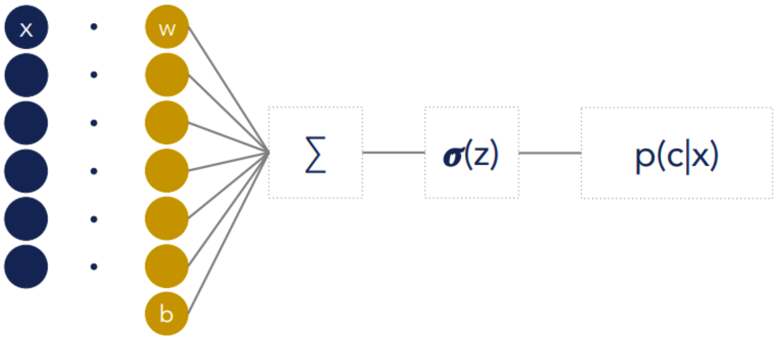
$$y=\sigma(z)=\frac{1}{1+e^{-z}}$$

make sure that p(y=1) and p(y=0) sum to 1!

p(y=1) = $\sigma$(z)

p(Y=0) = 1 - $\sigma$(z)

**A graphical representation**



**Objective Function: Loss**

The function that tells the classifier how good or bad it's doing in classifying training instances: how far is the prediction from the class?

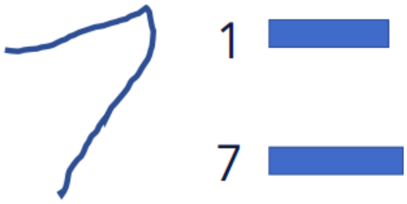
**Loss**

The function that tells the classifier how good or bad it's doing in classifying training instances: how far is the prediction from the class?

Several types of loss functions for continuous data (e.g., MSE) or categorical data (e.g., cross-entropy): choose wisely!

**Probabilistic classifiers**

Rather than providing the best guess as output, a probabilistic classifier indicates the probability of an observation belonging to each possible class.



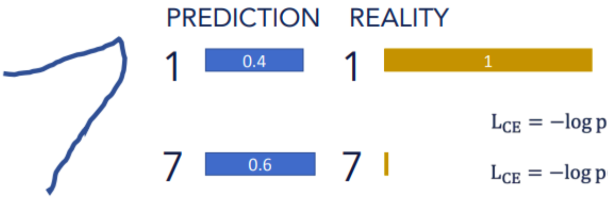
**Why probabilistic distributions?**

Keep options open, maybe what comes next will provide more information in one direction or the other.

However, not always tenable to keep all options: then keep some whenever possible.

**Cross-entropy (or negative log-likelihood)**

Lowest (no error) when $\boldsymbol{\sigma}$(z) = 1 for the correct class. Average loss for each item to obtain the loss for the whole dataset.



$$L_{CE} = -\log p(y|x) = -[y \log \hat{y} + (1-y)\log(1-\hat{y})]$$

$$L_{CE} = -\log p(1|x) = -\log 0.4 = 0.39794$$

**More than one label**

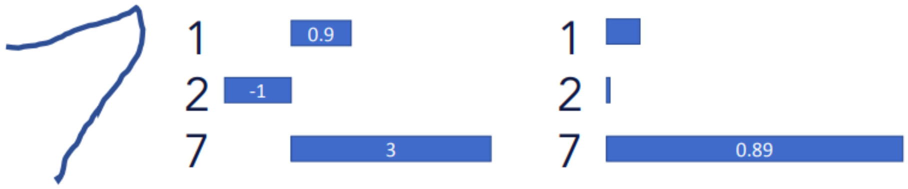
We also have 2s in our world: so three choices!

We estimate a different vector of weights and bias for each class, rather than a single vector.

$$p(y=c|x)=\frac{e^{w_c * x + b_c}}{\sum_{k=1}^{K} e^{w_k * x + b_k}}$$

**Softmax**

$$softmax(z_i)=\frac{e^{z_i}}{\sum_{k=1}^{K} e^{z_k}}$$



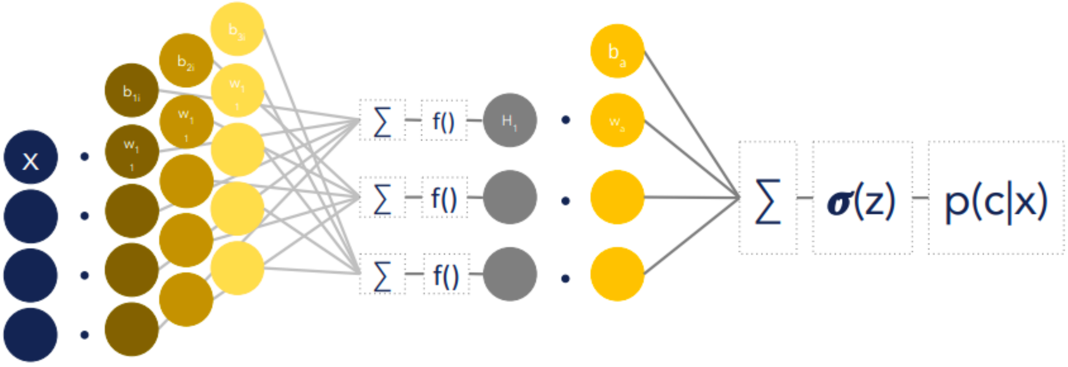
Feed forward- neural nets

**Stacked NNs**

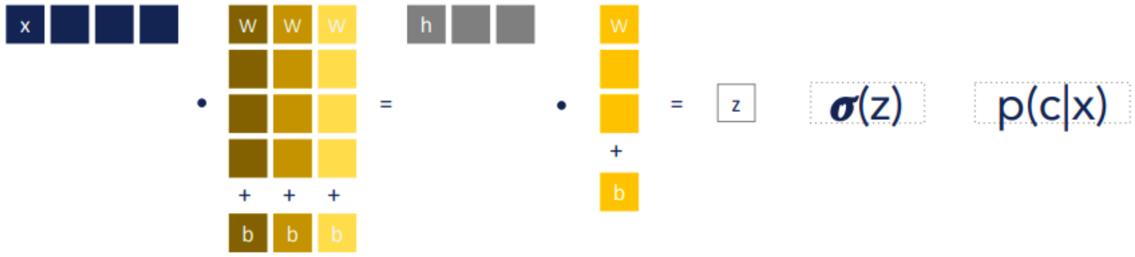
A logistic regression is a linear classifier, so can only learn linear boundaries. What if we combine two, such that the output of the first becomes the input to the second?

We have non-linear classifiers which can learn arbitrary boundaries. We have neural networks.

**A graphical representation**



### A matrix representation



### Multilayer

A feed-forward NN is a multilayer network in which the units are connected directly, such that the output of a lower layer only affects higher layers and is not fed back onto itself or onto lower layers.
Usually called MultiLayer Perceptrons (MLPs) although perceptrons don't use non-linear activation functions, while FFNNs do.

### Hidden layers

Are formed by hidden units, i.e. the processing units which take input values and output one non-linearly transformed value.
*Can be arbitrary many*: even though one is enough to learn any kind of function (given infinite time and memory), stacking hidden layers may help learning *irl*.

### Fully-connected

MLPs are typically fully-connected (or dense): each hidden unit computes the weighted sum over all the units in the previous layer.
There is a direct connection for every pair of hidden units (or neurons) in each pair of adjacent layers (including the input and the output layers)

### Output layer

The output layer is just a standard logistic regression (using the sigmoid for binary and the softmax for multinomial classification).
The input to the output layer is the h n vector (with n=number of hidden layers), i.e. a representation of the input which has been warped to better solve some target problem (usually classification).

### Backpropagation

How do we change all the weights in all hidden layers? We use an algorithm called backpropagation:
First we apply the network to the input (forward pass) and compute the result (the predicted y).
Then we do a backward pass in which we compute all the derivatives we need (we have the partial results we need at intermediate layers).

**Learning rate**

Once we have determined how we need to change the weights, we usually scale these numbers by the learning rate, which essentially models how confident we want to be in updating our hypothesis given the input data we have just experienced.
Can be adapted dynamically (larger at the beginning, smaller later on) using existing algorithms.

**Create a FFNN**

NNs are initialized with small random numbers.
NNs have parameters which are learned (the weights) and hyper-parameters which are set by the modeler (you): how many layers, how many units in each layer, which activation function, which loss, which optimizer, which input representation, the learning rate, … It is important to explore several constellations and check on a dev set how they fare.

## Word2Vec

**NNS for distributional semantics**

We can use neural networks to create word vectors: instead of counting co-occurrences we train a model to predict the neighboring words.
The hidden layer before the output encodes representation of words in a vector space, so they are embeddings!

**Dense from the start**

Rather than building word vectors first collecting sparse vectors and then projecting them on a lower-dimensional space, we could directly build dense vectors.
A very popular technique to do this is Word2Vec.

**Skip-gram with Negative Sampling**

One of the two possible engines of Word2Vec, built around the intuition that you don't need to count co-occurrences but you should predict co-occurring words.
The prediction is not the goal, it's only a way to learn representations. We don't care about the prediction outcome, only about how it influences the embeddings.

**Text supervises itself**

- pick a target word
- isolate a window of co-occurring context words
- update the representation of the target such that it maximizes the probability of predicting its neighbors as neighbors and minimizes the probability of predicting the other words as neighbors.

**Negative sampling**

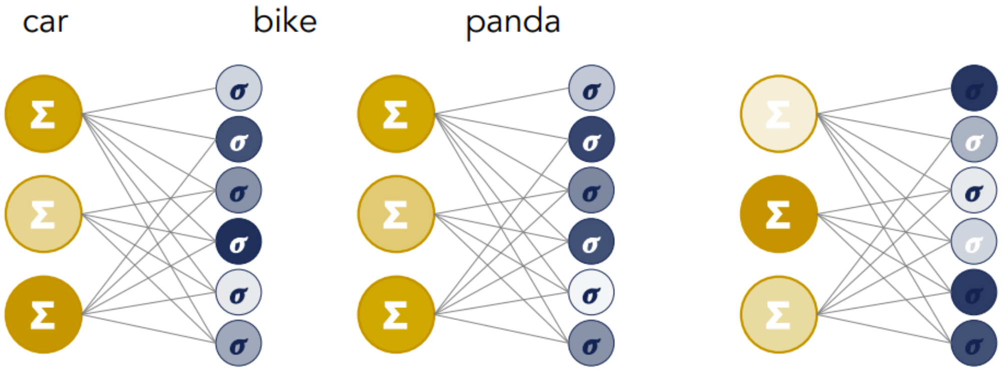
Too many words though! If we compare each target to all the words which don't occur in the window, we're never done. Plus, not all non-cooccurring words are equally informative.
Sample a few words from the vocabulary, with probability inversely proportional to the word frequency and update the NN based on these few words only.

**A simple binary prediction task**
The system doesn't predict which word occurs in the context, but only if a word does: binary vs multinomial classification, much more efficient!

**What's the network guessing?**
The output layer gets the hidden vector corresponding to the input word and applies the softmax to the weighted hidden vector.
The number in each output node is the probability of picking the corresponding word as neighbor of the target.

**All about similarity**
If two different words have very similar "contexts" (that is, more or less the same words are likely to appear around them), then the model has to output very similar predictions for them.
One way for the network to output similar context predictions for these two words is if their hidden layer representations are similar. The network is trained to push vectors closer for words with similar contexts (and foils).

**Embeddings**



**Representation learning**
The algorithm starts with random embeddings and then changes them at every learning instance to be better at predicting the true contexts as neighbors and the foils as foils.
The idea is that better representations (i.e. vectors) support a better prediction, hence by optimizing for prediction we learn better representations.

**Two embeddings pre word**
Each word can be a target or a context (true or foil): a different embedding is estimated for word-as-target and word-as-context.
Embeddings are the weights in a NN hidden layer, that are used to warp the input to minimize some loss function: the final prediction doesn't matter, it's a training device to learn representations.

**Continuous Bag-Of-Word**
The other engine of Word2Vec: flips the problem as compared to the skip-gram.

Predict a target from context words, discarding order information and averaging input vectors. Computationally more efficient.

**CBoW v. SGNS**

*The cate ate the mouse*

CBoW: predict 'ate' from [average('The', 'cat', 'the', 'mouse')],
SGNS: [predict ‘The' from ‘ate'], [predict ‘cat' from ‘ate'], [predict ‘the' from ‘ate'], [predict ‘mouse' from ‘ate'].

**Rare words**

CBoW maximizes the probability of the target word given the context. Hence, given the context yesterday was a really [...] day, CBoW will get rewarded more often for predicting beautiful or nice, than for saying delightful.
SGNS predicts the context from the target. Given the word delightful, it assigns a large probability to the context words yesterday, was, really, day. Independently.

**Competition**

In CBoW, delightful competes with beautiful (and loses). In SGNS, delightful + context pairs are treated as new observations, avoiding the competition which penalizes rare words.
Choose SGNS or CBoW depending on your task and constraints! Which looks more in line with what people do when processing language?

## Evaluation & Bias

**Visualizations**

Project the n-dimensional embeddings in a 2 (or 3) dimensional space and plot.
Use PCA or tSNE (in combination with PCA for large spaces) to project embeddings on a plane and see whether things that should cluster together actually do.

**Semantic drift**

If we have a diachronic corpus, we can estimate different embedding spaces for different time periods and look at which words have changed in meaning and track this change.
E.g. in Italian, the word popolare has recently shifted from ‘of the people’ to ‘famous’ following English influence (popular).

**Intrinsic evaluation**

Assess the goodness of a semantic space by how well it can capture human intuitions about world relations:

- Word similarity and relatedness: WordSim353, SimLex999, TOEFL
- Context-dependent word similarity: SCWS
- Sentence similarity (useful for what?)
- Analogy: $\overrightarrow{Paris} - \overrightarrow{France} + \overrightarrow{Italy} \approx \overrightarrow{Rome}$

**Extrinsic evaluation**

Learn embeddings to improve other tasks (all tasks, actually): if learning a vector space improves the target metric (accuracy, perplexity, or whatever), then it’s useful.
Today, modules for learning word embeddings are part of almost every architecture.

**Biases**

Your model is only good and trustworthy as your data. There's no magic: garbage in, garbage out.

Choose your data carefully to be representative of your problem, but don't stop there: check them!

**Real examples**

Even in carefully chosen corpora, biases are everywhere. The following have been reported:

Gender stereotypes: father: doctor = mother: ? → nurse

Connotations: black names closer to negative words, Caucasian names closer to positive words (think what could happen in a sentiment classifier).

What do we do about it?

**Summary**

We use neural networks to learn better word vectors by optimizing hidden layers to yield better prediction in a binary classification task. Given a source and a target word, predict whether the target is a co-occurring word or a foil.

After learning, the hidden layer for each word encodes its semantic representation as learned from text.

## GUEST LECTURE (Notes)

Inside the Black Box of natural language processing.

## KNOWLEDGE CLIPS 10 – Current trends

Recurring networks to do pretty much anything

### LM with FFNN

**A multimodal classification problem**

Predict the next word out of a finite set of options (however large), hence classify the next word as one out of many.

What is the feature representation we use as input to such a model?

**The good ol' Markov assumption**

Even some neural language models approximate the probability of a word given a history by considering local histories.

Hence, the input is the n-gram preceding the word we need to classify.

**Advantages of FFNNs**

- No need to apply smoothing
- Can handle much longer histories (n-grams with very large n)
- Generalize better over histories consisting of similar but not identical words

**Embeddings as an antidote to sparsity**

We know them already: how can we use them here?

Suppose we need to predict the next word given the n-gram *the silver pike* … but we never saw this n-gram in our corpus. However, we have seen each word, and we have derived a continuous representation for them: *pike* looks a lot like *fish*, so probably it *swims*.

**Pre-training or not**

We can get off-the-shelves pre-trained embeddings and fine-tune them during the training of the language model or train them from scratch.

The important thing is: we are changing the representation of the input so that we can leverage syntactic and semantic relations to help us in times of need (when we see something new at test).

**Are we doing everything we can?**

The next word is influenced by the local history (Markov assumption). We can increase n, but not too much: the more we increase it, the larger the corpus we need, even using embeddings.

Can we drop it, such that we are considering the whole history, maybe weighing contributions from far words less?

## RNNs

**Avoid the sliding window**

In order to look at the preceding n-gram, we process each sentence by 'moving' a window such that we use the first n-1 units to predict the $n^{th}$ .

However, this limits the context arbitrarily: sometimes we care about short windows, sometimes about longer ones in the same model.

**Recurrence**

Rather than using the previous n-gram as the input to predict the next event, allow the previous state of the model to influence the current one.

Thanks to recurrence, the value of a hidden unit doesn't simply reflect the weighted sum of the input but also reflects the previous hidden state.

**One at a time**

Rather than parsing the input into n-grams, we feed the network one word at a time: previous words will influence the current prediction through recurrence.

If the current prediction is influenced by the previous hidden state, the chain of recurrence goes back to the beginning (almost).

**A new set of weights**

A recurrent network consists of three sets of weights:

- those connecting the input to the hidden layer (W)
- those between the previous hidden state and the current hidden states (U)
- those between the current hidden state and the output (V)

All these weights are trained in the usual way (loss + gradient descent + backpropagation)

**With or without embeddings**
RNNs can work with word embeddings as well (they usually do these days).
We add a fourth set of weights E connecting the input to the embedding layer, which are then connected to the hidden layer through the set of weights W.

**Backpropagation through time**
In the forward pass, we go through the whole sequence and keep track of the loss.
Only at the end of the input sequence we start the backward pass and go all the way processing the sequence in reverse: at every step we compute the required error terms gradients and save the error to compute the gradient at the next (i.e. previous) time step.

**Stacked**
We can have as many recurrent hidden layers as we want (but training becomes slow).
In this case we use the entire sequence of outputs from one RNN as the input to the next.
NNs are mostly self-contained modules which we can recombine: but mind how you do it and why. Just because you can doesn't mean you have to!

**Successive abstractions**
Stacking layers can be useful when the input is noisy and we want the network to learn abstract feature bundles from the input.
Each layer computes a slightly more abstract representation which is passed up (~ vision or sound in the brain).

**Biodirectory**
We usually conceive of RNNs as moving left to right. But there may be useful information in the reverse too.
Nothing prevents us from having two RNNs which read the input sequences in both directions combining their higher hidden layers to encode a single representation of the sequence.
RNNs that do this are called bidirectional: representations are combined through concatenation or elementwise sum/multiplication

**Vanishing gradient**
If the sequence (sentence) is long, gradients in backpropagation through time are multiplied several times and may end up being zero, so exerting no change in weights.
LSTMs and GRUs modify the recurrent layers to handle this by forgetting some information, which won't be carried on to later input steps (and hence doesn't influence gradient computation).

*"The de facto consensus in NLP in 2017 is that no matter what the task, you throw a BiLSTM at it, with attention if you need information flow."*
Christopher Manning, Stanford University

## Applications of RNNs

**Another arrow for language modeling**
We know how to build statistical language models using Markov Chains, and neural language models using FFNN with sliding windows. We can also do it with RNNs.
Usually better but slower and harder to train.

**No Markov, no smoothing**

Using RNNs for language modeling means we can:

- avoid the Markov assumption: the hidden layer encodes information from all states (more about the most recent ones)
- avoid smoothing: the embeddings project words into a continuous space where we can leverage similarity relations to guess how our OOV word could behave.

**Predict the next word**

Prediction depends on two things:

- The embedding of the current input word
- The hidden layer at the previous time step

This information is summed to obtain a new hidden layer which is fed through the weights V to get a value which is transformed using the softmax.

**The whole sequence**

The softmax at each time step encodes the probability of each word as a continuation given the current network state (input embedding + previous hidden state).
The product of the probabilities of each <u>correct continuation</u> under the softmax is the probability of the whole sequence under the trained language model.

**Autoregressive**

RNNs language models can also generate language:

- Take the BOS, run the trained network and get the word with the highest softmax probability.
- Take the embedding of the generated word, run the network on it, and again take the word with highest probability in the softmax.
- Repeat until you generate the EOS symbol.

**Label sequences**

We know HMMs to do this, how can we use NNs?
With FFNN it's very similar to language models except for two key differences:

- the softmax is not computed over the vocabulary but over the set of labels.
- the loss is computed considering the correct label, not the next word (not autoregressive).

**Same old, same old**

A recurrent neural sequence tagger still takes word embeddings and is trained to minimize the error on the target classes by updating its weights through back-propagation.
NNs are highly modular!

**Applications of RNN sequence taggers**

<u>PoS tagging</u> (but not much better than HMMs or MEMMs: why?)
Named Entity Recognition and Labeling: find and appropriately label words which denote entities (countries, people, organizations, …)
Structure prediction: given an input, produce the correct set of action to achieve the desired output.

**Extra 1: sequence classification**
We can use RNNs to classify whole sequences (these models are called Seq2Label).
We run the model through a whole sequence: at the end, the hidden layer will encode a representation of the whole sequence. We can then put a simple logistic regression on top to learn to correctly classify this representation. The loss now refers to the class of the whole sequence. This is a discriminative model!

**Extra 2: sequence to sequence**
Rather than emitting a label at the final step, predicting a label for the current word or the next word altogether, we may want to go from a sequence to a different sequence, without a necessary one-to-one mapping between units in both sequences. Models to do this are called Seq2Seq.

**Extra 2: encoding-decoding**
Seq2Seq models consist of two connected sub-networks. The first encodes the source sequence (input) in a hidden state.
The second decodes the representation from the hidden state into the target sequence (output). Training happens by jointly updating the weights of the encoder and the decoder in such a way that the decoded output resembles the target output as closely as possible.

**Extra 3: fight sparseness**
We usually think of languages in terms of words, but that's problematic.
Dispense with the notion of word altogether and use characters: RNNs are well suited to this because they don't need the Markov assumption.
You can also learn character embeddings (what could they capture? How could they differ when learned on text or speech?)

**Transformers**
Persisting issues RNNs

- RNNs seem to be useful for any application in Natural Language Processing

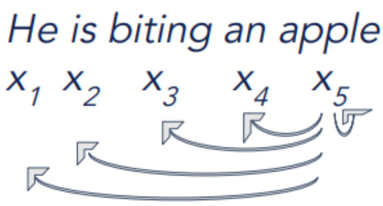
However:

- the vanishing problem remains!
    - LSTM & GRU architectures mitigate this issue
    - but information is still lost through recurrence
    - problem for long distance dependencies

**Transformers to the rescue**

- Can we get rid of recurrence while still dropping the Markov assumption?
- Self-attention layer:
    - maps an input sequence ($x_1$ , $x_2$ , …, $x_n$ ) to an output sequence ($y_1$ , $y_2$ , … $y_n$ )
    - extracts information from context of any length
- How is this done?

**Self-Attention**
The self-attention layer compares an element (word) to a collection of other elements (words) of the context (including the word itself)



$x_5$ is compared to $x_1$ , $x_2$ , $x_3$ , $x_4$ and $x_5$

Each comparison gives us a score ($\boldsymbol{\alpha}_{1,5}$, $\boldsymbol{\alpha}_{2,5}$, ... , $\boldsymbol{\alpha}_{5,5}$) for each word pair

- $\boldsymbol{\alpha}_{1,5}$ = score($x_1$ ,$x_5$ )

The scores tell us the relevance of each compared word

- Normally computed with (scaled) dot product

Each output is a weighted sum of the (normalized) scores:

- $y_5$ = sum($\boldsymbol{\alpha}_{1,5}x_1$ , $\boldsymbol{\alpha}_{2,5}x_2$ , ... , $\boldsymbol{\alpha}_{5,5}x_5$ )

Each word can have three roles:

- as a query (q), when it is compared to other words in the context (in our example, $x_5$ plays this role)
- as a key (k), when it is part of the context (in our example, $x_1$ to $x_5$ act as keys)
  - so the scores can be written as $\boldsymbol{\alpha}_{i,j}$ = score($k_i$ ,$q_j$ )
- as a value (v) when used to compute the output:
  - $y_5$ = sum($\boldsymbol{\alpha}_{1,5}v_1$ , $\boldsymbol{\alpha}_{2,5}v_2$ , ... , $\boldsymbol{\alpha}_{5,5}v_5$ )

We learn separate weights for each role!

- Each input $x_i$ is transformed into its role by multiplying the embedding with the corresponding weights:
  - $q_i=W^Qx_i$
  - $k_i=W^Kx_i$
  - $v_i=W^Vx_i$

We have lots of weights to learn!

But this is still more efficient than recurrence:

- we get rid of vanishing gradient problem
- independent computations for each score allow for parallelization

**Multi-head Self-attention**

We can learn even more weights!

- The same pair of words in a sentence can be related in a different ways depending on whether we focus on syntax, semantics, discourse, etc.
  - Can we have specialized attention for different relations between words?

Yes! We "just" need to learn separate weights for each of these

  - the model size increases
  - but we can still learn these weights in parallel!
- Each set of self-attention layers is called a head

**Positional Embeddings**

What happens to the Markov assumption in Transformers?

- We can model sequences of any length
  - So we don't need the Markov assumption
- But… are these really still sequences?
  - By dropping recurrence, we have lost information of word order!

How can we incorporate a notion of word order while maintaining independent computations between words?

- We use a function that transforms the position of the word (1,2, 3, …, n) to a continuous value that preserves the ordinal information
- Often this is done with sine or cosine functions

**The transformer revolution**

With these ingredients, we can build very powerful models!

Some of the most popular:

- BERT (Bidirectional Encoder Representations from Transformers), by Google
  - and variants like RoBERTa, XLNet, DistilBERT
- GPT (Generative Pre-trained Transformer ), by OpenAI
  - GPT-1 to GPT-3

Transformer-based language models currently receive great attention due to their outstanding performance in many tasks

- sparking debates about what these models can and cannot do, how close they are to achieve human-like linguistic abilities and even general Artificial Intelligence

**Implications for Computational Linguistics**

Open questions:

- How much do Transformers really learn about Natural Language?
  - e.g. Do Transformers learn the same syntactic relations defined in linguistics?
- Do Transformers emulate human linguistic abilities?
  - e.g. Do Transformers process sentences in a similar way?

Appendix

**Implications for Computational Linguistics**

Unlike humans, these models learn from HUGE amount of data

- More data → Better performance
  - BERT: 16 GB (3.3B tokens)
  - RoBERTa: 10 times more than BERT (for 2-20% improvement)
  - GPT-2: 40GB (8B tokens)
  - GPT-3: over 12 times more than GPT-2

What does this tell us about nature vs. nurture of the human linguistic capacity?