Natural Language Processing

Matteo Donati

September 29, 2022

Contents

1 Introduction

Natural Language Processing (NLP) refers to the trend of automating the analysis, generation, and acquisition of human (i.e. natural), language. In particular:

- Analysis (or "understanding", or "processing"): the input to analysis is language, while the output is some representation that supports useful action.
- Generation: the input to generation is the aforementioned representation, while the output is language.
- Acquisition: is about obtaining the representation and the necessary algorithms from knowledge and data.

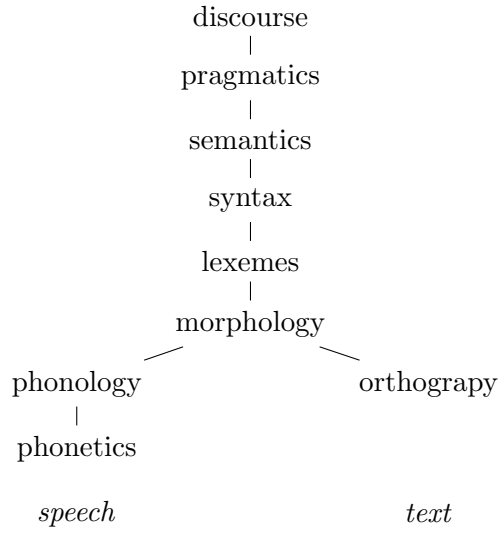
An NLP model is an abstract, theoretical, predictive construct which includes:

- A (partial) representation of the world.
- A method for creating or recognizing worlds.
- A system for reasoning about worlds.

NLP is related with many disciplines, such as computational linguistics, information retrieval, text mining and big data analytics.

1.1 Levels of Representation

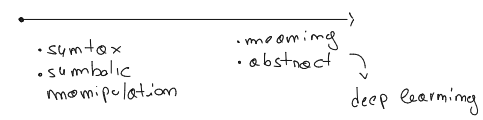
The levels of linguistic representation are the following:



In general, the mappings between levels are extremely complex, and it is difficult to obtain training data for each aspect. Moreover, each string may have many possible interpretations at every level, and the correct resolution of the ambiguity depends on the intended meaning, which is often inferable from the context. In particular:

- **Morphology** is the analysis of words into meaningful components.
- **Lexemen** is about lexical analysis, namely about normalizing and disambiguating words.
- **Syntax** is about transforming a sequence of symbols into a hierarchical or compositional structure. Syntax is closely related to linguistic theories about what makes some sentences well-formed and others not. In general, the number of ambiguities explode combinatorially.
- **Semantics** is about the mapping of natural language sentences into domain representations.
- **Pragmatics** is about any *non-local* meaning phenomena (e.g. "can you pass the salt?" or "Are you 18?", "Yes, I'm 25").
- **Discourse** is about texts, dialogues, multi-party conversations.

NLP:



2 Basic Text Processing

Examples of basic text processing are: regular expressions and text normalization.

2.1 Regular Expressions — formal language for specifying text strings

Regular expressions are used to describe text patterns, to extract information from the text, and to convert text into a convenient/standard form. In particular:

- **Disjunctions** can be of the form:

RE	Match	Example Patterns
`/[wW]oodchuck/`	Woodchuck or woodchuck	"Woodchuck"
`/[abc]/`	'a', 'b', *or* 'c'	"In uomini, in soldati"
`/[1234567890]/`	any digit	"plenty of 7 to 5"

Figure 2.2 The use of the brackets `[]` to specify a disjunction of characters.

disjunction for single characters

RE	Match	Example Patterns Matched
`/[A-Z]/`	an upper case letter	"we should call it 'Drenched Blossoms' "
`/[a-z]/`	a lower case letter	"my beans were impatient to be hoed!"
`/[0-9]/`	a single digit	"Chapter 1: Down the Rabbit Hole"

Figure 2.3 The use of the brackets `[]` plus the dash - to specify a range.

RE	Match (single characters)	Example Patterns Matched
`/[^A-Z]/`	not an upper case letter	"Oyfn pripetchik"
`/[^Ss]/`	neither 'S' nor 's'	"I have no exquisite reason for't"
`/[^.]/`	not a period	"our resident Djinn"
`/[e^]/`	either 'e' or '^'	"look up ^ now"
`/a^b/`	the pattern 'a^b'	"look up a^ b now"

Figure 2.4 The caret ^ for negation or just to mean ^. See below re: the backslash for escaping the period.

disjunction for words

Alternative words can added using the `|` pipe (e.g. `a|b|c`).

e.g. [wW]oodchucks?

- **Optional elements and wildcards** can be of the form: `?` for optional previous character, `*` for zero or more instances of the previous character (Kleene *), `+` for one or more instances of the previous character (Kleene +), `.` for any character.

e.g /beg.m/ -> anything between "beg" and "m"

- **Anchors** can be of the form: `^` for the start of a line (e.g. `/^The/`), `$` for the end of a line (e.g. `/^The dog\.$/`), `b` for word boundary (e.g. `/\bthe\b/`), `B` for word non-boundary (e.g. `/\Bthe\B/`).

dot escape, otherwise "." means something else

- More regular expressions operators can be common sets of characters:

RE	Expansion	Match	First Matches
\d	[0-9]	any digit	Party␣of␣5
\D	[^0-9]	any non-digit	Blue␣moon
\w	[a-zA-Z0-9_]	any alphanumeric/underscore	Daiyu
\W	[^\w]	a non-alphanumeric	!!!!
\s	[␣\r\t\n\f]	whitespace (space, tab)	
\S	[^\s]	Non-whitespace	in␣Concord

Figure 2.7 Aliases for common sets of characters.

counting operators:

RE	Match
*	zero or more occurrences of the previous char or expression
+	one or more occurrences of the previous char or expression
?	exactly zero or one occurrence of the previous char or expression
{n}	*n* occurrences of the previous char or expression
{n,m}	from *n* to *m* occurrences of the previous char or expression
{n,}	at least *n* occurrences of the previous char or expression
{,m}	up to *m* occurrences of the previous char or expression

Figure 2.8 Regular expression operators for counting.

escape symbols:

RE	Match	First Patterns Matched
*	an asterisk "*"	"K*A*P*L*A*N"
\.	a period "."	"Dr. Livingston, I presume"
\?	a question mark	"Why don't they come and lend a hand?"
\n	a newline	
\t	a tab	

Figure 2.9 Some characters that need to be backslashed.

Moreover, regular expressions are used in substitutions. In particular, substitutions make use of capture groups to refer to specific sub-parts of the string matching a pattern. For example:

```
/the (.*)er they were, the \1er they will be/
/the (.*)er they (.*), the \1er we \2/
```

capture groups

2.2 Text Normalization

Every NLP task needs to do text normalization: segmenting/tokenizing words in running text, normalizing word formats, segmenting sentences in running text. To define some concepts, different examples are considered:

- *I do uh main- mainly business data processing.* In this sentence there are two kinds of **disfluencies**: fragments and filled pauses.
- *Seuss's cat in the hat is different from other cats!. cat* and *cats* are the same **lemma**, but they have different **wordforms**.

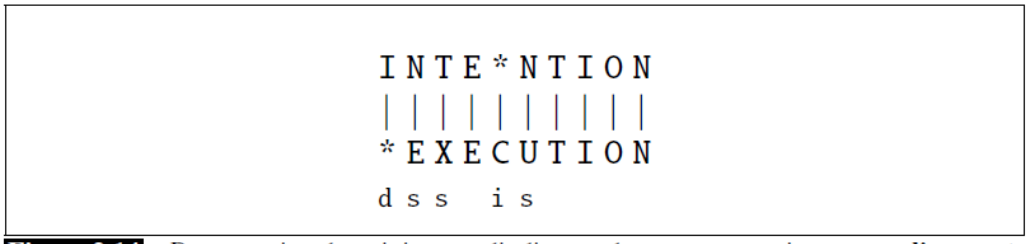
Moreover:

- A **type** is an element of a vocabulary, while a **token** is an instance of a type. In general, V is the vocabulary and N the number of tokens. Herdan's law states that $|V| = kN^{\beta}$. Moreover, **tokenization** could have issues with some specific type of words or part of the text (e.g. contractions, etc.), and with some languages.
- **Word normalization** is the task of putting words/tokens in a standard form (e.g. how to match U.S.A. and USA).
- **Case folding** (e.g. mapping everything to lowercase) is an example of normalization.
- **Lemmatization** reduces inflections or variant forms to their base form (e.g. *am*, *are*, *is* become *be*). An accurate lemmatization requires complete **morphological parsing**.
- **Stemming** is a naive version of morphological analysis which reduces terms to their stems (i.e. the core meaning-bearing units). This implies a crude chopping of affixes (i.e. bits and pieces that adhere to stems). For example, *automate*(s), *automatic*, *automation* become *automat*.
- **Sentence segmentation** is the process of segmenting sentences in the running text. Questions marks and exclamation points are relatively unambiguous, while periods are more ambiguous.

3 Edit Distance, Spelling Correction and the Noisy Channel

3.1 Edit Distance

The concept of edit distance defines how similar are two strings. In particular, the **minimum edit distance** represents the minimum number of edit operations needed to transform one string into another. These edit operations can be: insertions, deletions, substitutions. For example:



alignment problem (search problem)

Figure 2.14 Representing the minimum edit distance between two strings as an **alignment**. The final row gives the operation list for converting the top string into the bottom string: d for deletion, s for substitution, i for insertion.

A cost is assigned to each operation. For instance: insertions costs one, deletion costs one, while substitution costs two (but substituting a letter for itself costs zero).
Finding the edit distance can be viewed as a search problem. In particular, one could be interested in finding the shortest path from one string to another. The search space is huge but many paths lead to the same destination. One could exploit dynamic programming so to re-use always visited portions of this space.

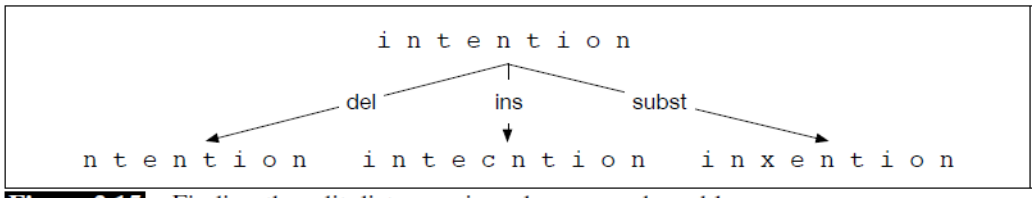


Figure 2.15 Finding the edit distance viewed as a search problem

The algorithm for computing the minimum edit distance is illustrated below:

1. Computation of the minimum edit distance between *intention* and *execution* using Levenshtein distance with cost of one for insertions or deletions, two for substitutions.

2. When entering a value in each cell, one marks which of the three neighbouring cells one came from with up to three arrows. After the table is full one computes an alignment (minimum edit path) by using a backtrace, starting (in this case) at the eight in the lower-right corner and following the arrows back. The sequence of bold cells represents one possible minimum cost alignment between the two strings.

Src\Tar	#	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

	#	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**

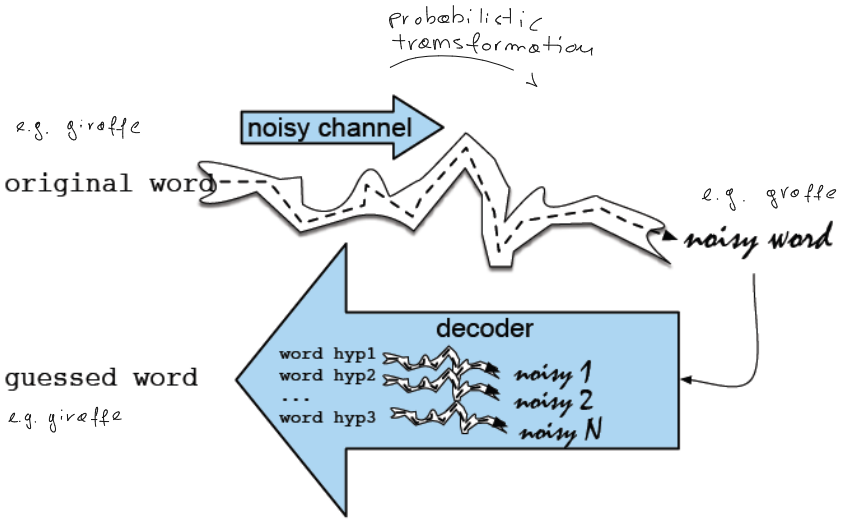
3.2 Spelling Correction and the Noisy Channel

There exist two perspectives to spelling correction:

- **Non-word spelling correction**, for example *graffe* for *giraffe*. Non-word spelling correction can be detected using a dictionary. Then, a ranked list of candidates can be generated, and the ranking is based on a distance metric such as the minimum edit distance. The preference is given to frequent words.

- **Real word spelling correction**, which derives from typographical errors (insertions, deletions, transpositions), or cognitive errors (e.g. *dessert* for *desert*). In real word spelling error detection, any input word could be an error.

In the **noisy channel** model, one can image that the surface form one sees is actually a distorted form of an original word passed through a noisy channel. The decoder passes each hypothesis through a model of this channel and picks the word that best matches the surface noisy word:



In this context, the goal is to build a model of the channel. One can pass every word of the language through the channel, so to see which one comes out closest to the observed word. One can see this process as a kind of Bayesian inference: out of all the words in the vocabulary, one tries to find the word w such that $P(w|x)$ is the highest (w is the original word, x is the noisy word):

$$\hat{w} = \underset{w \in V}{\operatorname{argmax}} P(w|x) \tag{1}$$

By Bayes rule:

$$P(w|x) = \frac{P(x|w)P(w)}{P(x)} \tag{2}$$

Because $P(x)$ is independent of w, one has:

$$\hat{w} = \underset{w \in V}{\operatorname{argmax}} P(w|x) = \underset{w \in V}{\operatorname{argmax}} P(x|w)P(w) \tag{3}$$

Finally, one can restrict the analysis to a set of candidates $C \subset V$:

$$\hat{w} = \underset{w \in C}{\operatorname{argmax}} \underbrace{P(x|w)}_{\text{channel model}} \underbrace{P(w)}_{\text{prior}} \tag{4}$$

The prior probability of a word is usually defined by the contest c ($c \rightarrow w \rightarrow x$), so that:

$$\hat{w} = \underset{w \in C}{\operatorname{argmax}} \underbrace{P(x|w)}_{\text{channel model}} \underbrace{P(w|c)}_{\text{language model}} \tag{5}$$

3.2.1 The Noisy Channel Spelling Method

This method can be executed as follows:

1. Find words with similar spelling to the input word. Usually, the majority of error are single-letter changes. To do so, one can use the Damerau-Levenshtein edit distance (includes four edits: insert, delete, substitute, transpose adjacent). Then, a list of candidates at distance one is computed.

2. Score each candidate based on the language model and the channel model. To do so, one could use an $|A| \times |A|$ confusion matrix:

sub[X, Y] = Substitution of X (incorrect) for Y (correct)

X	Y (correct)																									
	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o	p	q	r	s	t	u	v	w	x	y	z
a	0	0	7	1	342	0	0	2	118	0	1	0	0	3	76	0	0	1	35	9	9	0	1	0	5	0
b	0	0	9	9	2	2	3	1	0	0	0	5	11	5	0	10	0	0	2	1	0	0	8	0	0	0
c	6	5	0	16	0	9	5	0	0	0	1	0	7	9	1	10	2	5	39	40	1	3	7	1	1	0
d	1	10	13	0	12	0	5	5	0	0	2	3	7	3	0	1	0	43	30	22	0	0	4	0	2	0
e	388	0	3	11	0	2	2	0	89	0	0	3	0	5	93	0	0	14	12	6	15	0	1	0	18	0
f	0	15	0	3	1	0	5	2	0	0	0	3	4	1	0	0	0	6	4	12	0	0	2	0	0	0
g	4	1	11	11	9	2	0	0	0	1	1	3	0	0	2	1	3	5	13	21	0	0	1	0	3	0
h	1	8	0	3	0	0	0	0	0	0	2	0	12	14	2	3	0	3	1	11	0	0	2	0	0	0
i	103	0	0	0	146	0	1	0	0	0	0	6	0	0	49	0	0	0	2	1	47	0	2	1	15	0
j	0	1	1	9	0	0	1	0	0	0	0	2	1	0	0	0	0	0	5	0	0	0	0	0	0	0
k	1	2	8	4	1	1	2	5	0	0	0	0	5	0	2	0	0	0	6	0	0	0	4	0	0	3
l	2	10	1	4	0	4	5	6	13	0	1	0	0	14	2	5	0	11	10	2	0	0	0	0	0	0
m	1	3	7	8	0	2	0	6	0	0	4	4	0	180	0	6	0	0	9	15	13	3	2	2	3	0
n	2	7	6	5	3	0	1	19	1	0	4	35	78	0	0	7	0	28	5	7	0	0	1	2	0	2
o	91	1	1	3	116	0	0	0	25	0	2	0	0	0	0	14	0	2	4	14	39	0	0	0	18	0
p	0	11	1	2	0	6	5	0	2	9	0	2	7	6	15	0	0	1	3	6	0	4	1	0	0	0
q	0	0	1	0	0	0	27	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
r	0	14	0	30	12	2	2	8	2	0	5	8	4	20	1	14	0	0	12	22	4	0	0	1	0	0
s	11	8	27	33	35	4	0	1	0	1	0	27	0	6	1	7	0	14	0	15	0	0	5	3	20	1
t	3	4	9	42	7	5	19	5	0	1	0	14	9	5	5	6	0	11	37	0	0	2	19	0	7	6
u	20	0	0	0	44	0	0	0	64	0	0	0	0	2	43	0	0	4	0	0	0	0	2	0	8	0
v	0	0	7	0	0	3	0	0	0	0	0	1	0	0	1	0	0	0	8	3	0	0	0	0	0	0
w	2	2	1	0	1	0	0	2	0	0	1	0	0	0	0	7	0	6	3	3	1	0	0	0	0	0
x	0	0	0	2	0	0	0	0	0	0	0	0	0	0	0	0	0	0	9	0	0	0	0	0	0	0
y	0	0	2	0	15	0	1	7	15	0	0	0	2	0	6	1	0	7	36	8	5	0	0	1	0	0
z	0	0	0	7	0	0	0	0	0	0	0	7	5	0	0	0	0	2	21	3	0	0	0	0	3	0

Considering the sentence *"was called a stellar and versatile acress whose combination of sass and glamour has defined her"*, one can define the following table containing the candidate corrections for the misspelling *acress* and the transformations that would have produced the error:

		Transformation			
Error	Correction	Correct Letter	Error Letter	Position (Letter #)	Type
acress	actress	t	—	2	deletion
acress	cress	—	a	0	insertion
acress	caress	ca	ac	0	transposition
acress	access	c	r	2	substitution
acress	across	o	e	3	substitution
acress	acres	—	s	5	insertion
acress	acres	—	s	4	insertion

Supposing priors are computed without taking the context into account, so that $P(w) = \text{count}(w)/|N|$, where N is the number of tokens in the corpus, then (considering COCA, which is composed of 404,253,213 words):

channel model (error model) — prior

Candidate Correction	Correct Letter	Error Letter	x|w	P(x|w)	P(w)	10^9*P(x|w)P(w)
actress	t	-	c|ct	.000117	.0000231	2.7
cress	-	a	a|#	.00000144	.000000544	0.00078
caress	ca	ac	ac|ca	.00000164	.00000170	0.0028
access	c	r	r|c	.000000209	.0000916	0.019
across	o	e	e|o	.0000093	.000299	2.8
acres	-	s	es|e	.0000321	.0000318	1.0
acres	-	s	ss|s	.0000342	.0000318	1.0

Figure B.5 Computation of the ranking for each candidate correction, using the language model shown earlier and the error model from Fig. B.4. The final score is multiplied by 10^9 for readability.

One can notice how *actress* is only the second most likely word, and not the first one. Instead, if one also considers the context c so that $P(w_i) = P(w_i|w_{i-1})$:

- $P(actress|versatile) = .000021$
- $P(across|versatile) = .000021$
- $P(whose|actress) = .001$
- $P(whose|across) = .000006$

Then:

- $P(versatile\ actress\ whose) = .000021 * .001 = 210 * 10^{-10}$
- $P(versatile\ across\ whose) = .000021 * .000006 = 1 * 10^{-10}$

Lastly:

- $P(versatile\ actress\ whose|versatile\ acress\ whose) = 2.7 * 210 * 10^{-19}$
- $P(versatile\ across\ whose|versatile\ acress\ whose) = 2.8 * 10^{-19}$

× When considering real word spelling errors, between 25% and 40% of such speeling errors are valid English words. However, one can still use the noisy channel method by considering the following:

1. Given an input sentence $X = \{x_1, x_2, \ldots, x_k, \ldots, x_n\}$, generate a set of candidate correction sentences $C(X)$.

2. Pick the sentence with the highest language model probability. Here, the channel model accounts for the possibility that a word in not an error, for example:

$$P(x|w) = \begin{cases} \alpha & \text{if } x = w \\ \frac{1-\alpha}{|C(x)|} & \text{if } x \in C(x) \\ 0 & \text{otherwise} \end{cases} \tag{6}$$

Considering the sentence *"This used to belong to thew queen. They are leaving in about fifteen minuets to go to her house."*, then the noisy channel method would yield:

x	w	x|w	P(x|w)	$P(w\|w_{i-2}, w_{i-1})$	$10^8 P(x\|w)P(w\|w_{i-2}, w_{i-1})$
thew	the	ew|e	0.000007	0.48	333
thew	thew		α=0.95	9.95×10^{-8}	9.45
thew	thaw	e|a	0.001	2.1×10^{-7}	0.0209
thew	threw	h|hr	0.000008	8.9×10^{-7}	0.000713
thew	thwe	ew|we	0.000003	5.2×10^{-9}	0.00000156

Figure B.6 The noisy channel model on 5 possible candidates for thew, with a Stupid Backoff trigram language model computed from the Google N-gram corpus and the error model from Norvig (2009).

4 N-Gram Language Models

4.1 Probabilistic Language Models

Assigning a probability to a sentence allows one to: spell and grammatical error correction, smart typing, machine translation, augmentative and alternative communication systems, speech recognition (e.g. $P(\textit{werewolf}) > P(\textit{where wolf})$), summarization, question-answering, and similar. The goal is to compute the probability of a sentence or sequence of words:

$$P(W) = P(w_1, w_2, \ldots, w_n) \tag{7}$$

One related task is the one of computing the probability of an upcoming word:

$$P(w_5|\overbrace{w_1, w_2, w_3, w_4}^{\text{context}}) \tag{8}$$

A model that computes $P(W)$ or $P(w_5|w_1, w_2, w_3, w_4)$ is called a **language model**. Moreover, one can use the notation w_1^n to indicate a sequence of n words: $w_1, w_2, \ldots, w_n$. Remembering the chain rule:

$$\begin{aligned} P(w_1^n) &= P(w_1)P(w_2|w_1)P(w_3|w_1^2)\cdots P(w_n|w_1^{n-1}) \\ &= \prod_{i=1}^{n} P(w_i|w_1^{i-1}) \end{aligned} \tag{9}$$

For example: $P(\textit{its water is so transparent}) = P(\textit{its})*P(\textit{water}|\textit{its})*P(\textit{is}|\textit{its}, \textit{water})*P(\textit{so}|\textit{its}, \textit{water}, \textit{is})$ $*P(\textit{transparent}|\textit{its}, \textit{water}, \textit{is}, \textit{so})$. In particular, one could be tempted to estimate such probability as follows:

$$P(\textit{the}|\overbrace{\textit{its water is so transparent that}}) = \frac{\text{count}(\textit{its water is so transparent that the})}{\text{count}(\underbrace{\textit{its water is so transparent that}}_{\text{context}})} \tag{10}$$

However, there are too many possible sentences and one will never see enough data for estimating these. Thus, one can make the assumption that: Markov assumption

$$P(\textit{the}|\textit{its water is so transparent that}) \approx P(\textit{the}|\underbrace{\textit{that}}_{k=1}) \tag{11}$$

or

$$P(\textit{the}|\textit{its water is so transparent that}) \approx P(\textit{the}|\underbrace{\textit{transparent, that}}_{k=2}) \tag{12}$$

In general, for $k \geq 1$:

$$P(w_i|w_1^{i-1}) \approx P(w_i|w_{i-k}^{i-1}) \tag{13}$$

Thus:

probability of the word w_i given all the previous words w_1^{i-1}

probability of the word w_i given the probability of the previous k words w_{i-k}^{i-1}

probability of a sentence

$$P(w_1^n) \approx \prod_{i=1}^{n} P(w_i | w_{i-k}^{i-1}) \tag{14}$$

The possible types of model are the following:

- **Unigram model**: history does not matter ($k = 0$):

$$P(w_1^n) \approx \prod_i P(w_i) \tag{15}$$

- **Bigram model**: only the previous word matters ($k = 1$):

$$P(w_1^n) \approx \prod_i P(w_i | w_{i-1}) \tag{16}$$

the main problem of N-gram models are long-range dependencies

- **N-gram models**: the same concept is extended to trigrams, 4-grams, and so on. In general, this is an insufficient model of language (long-range dependencies).

Given these models, one can estimate probabilities using their maximum likelihood estimate. For example, in the case of bigram probabilities:

$$P(w_i | w_{i-1}) = \frac{\text{count}(w_{i-1}, w_i)}{\text{count}(w_{i-1})} \tag{17}$$

Finally, one can do everything in log space so to avoid underflow, and also because adding is faster than multiplying ($\log(\prod_i P_i) = \sum_i \log(P_i)$).

4.2 Evaluating Models

There exist two main approaches to evaluate a language model: extrinsic and intrinsic evaluation:

- **Extrinsic/end-to-end/in-vivo/downstream evaluation** (based on the downstream task) is the only way to know if an improvement in a component is really going to help the task at hand. This approach proceeds as follows:

  1. Put each model in a task.
  2. Run the task, get an accuracy for model A and for model B.
  3. Compare the accuracy for A and B.

  This approach is the best for comparing model A and model B. However, running NLP systems end-to-end is often very costly.

- **Intrinsic evaluation** measures the quality of component independently of the application. This type of evaluation produces a metric to evaluate the potential improvements in a language model. Such metric is probability-based and it is called **perplexity**. Usually, this approach gives a bad approximation of performances, so in general it is only useful in pilot experiments.

  According to perplexity, the best language model is one that best predicts an unseen test set. Perplexity if the inverse probability of the test set, normalized by the number of words:

  $$PP(w_1^N) = P(w_1^N)^{-\frac{1}{N}} = \sqrt[N]{\prod_i \frac{1}{P(w_i|w_{i-1})}} \tag{18}$$

  For example, given a sentence consisting of random digits and a model that assigns $P = 1/10$ to each digit, the perplexity of such model can be computed as:

  $$PP(w_1^N) = P(w_1^N)^{-\frac{1}{N}} = \left(\underbrace{\frac{1}{10} * \frac{1}{10} * \cdots * \frac{1}{10}}_{N \text{ times}} \right)^{-\frac{1}{N}} = \frac{1}{10}^{-1} = 10 \tag{19}$$

  In general, minimizing perplexity is equal to maximizing the test set probability according to the given language model.

Some problems of such language models are the following:

- **Overfitting**. Models work well for prediction if the test corpus is similar to the training corpus, at least if these two share the same genre and dialect. N-grams not seen in the training corpus have zero probability.
- **Out-Of-Vocabulary words** (**OOV**). Open vocabulary systems have to deal with unknown words. To deal with such words one can add a pseudo-word called `<UNK>`. The solution to this problem is to turn the problem into closed vocabulary:
  - either by choosing a vocabulary and modelling all other words as `<UNK>`;
  - or by modelling all least frequent words as `<UNK>`.

  In general, perplexity if artificially reduced by choosing a small vocabulary. Thus, perplexities have to be compared only across language models with the same vocabulary.

4.3 Smoothing

Smoothing is a technique which can be used to avoid the assignment of zero probability to unseen words or contexts. The idea behind this technique is to shave off a bit of probability mass from some more frequent words and give it to unseen events. Below are some of the most important types of smoothing:

- **Laplace smoothing**. This method adds one to all counts and then re-normalizes to obtain a probability. An example on bigram probability estimates is the following:

$$P(w_i|w_{i-1}) = \frac{\text{count}(w_{i-1}, w_i)}{\text{count}(w_{i-1})} \tag{20}$$

$$P^*_{Laplace}(w_i|w_{i-1}) = \frac{\text{count}(w_{i-1}, w_i) + 1}{\text{count}(w_{i-1}) + V} \tag{21}$$

  where the adjusted counts are computed as follows:

$$\text{count}^*(w_i) = (\text{count}(w_i) + 1)\frac{N}{N - V} \tag{22}$$

- **Add-k smoothing**. Is a generalization of the Laplace smoothing:

$$P^*_{Addk}(w_i|w_{i-1}) = \frac{\text{count}(w_{i-1}, w_i) + k}{\text{count}(w_{i-1}) + kV} \tag{23}$$

  Usually k is to be optimized.

- **Backoff**. When considering the problem of finding, for example, $P(w_i|w_{i-2}, w_{i-1})$ with no examples of a particular trigram $w_{i-2}w_{i-1}w_i$, one can estimate such probability using $P(w_i|w_{i-1})$ or simply $P(w_i)$. Namely, one can use less context and back off to a lower-order n-gram. In general, to apply backoff smoothing, one needs additional parameters (α, P^*) to discount the higher-order n-grams and save some probability mass:

$$P_{Backoff}(w_i|w^{i-1}_{i-n+1}) = \begin{cases} P^*(w_i|w^{i-1}_{i-n+1}) & \text{if count}(w^i_{i-n+1}) > 0 \\ \alpha(w^{i-1}_{i-n+1})P_{Backoff}(w_i|w^{i-1}_{i-n+2}) & \text{otherwise} \end{cases} \tag{24}$$

- **Interpolation**. This technique is a generalization of the backoff smoothing:

$$P(w_i|w_{i-2}w_{i-1}) = \lambda_1 P(w_i|w_{i-2}w_{i-1}) + \lambda_2 P(w_i|w_{i-1}) + \lambda_3 P(w_i) \tag{25}$$

  with $\sum_i \lambda_i = 1$.

- **Kneser-Ney**. This method uses absolute discounting and looks at the difference between training set and test set to assign probabilities to `<UNK>`.

- **Stupid backoff**. This method was realised to cope with Google n-grams. It uses backoff smoothing without normalization. Thus, it does not compute probabilities:

$$S(w_i|w^{i-1}_{i-k+1}) = \begin{cases} \frac{\text{count}(w^i_{i-k+1})}{\text{count}(w^{i-1}_{i-k+1})} & \text{if count}(w^i_{i-k+1}) > 0 \\ \lambda S(w_i|w^{i-1}_{i-k+2}) & \text{otherwise} \end{cases} \tag{26}$$

5 Text Classification with Linear Models

Text categorization is the task of assigning a label or category to an entire text of document. Practical examples of this are: sentiment analysis/opinion mining, SPAM detection, language id, authorship attribution, subject category classification. Indeed, many NLP tasks can be seen as classification tasks.
A **classification task**, formally, is a task in which, given an input x (often d for document), and a fixed set of output classes $Y = y_1, \ldots, y_M$ (often c for class), one is supposed to return a predicted class $y \in Y$ for the specific input x. In general, one could use one of the following approaches:

- A rule-based approach.
- A supervised machine learning approach, in which one uses a training set of N labelled documents, $\{d_i, c_i\}$, and a probabilistic classifier which predicts the output class for the a given test sample.

two different categorization

- A generative approach, in which one builds a model of how a class could generate some data. Usually, such systems compare the test data with some probabilistic models to find the best fit for that specific test data. An example of this approach are naive Bayes models.
- A discriminative approach, in which one learns what features are the most useful, without necessarily learning about the classes. An example of this is logistic regression.

5.1 Multinomial Naive Bayes Classifier

The multinomial naive Bayes classifier is a probabilistic classifier which, given d (a document), it returns the class $\hat{c}$ with the maximum posterior probability:

$$\hat{c} = \underset{c \in C}{\operatorname{argmax}} P(c|d) \tag{27}$$

Or, equivalently:

$$\hat{c} = \underset{c \in C}{\operatorname{argmax}} \underbrace{P(d|c)}_{\text{likelihood}} \underbrace{P(c)}_{\text{prior}} \tag{28}$$

If one represents d as a set of features $f_1, \ldots, f_n$:

$$\hat{c} = \underset{c \in C}{\operatorname{argmax}} \underbrace{P(f_1, \ldots, f_n|c)}_{\text{likelihood}} \underbrace{P(c)}_{\text{prior}} \tag{29}$$

Assuming mutual feature independence given the class c:

features are words

$$c_{NB} = \underset{c \in C}{\operatorname{argmax}} P(c) \prod_i P(f_i|c) = \underset{c \in C}{\operatorname{argmax}} P(c) \prod_i P(w_i|c) = \underset{c \in C}{\operatorname{argmax}} \log P(c) + \sum_i \log P(w_i|c) \tag{30}$$

In particular, $P(c)$ and $P(w_i|c)$ are learned from frequencies in the data. Indeed, the maximum likelihood estimates are:

$$\hat{P}(c) = \frac{N_c}{N_{doc}} \tag{31}$$

$$\hat{P}_{Laplace}(w_i|c) = \frac{\text{count}(w_i, c) + 1}{(\sum_{v \in V} \text{count}(v, c)) + |V|} \tag{32}$$

5.1.1 Sentiment Classification

Sentiment analysis is about detecting attitudes. A practical example of sentiment analysis is the following:

Category	Documents
-	just plain boring
-	entirely predictable and lacks energy
-	no surprises and very few laughs
+	very powerful
+	the most fun film of the summer
?	predictable with no fun

where:

$$\frac{P(+|\textit{predictable with no fun})}{P(-|\textit{predictable with no fun})} = \frac{P(+)P(\textit{predictable}|+)P(\textit{no}|+)P(\textit{fun}|+)}{P(-)P(\textit{predictable}|-)P(\textit{no}|-)P(\textit{fun}|-)}$$

Each one of these probabilities can be computed from frequencies. Moreover, there exist some best practices that are commonly used when dealing with sentiment analysis:

- Word occurrence matters more than frequency. Namely, a single occurrence of one word has the same impact of five occurrences of the same word.
- Negation is dealt with by prepending `NOT_` to each word between negation and punctuation. For example:

  ```
  didnt like this movie, but I ...
  ```

  becomes:

  ```
  didnt NOT_like NOT_this NOT_movie, but I ...
  ```

- When the training data is not sufficient, one can use specialized sentiment lexicons. The idea behind lexicons is to focus on specific words instead of using every word as a feature.

5.2 Lexicons for Sentiment, Affect, and Connotation

There are two influential theories of emotion:

- Basic emotions, where fixed and atomic units are combined into others. An example of this is the following:

Word	anger	anticipation	disgust	fear	joy	sadness	surprise	trust	positive	negative
reward	0	1	0	0	1	0	1	1	1	0
worry	0	1	0	1	0	1	0	0	0	1
tenderness	0	0	0	0	1	0	0	0	1	0
sweetheart	0	1	0	0	1	1	0	1	1	0
suddenly	0	0	0	0	0	0	1	0	0	0
thirst	0	1	0	0	0	1	1	0	0	0
garbage	0	0	1	0	0	0	0	0	0	1

- Continuum of emotions in 2/3-dimensional space. An example of this is given by space defined by three different features:
  - **Valence**, which describes the pleasantness of the stimulus.
  - **Arousal**, which describes the intensity of the emotion provoked by the stimulus.
  - **Dominance**, which describes the degree of control exerted by the stimulus.

  Some lexicons assign to each word a value on all three affective dimensions. An example of this is the following:

Valence		Arousal		Dominance	
vacation	.840	enraged	.962	powerful	.991
delightful	.918	party	.840	authority	.935
whistle	.653	organized	.337	saxophone	.482
consolation	.408	effortless	.120	discouraged	.0090
torture	.115	napping	.046	weak	.045

Figure 21.4 Samples of the values of selected words on the three emotional dimensions from Mohammad (2018a).

Lexicons, in general, can be created in different ways: by expert annotators, by crowdsourcing, by semi-supervised induction of labels (i.e. propagating from small set of seed labels), by supervised learning.

5.3 Logistic Regression

Logistic regression is a probabilistic classifier the uses supervised machine learning to directly compute $P(c|d)$, instead of building a model of a class c as Naive Bayes does. The main components of a probabilistic machine learning classifier are the following:

- A feature representation of the input $x_i^{(j)}$.
- A classification function computing $\hat{y}$ via $P(y|x)$.
- An objective function involving minimizing an error.
- An optimization algorithm.

Logistic regression computes the output probability by applying the following procedure:

$$z = \left(\sum_{i=1}^{n} w_i x_i \right) + b = w \cdot x + b \tag{33}$$

then by applying a logistic function to force the output to be a legal probability:

$$P(y = 1) = \sigma(z) = \frac{1}{1 + e^{-z}} = \frac{1}{1 + e^{-(w \cdot x + b)}} \tag{34}$$

$$P(y = 0) = 1 - P(y = 1) = \frac{e^{-(w \cdot x + b)}}{1 + e^{-(w \cdot x + b)}} \tag{35}$$

and lastly by considering 0.5 as decision boundary. An example of input definition is the following:

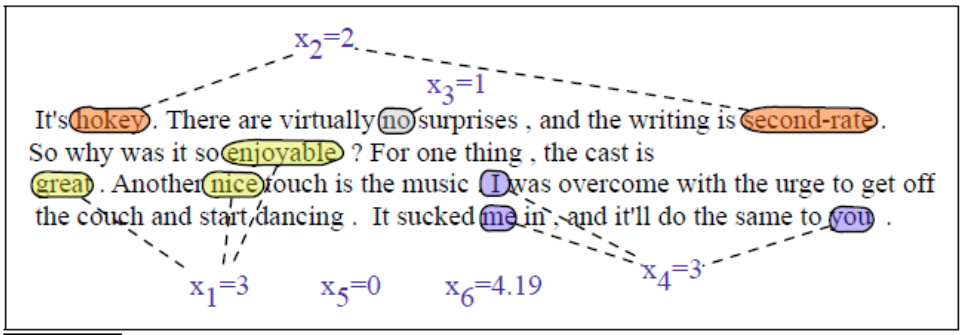


Figure 5.2 A sample mini test document showing the extracted features in the vector x.

where:

- x_1: count(positive lexicon $\in d$).
- x_2: count(negative lexicon $\in d$).

- x_3: 1 if "no" $\in d$, 0 otherwise.
- x_4: count(1st, 2nd pronouns $\in d$).
- x_5: 1 if "!" $\in d$, 0 otherwise.
- x_6: log(word count of d).

The loss function $L(\hat{y}, y)$ is the distance between class and prediction. The main objective is to learn the parameters (w, b) that maximize the probability of the correct labels in the training data given the observations $P(y|x)$:

$$P(y|x) = \hat{y}^y(1-\hat{y})^{1-y} = \begin{cases} 1-\hat{y} & \text{if } y = 0 \\ \hat{y} & \text{if } y = 1 \end{cases} \tag{36}$$

$$\log P(y|x) = y \log \hat{y} + (1-y)\log(1-\hat{y}) \tag{37}$$

Hence, the cross-entropy loss is obtained by flipping the sign:

$$L_{CE}(\hat{y}, y) = -\log P(y|x) = -[y \log \hat{y} + (1-y)\log(1-\hat{y})] \tag{38}$$

L_{CE} for logistic regression is convex. A procedure to find the values for $\theta = (w, b)$ that minimize the loss over training samples (batch of m samples) is given by:

$$\hat{\theta} = \underset{\theta}{\operatorname{argmax}} \sum_{i=1}^{m} \log P(y^{(i)}|x^{(i)}) - \alpha R(\theta) \tag{39}$$

which is exactly what stochastic gradient descent, with mini-batch training, does. The regularization factor $R(\theta)$ is used to avoid overfitting:

- **L2 regularization**: $R(\theta) = \|\theta\|_2^2 = \sum_{j=1}^{n} \theta_j^2$.
- **L1 regularization**: $R(\theta) = \|\theta\|_1 = \sum_{j=1}^{n} |\theta_j|$.

In cases which consider more than two classes, the softmax function can be used to normalize the output into probabilities. In this case, the loss function can be computed as:

$$L_{CE}(\hat{y}, y) = -\sum_{c=1}^{|C|} 1\{y = c\} \log P(y = c|x) \tag{40}$$

5.4 Evaluation Methods

In order to evaluate the performance of a machine learning model one can use a gold standard building a contingency matrix and/or a confusion matrix. Having built these matrices, one can either use a micro-average or macro-average type of metrics (where the possible metrics are precision, recall, accuracy, F_1-score). Moreover, one could also evaluate the model using cross-validation or some statistical significant test.

6 Vector Semantics and Embeddings - Sparse Representations

6.1 Lexical Semantics

The form of a word is the **lemma** or **citation form** for that word all its derivatives. The **word sense** or **concept** is the meaning component of that particular word. In general, lemmas can be polysemous and the word sense of the word can be used for disambiguation. Moreover:

- **Synonymy** is a relation of (near) identity between two sense of two different words. For example, *couch* and *sofa*. However, the **principle of contrast** states that a difference in linguistic form is always associated with some difference in meaning.
- **Antonomy** is about sense that are opposite with respect with one feature of meaning. For example, *dark* and *light*.
- There exist other relations between senses: **subordination**, **hyponymy** or **IS-A** define the concept of subordinate (e.g. *car* is a subordinate of *vehicle*), **superoprdination** or **hypernymy** define the opposite of subordination, and **meronymy**.
- Words have few synonyms, but lots of similar words. For example, *cat* and *dog* are not synonyms, but certainly have similar meaning.
- **Relatedness** is about any relation between words via a semantic frame or field. For example, *car* and *fuel* are somehow related/associated. In general, a **semantic field** is about words that cover a particular domain and bear structured relations with each other. This concept is related to **topic models**. A **semantic frame**, instead, is about words that denote perspectives or participants in a particular type of event.

6.2 Distributional Semantics

Vector semantics define an alternative point of view where words are defined by their usage. In particular, a word is defined by its environment or distribution in the language use, where a distribution is the set of contexts in which it occurs (i.e. the neighbouring words or grammatical environments). Very similar distributions usually imply same meaning. This intuition is called **distributionalist intuition**. However, there exist another intuition, the **vector intuition**, according to which the meaning of a word is represented as a point in an N-dimensional space:

The vector intuition is about embedding each word into a space where similar words are nearby in space. This is the standard way to present meaning in NLP. In particular, to apply such embedding, one would need to chose many things:

- **Matrix design**. Examples are: word × word, word × document, word × discourse context, and others.
- **Reweighting**. Examples are: probabilities, length normalization, TF-IDF, PMI, positive PMI, and others.
- **Dimensionality reduction**. Examples are: latent semantic analysis, non-negative matrix factorization, probabilistic LSA, and others.
- **Vector comparison**. Examples are: Euclidean distance, cosine distance, and others.

6.3 Words and Vectors

A **co-occurrence matrix** represents hoe often words co-occur. For example:

	As You Like It	Twelfth Night	Julius Caesar	Henry V
battle	1	0	7	13
good	114	80	62	89
fool	36	58	1	4
wit	20	15	2	3

Figure 6.2 The term-document matrix for four words in four Shakespeare plays. Each cell contains the number of times the (row) word occurs in the (column) document.

	As You Like It	Twelfth Night	Julius Caesar	Henry V
battle	1	0	7	13
good	114	80	62	89
fool	36	58	1	4
wit	20	15	2	3

Figure 6.3 The term-document matrix for four words in four Shakespeare plays. The red boxes show that each document is represented as a column vector of length four.

is a $|V| \times |D|$ matrix, in which each document is a count vector, identifying a point in a V-dimensional space. Moreover, words can be vectors too. Indeed, one could define a $|V| \times |V|$ **word-word matrix** or **term-term matrix** in which rows are labelled by (target) words, and columns are labelled by (context) cords, rather than documents. In the following example, the context is four words on each side:

is traditionally followed by	**cherry**	pie, a traditional dessert
often mixed, such as	**strawberry**	rhubarb pie. Apple pie
computer peripherals and personal	**digital**	assistants. These devices usually
a computer. This includes	**information**	available on the internet

However, keeping words after the most frequent ones may not be helpful. There is the need to efficiently handle sparse representations.

	aardvark	...	computer	data	result	pie	sugar	...
cherry	0	...	2	8	9	442	25	
strawberry	0	...	0	0	1	60	19	
digital	0	...	1670	1683	85	5	4	
information	0	...	3325	3982	378	5	13	

Figure 6.5 Co-occurrence vectors for four words in the Wikipedia corpus, showing six of the dimensions (hand-picked for pedagogical purposes). The vector for *digital* is outlined in red. Note that a real vector would have vastly more dimensions and thus be much sparser.

To do so, one could use larger windows ans scaling:

	is	then	followed	by	cherry	pie	,	a	dessert	whose
	4	3	2	1	0	1	2	3	4	5
Window: 3	is	then	followed	by	cherry	pie	,	a	dessert	whose
Scaling: flat	0	1	1	1	1	1	1	1	0	
Scaling: $\frac{1}{n}$	0	$\frac{1}{3}$	$\frac{1}{2}$	1	1	1	$\frac{1}{2}$	$\frac{1}{3}$	0	

6.3.1 Vector Comparison

At this point, vectors can be compared using different methods:

- dot-product$(w, v) = w \cdot v = \sum_{i=1}^{n} w_i v_i$.
- length$(v) = |v| = \sqrt{\sum_{i=1}^{n} v_i^2}$.
- cosine$(w, v) = \frac{w \cdot v}{|w||v|}$.

6.3.2 Reweighting

The goal of reweighting is to amplify the important, the trustworthy, the unusual, and to de-emphasize the mundane and the quirky. Using raw counts of words is usually a poor proxy for the above rules. One common way of reweighting is **TF-IDF**. This method works by first computing the term frequency and the inverse document frequency:

- **Term frequency** (**TF**) is the frequency count, usually log-transformed:

$$tf_{t,d} = \log_{10}(\text{count}(t, d) + 1) \tag{41}$$

  Words that occur many times in a document have a high tf.

- **Inverse document frequency** (**IDF**):

$$idf_t = \log_{10}\left(\frac{N}{df_t}\right) \tag{42}$$

  Words that occur only in a few documents have a high idf. df_t is the number of documents where t occurs.

Lastly:

$$tf\text{-}idf_{t,d} = tf_{t,d} * idf_t \tag{43}$$

An alternative to TF-IDF is **positive pointwise mutual information** (**PPMI**), which tries to compute at what extent w_1 and w_2 are associated by determining how much more w_1 and w_2 co-occur than by chance:

$$PMI(w, c) = \log_2 \frac{P(w, c)}{P(w)P(c)} \tag{44}$$

where $P(w, c)$ is the probability of co-occurrence, and $P(w)P(c)$ is the probability of chance co-occurrence. Lastly:

$$PPMI(w, c) = \max(PMI(w, c), 0) \tag{45}$$

However, PMI is biased towards infrequent events. Indeed, very rare words have very high PMI values. One can solve this problem by either use add-k smoothing or by giving rare words slightly higher probabilities:

$$PPMI_\alpha(w, c) = \max\left(\log_2 \frac{P(w, c)}{P(w)P_\alpha(c)}, 0\right) \tag{46}$$

where:

$$P_\alpha(c) = \frac{\text{count}(c)^\alpha}{\sum_c \text{count}(c)^\alpha} \tag{47}$$

Typically, $\alpha = 0.75$. This usually works because $P_\alpha(c) > P(c)$ for rare c.

7 Vector Semantics and Embeddings - Dense Representations

In general, TF-IDF and PPMI vectors are long and sparse (i.e. most elements are zero), while dense vectors are usually short and dense. Indeed, these dense vectors usually do not store explicit counts and may be easier to use as features in machine learning (less weights to tune). Examples of dense embeddings are Word2vec, GloVe, fastText. These three embeddings are built on ideas from neural language modelling.

7.1 Neural Language Modelling

Statistical language modelling suffers from the curse of dimensionality, and a lot of information in the training corpus is usually not used. For example, "*the cat is walking in the bedroom*" should help generalize to make "*a dog running in a room*" almost as likely. The idea of neural language modelling is to predict rather than to count. Each word $w \in V$ is associated with a **word feature vector** $C(w) \in \mathbb{R}^m$, and such vectors are used to express the joint probability function of sequences $f(w_t, \ldots, w_{t-n+1}) = \hat{P}(w_t | w_1^{t-1})$, where C and f are learned simultaneously.

$$\hat{P}(w_t = i | w_1^{t-1}) = f(i, w_t, \ldots, w_{t-n+1}) = g(i, C(w_{t-1}, \ldots, C(w_{t-n+1}))) \tag{48}$$

In general, similar words are expected to have similar feature vectors.

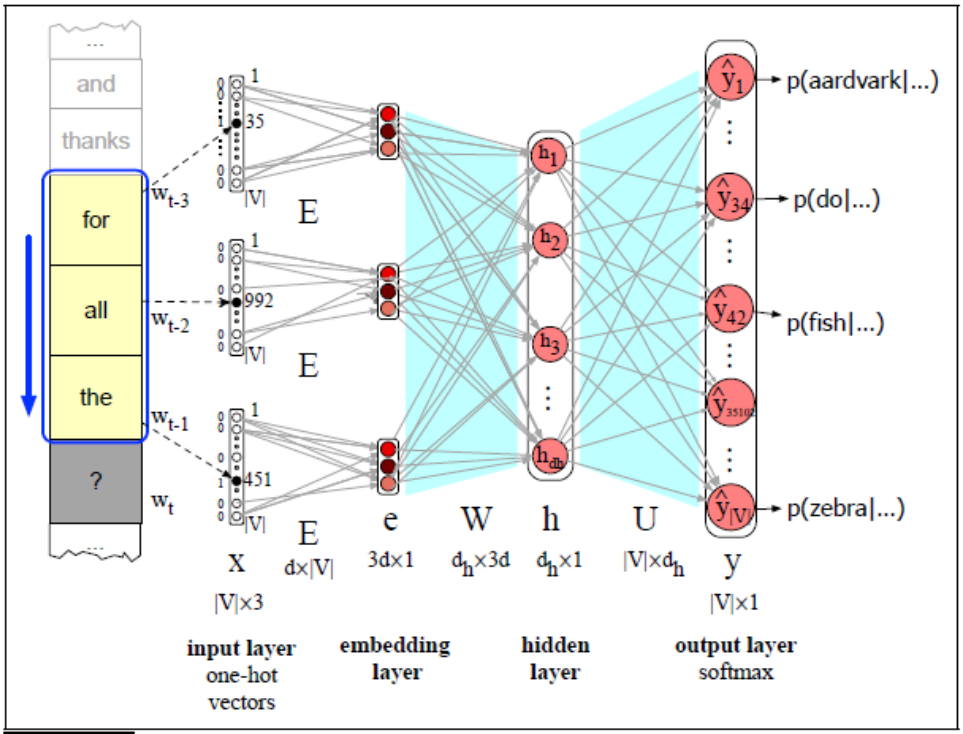
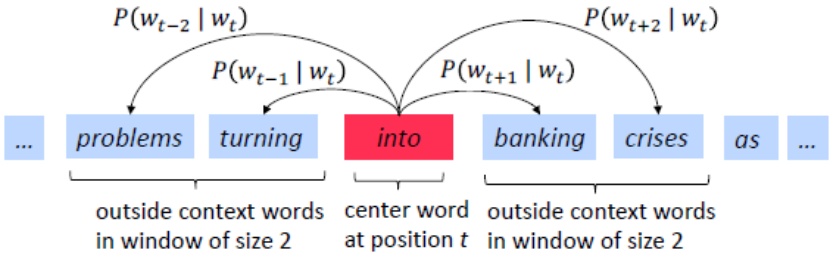


Figure 7.13 Forward inference in a feedforward neural language model.

7.2 Word2vec

Word2vec is the most popular embedding methods. It uses the running text as implicitly supervised training data. In particular, instead of counting how often each word w occurs near *apricot*, a classifier in trained to predict whether w is likely to show up near *apricot*. There is no need for hand-labelled supervision, a word s near *apricot* acts as gold standard to the problem. The result of this process will be a set of trained weights which will be used as word embeddings. The idea here is the following:

- Given a large corpus of text, every word in a fixed vocabulary is represented by a vector. One simply needs to iterate through each word in the corpus, fixing the target, or **center word** (c), and the context (o).
- The similarity between c and o is used to calculate the probability of o given c (or vice versa).
- The word vectors are adjusted so to maximize such probability.



Assuming all context words to be independent, for each position $t = 1, \ldots, T$, the context words within a window of fixed size m, given the center word w_t, are predicted:

$$Likelihood = L(\theta) = \prod_{t=1}^{T} \prod_{-m \leq j \leq m, j \neq 0} P(w_{t+j}|w_t; \theta) \tag{49}$$

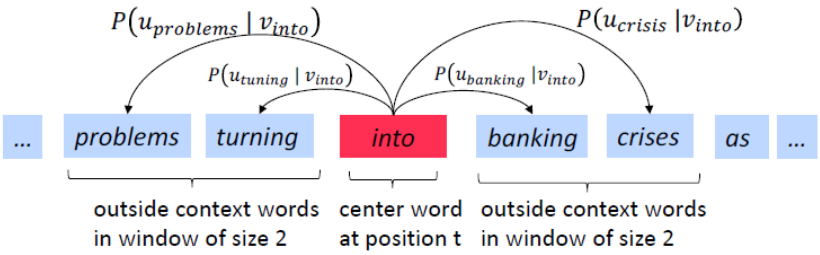
The objective function, $J(\theta)$, is the average negative log-likelihood:

$$J(\theta) = -\frac{1}{T} \log L(\theta) = -\frac{1}{T} \sum_{t=1}^{T} \sum_{-m \leq j \leq m, h \neq 0} \log P(w_{t+j}|w_t; \theta) \tag{50}$$

To calculate $P(w_{t+j}|w_t; \theta)$ one usually uses two vectors for the same word w: u_w when w is an outside word (u_o), and v_w when w is a center word (v_c). At this point one can compute:

$$P(o|c) = \frac{\exp(u_o^{\mathrm{T}} v_c)}{\sum_{w \in V} \exp(u_w^{\mathrm{T}} v_c)} \tag{51}$$

An example of this is the following:



where $P(u_{problems}|v_{into})$ is a shortcut for $P(problems|into; u_{problems}, v_{into}, \theta)$.

7.3 fastText

fastText is an extension of Word2vec which uses subword models to deal with OOV words and sparsity in languages with rich morphology. Each word is represented as itself plus a bag of contituent n-grams. For instance, with $n = 3$, *where* is represented by `<where>`, `<wh`, `whe`, `her`, `ere`, `re>`. Lastly, an embedding is learned for each constituent n-gram, and *where* is represented by the sum of all such embeddings.

7.4 GloVe

In general, there are two families of methods for learning word vectors: global matrix factorization methods, and local context window methods (based on learning the embeddings thanks to neural networks). GloVe combines the ideas of count matrices and neural network predictions. The aim is to have components of meaning as linear operations in the vector space. In particular, it uses **ratios of co-occurrence probabilities** to encode meaning components:

Table 1: Co-occurrence probabilities for target words *ice* and *steam* with selected context words from a 6 billion token corpus. Only in the ratio does noise from non-discriminative words like *water* and *fashion* cancel out, so that large values (much greater than 1) correlate well with properties specific to ice, and small values (much less than 1) correlate well with properties specific of steam.

Probability and Ratio	k = *solid*	k = *gas*	k = *water*	k = *fashion*
$P(k\|ice)$	1.9×10^{-4}	6.6×10^{-5}	3.0×10^{-3}	1.7×10^{-5}
$P(k\|steam)$	2.2×10^{-5}	7.8×10^{-4}	2.2×10^{-3}	1.8×10^{-5}
$P(k\|ice)/P(k\|steam)$	8.9	8.5×10^{-2}	1.36	0.96

These ratios of co-occurrence probabilities can be captured as linear meaning components in a word vector space thanks to a **log-bilinear model**: if the dot product can be made equal to the log of the co-occurrence probability

$$w_i \cdot w_j = \log P(i|j) \tag{52}$$

then a **vector difference** turns into a ratio of the co-occurrence probabilities:

$$w_x \cdot (w_a - w_b) = \log \frac{P(x|a)}{P(x|b)} \tag{53}$$

To obtain that, the loss function is defined as:

$$J = \sum_{i,j=1}^{V} f(X_{ij})(w_i^{\mathrm{T}} \tilde{w}_j + b_i + \tilde{b}_j - \log X_{ij})^2 \tag{54}$$

7.5 Bias and Embeddings

Examples of embeddings encode cultural stereotypes are the following:

- Embeddings pinpoint sexism implicit in text.
- Embeddings reflect and replicate all sorts of pernicious biases. An example of this is that embeddings for competence adjectives are biased toward men. Fortunately this bias is slowly decreasing.
- Embeddings reflect ethnic stereotypes over time.

8 Part-of-Speech Tagging

Parts of speech/word classes/syntactic categories are useful abstractions which reveal a lot about words and their neighbours. Parts of speech are defined based on syntactic and morphological function. There exist two board super-categories:

- **Closed class** types, such as prepositions like *from* and *to*, and function words that tend to be short, frequent, and have structuring uses in grammar. Examples of these types are propositions, particles, determiners, conjunctions, pronouns, auxiliary verbs, numerals, and others.
- **Open class** types, such as nouns, verbs, adjectives, and adverbs.

The Penn Treebank POS tags are defined as follows:

Tag	Description	Example	Tag	Description	Example	Tag	Description	Example
CC	coordinating conjunction	*and, but, or*	PDT	predeterminer	*all, both*	VBP	verb non-3sg present	*eat*
CD	cardinal number	*one, two*	POS	possessive ending	*'s*	VBZ	verb 3sg pres	*eats*
DT	determiner	*a, the*	PRP	personal pronoun	*I, you, he*	WDT	wh-determ.	*which, that*
EX	existential 'there'	*there*	PRP$	possess. pronoun	*your, one's*	WP	wh-pronoun	*what, who*
FW	foreign word	*mea culpa*	RB	adverb	*quickly*	WP$	wh-possess.	*whose*
IN	preposition/ subordin-conj	*of, in, by*	RBR	comparative adverb	*faster*	WRB	wh-adverb	*how, where*
JJ	adjective	*yellow*	RBS	superlatv. adverb	*fastest*	$	dollar sign	*$*
JJR	comparative adj	*bigger*	RP	particle	*up, off*	#	pound sign	*#*
JJS	superlative adj	*wildest*	SYM	symbol	*+,%, &*	“	left quote	‘ or “
LS	list item marker	*1, 2, One*	TO	“to”	*to*	”	right quote	’ or ”
MD	modal	*can, should*	UH	interjection	*ah, oops*	(	left paren	[, (, {, <
NN	sing or mass noun	*llama*	VB	verb base form	*eat*	)	right paren	],), }, >
NNS	noun, plural	*llamas*	VBD	verb past tense	*ate*	,	comma	,
NNP	proper noun, sing.	*IBM*	VBG	verb gerund	*eating*	.	sent-end punc	. ! ?
NNPS	proper noun, plu.	*Carolinas*	VBN	verb past part.	*eaten*	:	sent-mid punc	: ; ... – -

Figure 8.1 Penn Treebank part-of-speech tags (including punctuation).

Part-of-speech tagging is about assigning a POS marker to each input token. This can be seen as a disambiguation task, and the goal of POS tagging is to resolve ambiguities. For example, *that*, *back*, *down*, *put* and *set* are among the most frequent ambiguous words:

- *earning growth took a back/JJ seat.*
- *a small building in the back/NN.*
- *a clear majority of senators back/VBP the bill.*
- *Dave began to back/VB toward the door.*
- *enable the country to buy back/RP its debt.*
- *I was twenty-one back/RB then.*

However, not all tags are equally likely.

8.1 Hidden Markov Model Part-of-Speech Tagging

This process relies on a probabilistic sequence model (sequence classifier), to assign a label to each unit in a sequence. Given a sequence of units, this model returns a probability distribution over the possible sequences of labels. Such models are based on the **Markov assumption**, in which only the current state matters:

$$P(q_i = a|q_1 \dots q_{i-1}) = P(q_i = a|q_{i-1}) \tag{55}$$

and are based on the **Markov chain**:

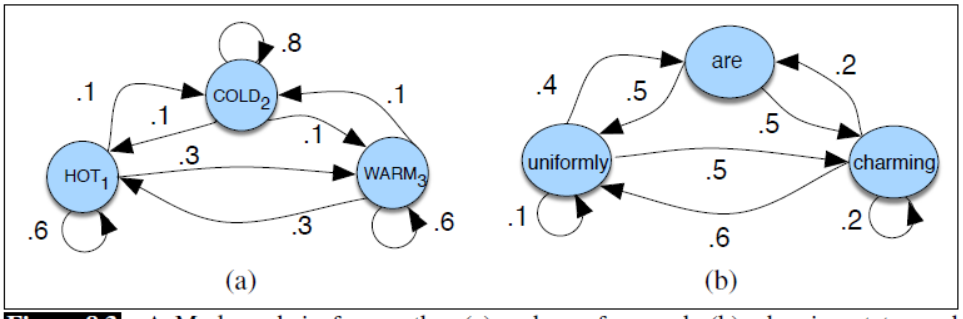


Figure 8.3 A Markov chain for weather (a) and one for words (b), showing states and transitions. A start distribution π is required; setting $\pi = [0.1, 0.7, 0.2]$ for (a) would mean a probability 0.7 of starting in state 2 (cold), probability 0.1 of starting in state 1 (hot), etc.

Formally, a Markov chain is defined by a tuple composed of three elements:

- Q, $\{q_i\}_{i=1}^N$, set of N states.
- A, $[a_{ij}]_{i,j=1}^N$, a transition probability matrix A, where each a_{ij} represents the probability of moving from state i to state j, such that $\sum_{j=1}^N a_{ij} = 1$.
- π, $[\pi_i]_{i=1}^N$, an initial probability distribution over states, such that $\sum_{i=1}^N \pi_i = 1$. π_i is the probability that the Markov chain will start in state i.
- O, $[o]_{i=1}^T$, a sequence of T observations, each one drawn from a vocabulary $V = v_1, \dots v_V$.
- B, $[b_i(o_t)]_{i=1,t=1}^{N\ T}$, a sequence of observation likelihoods, also called emission probabilities, each expressing the probability of an observation o_t being generated from a state q_i.

Usually, the events one is interested in are hidden. For instance, POS tags are not explicit in the text. An HMM represents both observed and hidden events. Based on the Markov assumption, then the output independence assumption states that:

$$P(o_i|q_1 \dots q_i \dots q_T, o_1 \dots o_i \dots o_T) = P(o_i|q_i) \tag{56}$$

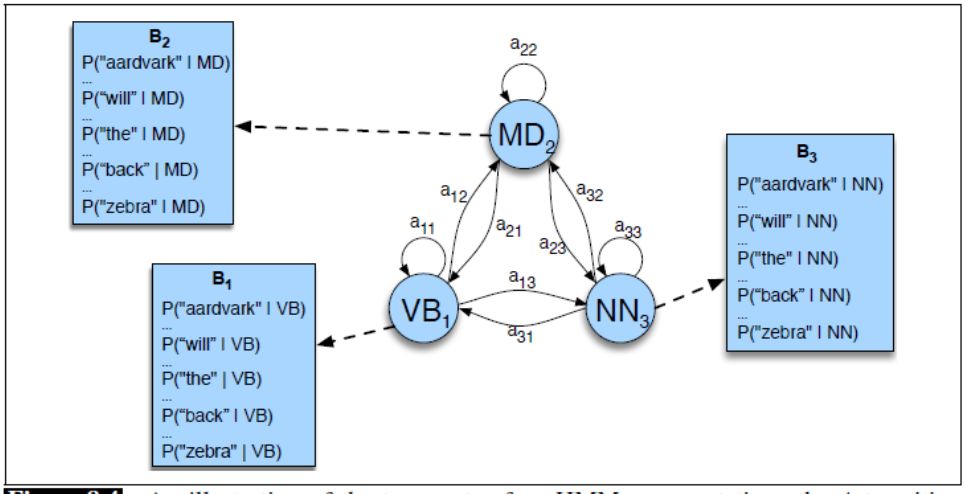


Figure 8.4 An illustration of the two parts of an HMM representation: the A transition probabilities used to compute the prior probability, and the B observation likelihoods that are associated with each state, one likelihood for each possible observation word.

An HMM tagger is composed of two main components: A and B probabilities. For example, *Janet will back the bill*:

- $A : P(t_i|t_{i-1})$:

$$P(VB|MD) = \frac{\text{count}(MD, VB)}{\text{count}(MD)}$$

- $B : P(w_i|t_i)$:

$$P(back|VB) = \frac{\text{count}(VB, back)}{\text{count}(VB)}$$

Decoding is about finding the most probable sequence of hidden states, starting from an observations sequence. Given an HMM $\lambda = (A, B)$, and a sequence of observations $w_1^n = w_1 \dots w_n$, the most probable sequence of hidden states $\hat{t}_1^n$ is given by:

$$\hat{t}_1^n = \underset{t_1^n}{\text{argmax}}\, P(t_1^n|w_1^n) \tag{57}$$

By Bayes' rule:

$$\hat{t}_1^n = \underset{t_1^n}{\text{argmax}}\, P(w_1^n|t_1^n)P(t_1^n) \tag{58}$$

By the Markov assumption (bigram assumption):

$$\hat{t}_1^n \approx \underset{t_1^n}{\text{argmax}}\, P(w_1^n|t_1^n) \prod_{i=1}^{n} P(t_i|t_{i-1}) \tag{59}$$

By output independence:

$$\hat{t}_1^n \approx \underset{t_1^n}{\operatorname{argmax}} \prod_{i=1}^{n} \underbrace{P(w_i|t_i)}_{\text{emission}} \underbrace{P(t_i|t_{i-1})}_{\text{transition}} \tag{60}$$

8.2 The Viterbi Algorithm

The Viterbi algorithm is a dynamic programming decoding algorithm for HMMs. It uses a probability matrix (lattice) $[v_t(j)]_{1\leq t\leq T, 1\leq j\leq N}$, where the columns are observations (tokens), the rows are hidden sates (tags), and each cell represents the probability that the HMM is in state j after the first t observations and passing through the most probable state sequence $q_1 \dots q_{t-1}$. Moreover, $v_t(j)$ is recursively defined based on the most probable path leading to it:

$$\begin{aligned} v_t(j) &= \max_{q_1^{t-1}} P(q_1^{t-1}, o_1^t, q_t = j|A, B) \\ &= \max_{i=1}^{n} \underbrace{v_{t-1}(i)}_{\text{prev. Viterbi path prob.}} \underbrace{a_{ij}}_{\text{transition prob.}} \underbrace{b_j(o_t)}_{\text{emission prob.}} \end{aligned} \tag{61}$$

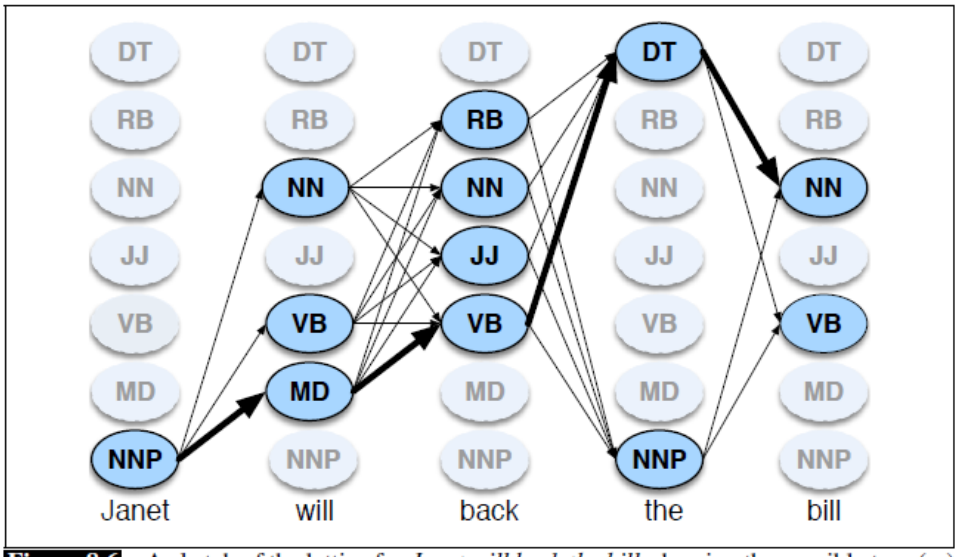


Figure 8.6 A sketch of the lattice for *Janet will back the bill*, showing the possible tags (q_i) for each word and highlighting the path corresponding to the correct tag sequence through the hidden states. States (parts of speech) which have a zero probability of generating a particular word according to the B matrix (such as the probability that a determiner DT will be realized as *Janet*) are greyed out.

This algorithm is slow when the number of states is large ($O(N^2T)$). This can be improved by keeping only the best few hypothesis. One can extend the Viterbi algorithm by using **beam search**, whose approach works as follows:

1. Compute Viterbi scores for each cell.
2. Sort the scores.
3. Keep only the β best-scoring states.

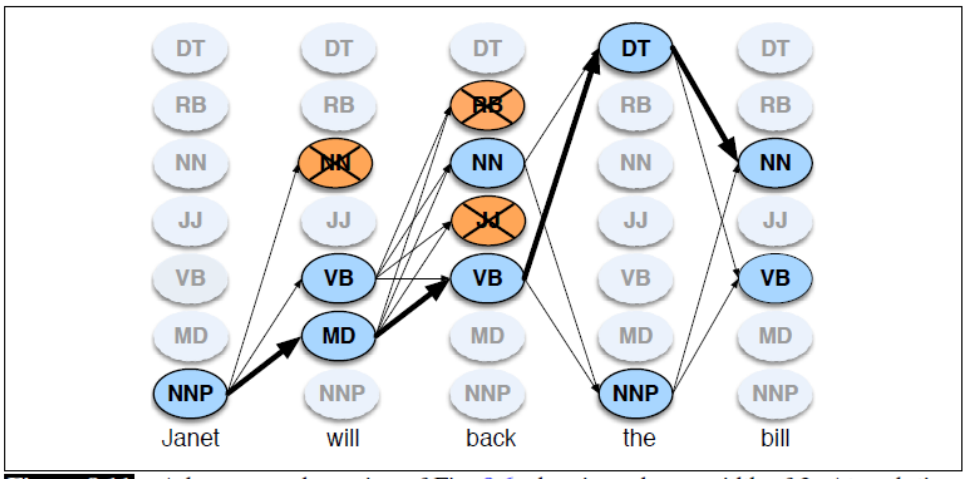


Figure 8.11 A beam search version of Fig. 8.6, showing a beam width of 2. At each time t, all (non-zero) states are computed, but then they are sorted and only the best 2 states are propagated forward and the rest are pruned, shown in orange.

9 Grammars and Parsing

9.1 Constituency Grammars

In the previous sections, some of the ways in which words are arranged together (namely, N-grams, POS-categories) have been discussed. There exist other ways to describe such concept. In particular:

- **context-free grammars** can express more sophisticated relations among words. These grammars are the backbone of many formal models of syntax of natural and computer languages.
- **Syntactic dependencies** is another formalism for describing a syntax.

In general, **constituents** are groups of words behaving as single units. For example, phrases that appear in similar syntactic environments (e.g. before a verb) are evidence of constituency. The most widely used formal system for modelling constituent structure in many languages is given by context-free grammars. A context-free grammar is composed of:

- A set of **terminal** symbols identifying tokens in the language.
- A set of **non-terminal** symbols to denote syntactic categories.
- A set of **rules** or **productions**.
- A designated **start** symbol.

An example of context-free grammar is given by $\mathcal{L}_0$, whose lexicon is given by: Noun, Verb, Adjective, Pronoun, Proper-Noun, Determiner, Preposition, Conjunction.

Grammar Rules			Examples
S	→	*NP VP*	I + want a morning flight
NP	→	*Pronoun*	I
	|	*Proper-Noun*	Los Angeles
	|	*Det Nominal*	a + flight
Nominal	→	*Nominal Noun*	morning + flight
	|	*Noun*	flights
VP	→	*Verb*	do
	|	*Verb NP*	want + a flight
	|	*Verb NP PP*	leave + Boston + in the morning
	|	*Verb PP*	leaving + on Thursday
PP	→	*Preposition NP*	from + Los Angeles

Figure 12.3 The grammar for $\mathcal{L}_0$, with example phrases for each rule.

$\mathcal{L}_0$ defines a formal language, and can be seen as a simplified model of how natural languages really work. In general, there exist a hierarchy of grammars (Chomsky hierarchy):

- **Type 0**: **recursively enumerable**, where the production rules are of the form $\alpha \rightarrow \beta$ with α being not empty.
- **Type 1**: **context dependent**, where the production rules are of the form $\alpha A \beta \rightarrow \alpha \pi \beta$ with A being a non-terminal symbol and π being not empty.
- **Type 2**: **context free**, where the production rules are of the form $A \rightarrow \pi$ with A being a non-terminal symbol and π being not empty.
- **Type 3**: **regular**, where the production rules are of the form $A \rightarrow a$ or $A \rightarrow aB$ with A and B being non-terminal symbols and a being a terminal. These grammars can be divided into right-linear grammars ($A \rightarrow aB$), and left-linear grammars ($A \rightarrow Ba$).

Given a grammar G, the task of verifying if a string α belongs to the language $\mathcal{L}(G)$ is called **syntactic analysis** or **parsing** of α. Syntactic parsing relies on derivations: one string derives another one if it can be re-written as the second one by some series of rule applications:

- When dealing with regular languages, one can apply parsing via finite-state automaton.
- When dealing with context-free grammars, one can apply parsing via push-down automaton. These grammars are interesting since they model programming languages.

Context-free grammars are a purely declarative formalism. Parsing algorithms employ context-free grammars to produce parse trees, which are useful in grammar checking, semantic analysis, question answering, and information extraction. However, one grammar can assign more than one parse to a sentence. Common types of ambiguity are:

- **Attachment ambiguity**, which occurs if a particular constituent can be attached to the parse tree at more than one place.
- **Coordination ambiguity**, which occurs if different sets of phrases can be conjoined by a conjunction like *and*.

Semantic disambiguation is the process of choosing the correct parse of the same sentence.

9.1.1 Probabilistic Context-Free Grammars

To address structural ambiguity, one can compute the probability of each interpretation anc choose the most probable interpretation. Probabilistic context-free grammars annotate production rules with conditional probabilities:

$$A \rightarrow \beta \ [p] \tag{62}$$

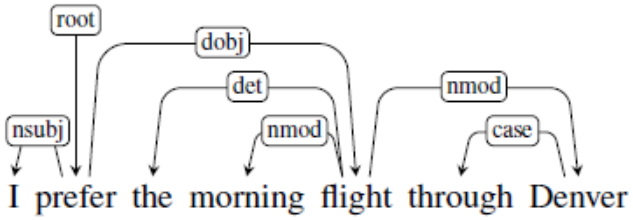
where p expresses $P(\beta|A)$ or $P(A \rightarrow \beta|A)$ or $P(\text{RHS}|\text{LHS})$. Moreover, $0 \leq p \leq 1$ and $\sum_{\beta} P(A \rightarrow \beta) = 1$.

Grammar		Lexicon
$S \rightarrow NP\ VP$	[.80]	$Det \rightarrow that$ [.10] | a [.30] | the [.60]
$S \rightarrow Aux\ NP\ VP$	[.15]	$Noun \rightarrow book$ [.10] | $flight$ [.30]
$S \rightarrow VP$	[.05]	| $meal$ [.05] | $money$ [.05]
$NP \rightarrow Pronoun$	[.35]	| $flight$ [.40] | $dinner$ [.10]
$NP \rightarrow Proper\text{-}Noun$	[.30]	$Verb \rightarrow book$ [.30] | $include$ [.30]
$NP \rightarrow Det\ Nominal$	[.20]	| $prefer$ [.40]
$NP \rightarrow Nominal$	[.15]	$Pronoun \rightarrow I$ [.40] | she [.05]
$Nominal \rightarrow Noun$	[.75]	| me [.15] | you [.40]
$Nominal \rightarrow Nominal\ Noun$	[.20]	$Proper\text{-}Noun \rightarrow Houston$ [.60]
$Nominal \rightarrow Nominal\ PP$	[.05]	| NWA [.40]
$VP \rightarrow Verb$	[.35]	$Aux \rightarrow does$ [.60] | can [.40]
$VP \rightarrow Verb\ NP$	[.20]	$Preposition \rightarrow from$ [.30] | to [.30]
$VP \rightarrow Verb\ NP\ PP$	[.10]	| on [.20] | $near$ [.15]
$VP \rightarrow Verb\ PP$	[.15]	| $through$ [.05]
$VP \rightarrow Verb\ NP\ NP$	[.05]	
$VP \rightarrow VP\ PP$	[.15]	
$PP \rightarrow Preposition\ NP$	[1.0]	

Figure 14.1 A PCFG that is a probabilistic augmentation of the $\mathscr{L}_1$ miniature English CFG

9.2 Dependency Grammars

Dependency grammars is a different family of formalisms, where there is no direct role of phrasal constituents and phrase-structure rules. In particular, the syntactic structure of sentences is described in terms of words and binary grammatical relations among words, where these relations are illustrated with directed, labelled arcs from **heads** to **dependents**.



In general, these grammars have a typed dependency structure, meaning that the labels are drawn from a fixed inventory of grammatical relations, and the root node marks the head of the entire structure.

10 Sequence Processing with Recurrent Networks

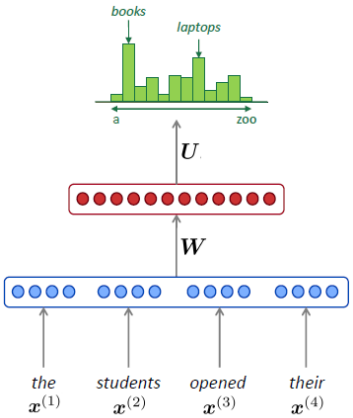
Language is inherently sequential. Neural language modelling works by applying the following:

output distribution
$\hat{y} = softmax(Uh + b_2) \in \mathbb{R}^{|V|}$

hidden layer
$h = f(We + b_1)$

concatenated word embeddings
$e = [e^{(1)}; e^{(2)}; e^{(3)}; e^{(4)}]$

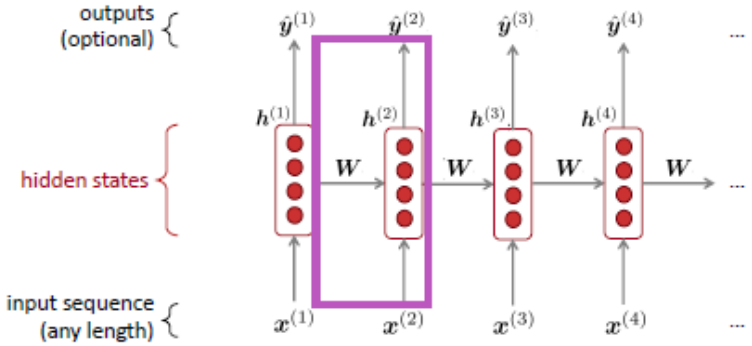
words / one-hot vectors
$x^{(1)}, x^{(2)}, x^{(3)}, x^{(4)}$



While a fixed-window neural language model is an improvement over N-gram language models, there still exist some problems such as: fixed content size (many language tasks require access to arbitrarily distant information), difficulty in learning systematic patters.

10.1 Recurrent Neural Networks

The idea behind recurrent neural networks is to apply the same weights W repeatedly:



These networks contain cycles, and the value of a unit depends on the earlier outputs as an input:

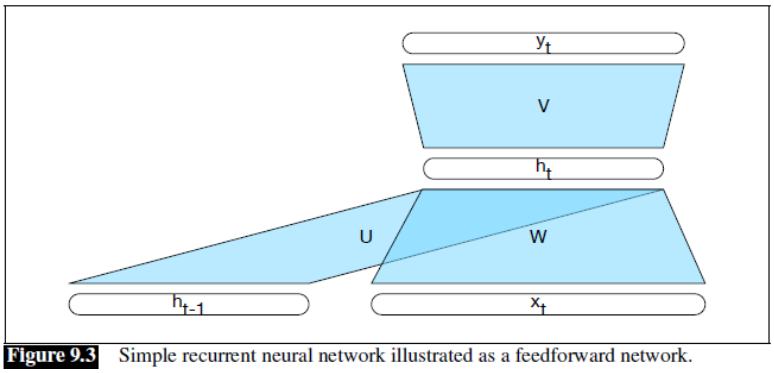


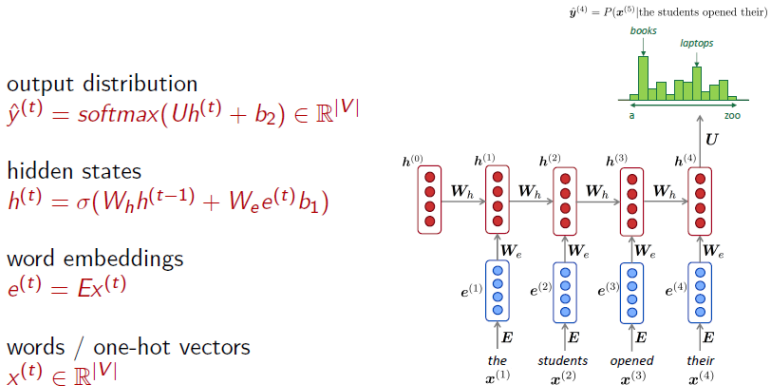
Figure 9.3 Simple recurrent neural network illustrated as a feedforward network.

where:

$$h_t = g(Uh_{t-1} + Wx_t) \tag{63}$$

$$y_t = f(Vh_t) \tag{64}$$

An initial h_0 is given, and f is usually a softmax. In particular, the input vector x_t is multiplied by a weight matrix W and passed through a non-linear activation function g to obtain the activation value for the hidden unit h_t. Such unit is also fed a recurrent link from h_{t-1}. h_{t-1} provides a form of memory or context, encoding earlier processing. Thus, there is no fixed limit on prior context, and the model's size does not increase for longer inputs. Lastly, the same weights are applied on every time step (less memory occupation).



In order to train an RNN language model:

- Get a corpus of text which is a sequence of words $x^{(1)}, \ldots, x^{(T)}$.
- Feed into RNN-LM and compute the output distribution $\hat{y}^{(t)}$ for every step t.
- The loss function on step t is the cross-entropy between the predicted probability distributions $\hat{y}^{(t)}$ and the true next word $y^{(t)}$ (one-hot encoding for $x^{(t+1)}$):

$$J^{(t)}(\theta) = CE(y^{(t)}, \hat{y}^{(t)}) = -\sum_{w \in V} y_w^{(t)} \log \hat{y}_w^{(t)} = -\log \hat{y}_{x_{t+1}}^{(t)} \tag{65}$$

- Average the loss values to get the overall loss for the entire training set:

$$J(\theta) = \frac{1}{T} \sum_{t=1}^{T} J^{(t)}(\theta) = \frac{1}{T} \sum_{t=1}^{T} -\log \hat{y}_{x_{t+1}}^{(t)} \tag{66}$$

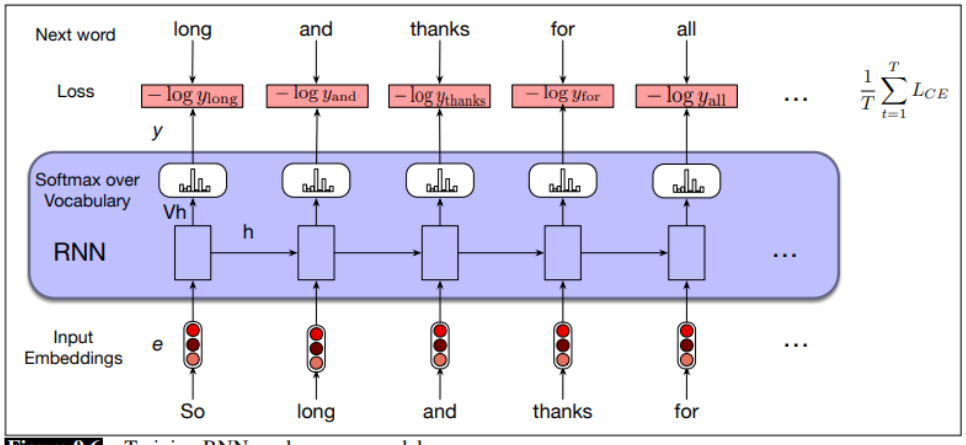


Figure 9.6 Training RNNs as language models.

However, computing the loss and gradients across the entire corpus is usually too expensive. Indeed, the derivative of $J^{(t)}(\theta)$ with respect to the repeated weight matrix W is computed, for every step t, as the sum of the gradients of $J^{(t)}(\theta)$ with respect to W for each step i that precedes t:

$$\frac{\partial J^{(t)}}{\partial W_h} = \sum_{i=1}^{t} \frac{\partial J^{(t)}}{\partial W_h}\bigg|_{(i)} \tag{67}$$

Examples of RNN applications are:

- Autoregressive generation (e.g. predicting words).
- Sequence labelling (e.g. POS-tagging).
- Sequence classification (e.g. sentiment analysis).
- Text generation (e.g. speech recognition, machine translation, summarization).

10.1.1 Fancier Architectures

Some fancier architectures are the following:

- **Stacked RNN**, where the sequence of outputs from one RNN is used as input to another RNN. This induces representations at different levels of abstraction. However, the training cost rises steeply with the number of levels.

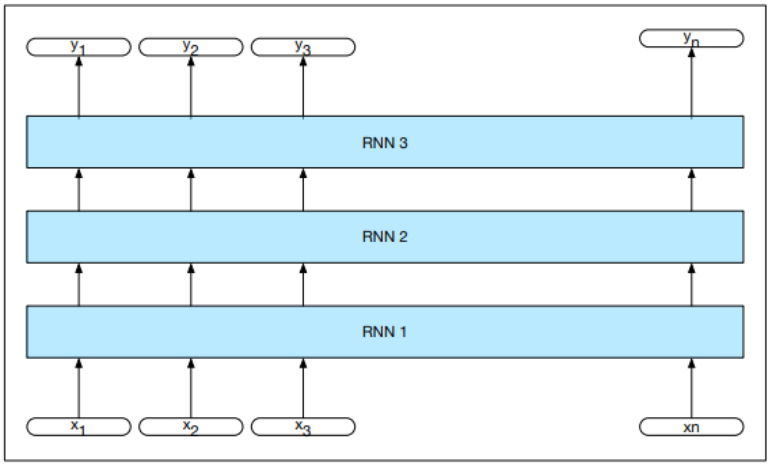


Figure 9.10 Stacked recurrent networks. The output of a lower level serves as the input to higher levels with the output of the last network serving as the final output.

- **Bidirectional RNN**, useful when the entire input sequence is available all at once (usually not for language models). These networks exploit the context to the right of the current input as well. Two independent RNNs, whose outputs are usually concatenated.

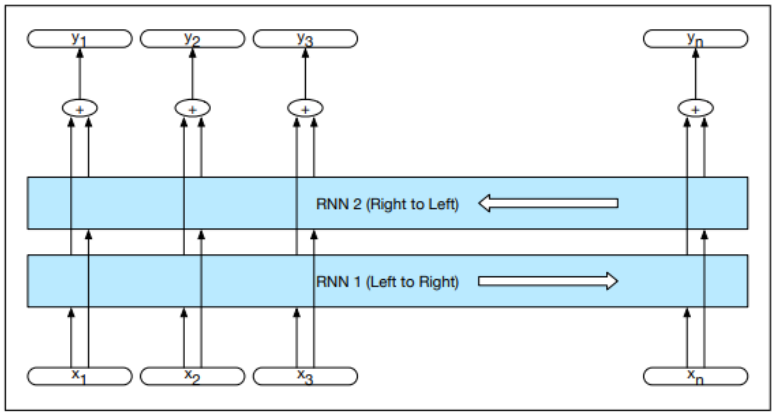


Figure 9.11 A bidirectional RNN. Separate models are trained in the forward and backward directions with the output of each model at each time point concatenated to represent the state of affairs at that point in time. The box wrapped around the forward and backward network emphasizes the modular nature of this architecture.

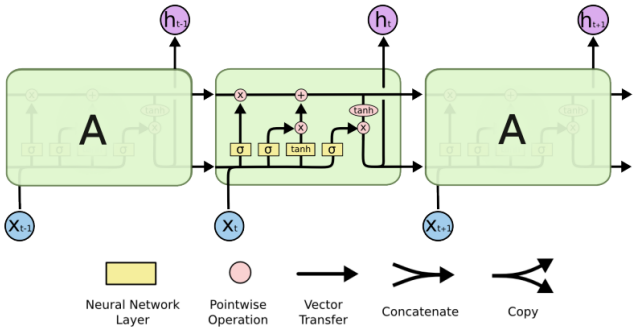
10.2 Exploding and Vanishing Gradient

In general:

- If the gradient becomes too big, then the SGD update step ($\theta^{new} = \theta^{old} - \alpha \nabla_\theta J(\theta)$) becomes too big. A solution to this problem is **gradient clipping** (i.e. scaling down the gradient if its norm exceeds a threshold).
- If the gradient is too small (neamely, less then zero), then the gradient signal from faraway is lost, and the model weights are updated only with respect to near effects, and not to long-term effects. As a solution to the vanishing gradient problem

10.2.1 Long Short-Term Memory

A solution to the vanishing gradient problem are long short-term memory (LSTM) networks. These networks are a type of RNN which, on step t, have an hidden state $h^{(t)}$ and a cell state $c^{(t)}$. Both are vectors of the same length n, and the cell state stores long-term information. In particular, the LSTM can erase, write and read information from the cell state, and the selection of which information to erase/write/read is controlled by three corresponding gates.



The values of the gates are computed based on the current context:

- **Forget gate** controls what is kept and what is forgotten from the previous cell state:

$$f^{(t)} = \sigma(W_f h^{(t-1)} + U_f x^{(t)} + b_f) \tag{68}$$

- **Input gate** controls what parts of the new cell state content are written to the cell state:

$$i^{(t)} = \sigma(W_i h^{(t-1)} + U_i x^{(t)} + b_i) \tag{69}$$

- **Output gate** controls what parts of the cell state are output to the hidden state:

$$o^{(t)} = \sigma(W_o h^{(t-1)} + U_o x^{(t)} + b_o) \tag{70}$$

Moreover:

- **New cell state content** is computed as:

$$\tilde{c}^{(t)} = \tanh(W_c h^{(t-1)} + U_c x^{(t)} + b_c) \tag{71}$$

- **Cell state**: erase come content from last cell state (forget), and write some new cell content (intput):

$$c^{(t)} = f^{(t)} \cdot c^{(t-1)} + i^{(t)} \cdot \tilde{c}^{(t)} \tag{72}$$

- **Hidden state**: read some content from the cell (output):

$$h^{(t)} = o^{(t)} \cdot \tanh(c^{(t)}) \tag{73}$$

The LSTM architecture makes it easier for the RNN to preserve information over many steps. For example, if the forget state is set to 1 for a cell and the input gate is set to 0, then the information of that cell is preserved indefinitely. By contrast, it is harder for a vanilla RNn to learn a recurrent weight matrix W_h that preserves information in the hidden state.

10.2.2 Gated Recurrent Units

Gated recurrent units (GRU) have been proposed as a simpler alternative to the LSTM. On each step t, one has the input $x^{(t)}$ and the hidden state $h^{(t)}$, and no cell state. Moreover:

- **Update gate** controls what parts of the hidden state are updated and what parts are preserved:

$$u^{(t)} = \sigma(w_u h^{(t-1)} + U_u x^{(t)} + b_u) \tag{74}$$

- **Reset gate** controls what parts of the previous hidden state are used to compute the new content:

$$r^{(t)} = \sigma(W_r h^{(t-1)} + U_r x^{(t)} + b_r) \tag{75}$$

- **New hidden state content**: the reset gate selects useful parts of the previous hidden state. This is used, alongside the current input, to compute the new hidden content:

$$\tilde{h}^{(t)} = \tanh(W_h(r^{(t)} \cdot h^{(t-1)}) + U_h x^{(t)} + b_h) \tag{76}$$

- **Hidden state**: the update gate simultaneously controls what is kept from the previous state, and what is updated in the new hidden state content:

$$h^{(t)} = (1 - u^{(t)}) \cdot h^{(t-1)} + u^{(t)} \cdot \tilde{h}^{(t)} \tag{77}$$

11 Machine Translation, Sequence-to-Sequence and Attention

11.1 Machine Translation

Machine translation is the task of translating a sentence x from one language (the source language) to a sentence y in another language (the target language). Machine translation was first introduced in the early 1950s, using rule-based systems using bilingual dictionaries. Then, in the 1990-2010s period, machine translation was carried out using statistical machine translation, while now neural machine translation is what is mostly used. In particular:

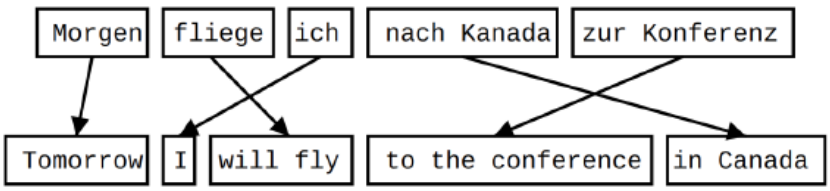
- **Statistical machine translation** is based on learning a probabilistic model from the data: learning which is the best y sentence given the x sentence ($\text{argmax}_y P(y|x)$). This approach uses Bayes rule to split such probability into two components to be learnt separately:
  - Translation model: fidelity, learnt from parallel data.
  - Language model: fluency, learnt from monolingual data.

$$\underset{y}{\text{argmax}} \quad \underbrace{P(x|y)}_{\text{translation model}} \quad \underbrace{P(y)}_{\text{language model}} \tag{78}$$

  In particular, the translation model $P(x|y)$ can be learned by breaking it down even further:

$$P(x, a|y) \tag{79}$$

  where a is the **alignment**, i.e. the word-level correspondence between source and target:

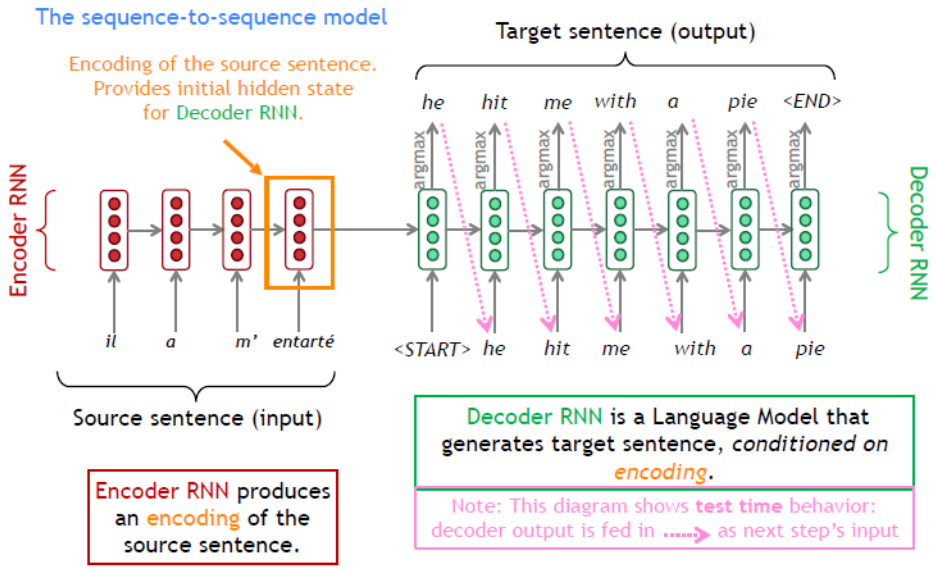


  In general, differences in structural/functional features between languages lead to complicated alignments. Moreover, alignment can be one-to-many, and some words do not have a counterpart. $P(x, a|y)$ is learned as a combination of many factors, including:

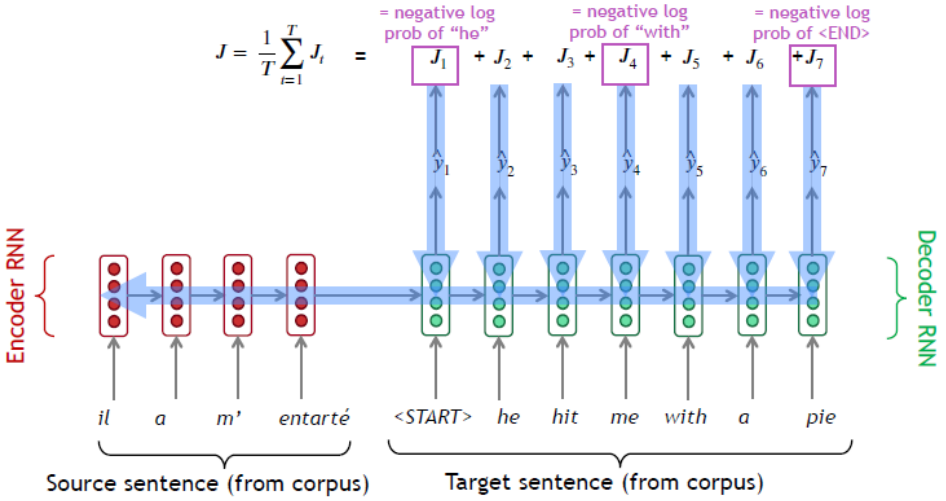
  - The probability of particular words aligning.
  - The probability of particular words having particular fertility (i.e. number of corresponding words)

  Lastly, alignments a are latent variables, thus they are not explicitly specified in the data, and require the use of special learning algorithms for learning the parameters of the distributions with such latent variables.

- **Neural machine translation** is based on doing machine translation using a single neural network which is called sequence-to-sequence or seq2seq. This approach involves two RNNs: one encoder and one decoder.



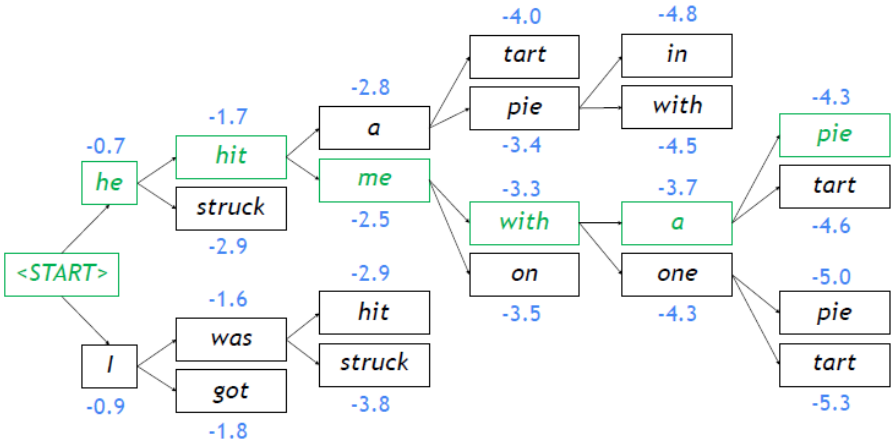
In order to train such systems, one can compute the total loss as:



In the aforementioned case, decoding is applied in a greedy way: the target sentence is generated by taking the argmax on each step of the decoder. However, this could generate a non-optimal translation. To solve this, one could one **beam search decoding**: on each step of the decoder,

one keeps track of the k most probable partial translations (i.e. hypothesis). These hypothesis are given a score of:

$$\text{score}(y_1, \ldots, y_t) = \log P_{LM}(y_1, \ldots, y_t | x) \tag{80}$$



Different hypothesis may produce the `<END>` tokens on different steps. An hypothesis that produced `<END>` is marked as complete. Complete hypothesis are placed aside and other hypothesis are explored, using beam search, until n hypothesis, or a maximum predefined length, are reached. Finally, the best hypothesis are selected by computing $\text{score}(y_1, \ldots, y_t)/t$.

11.2 Evaluation Metrics

Machine translation can be evaluated in different ways:

- **Bilingual Evaluation Understudy** (**BLEU**) compares the candidate machine-writtent translation to one or several reference human-written translation(s). The proposed similarity score is based on:
  - N-gram precision score (usually for 1, 2, 3 and 4-grams): the fraction of n-grams in the candidate translation that also appear in the reference translation.
  - Count at most one meatch for each n-gram in the reference translation.
  - Add a penalty for too-short system translations.

  BLEU is useful but imperfect. There are many valid ways to translate a sentence, so a good translation can get a poor BLEU score because it has low n-gram overlap with the human translation.
- **Character F-score** (**chrF**) is a more recent alternative to BLEU. It is based on character n-grams:

– $chrP$ is the percentage of char 1-grams, 2-grams, ..., k-grams in the hypothesis that occur in the reference, averaged.
– $chrR$ is the percentage of char 1-grams, 2-grams, ..., k-grams in the reference that occur in the hypothesis, averaged.

$$chrF\beta = (1+\beta^2)\frac{chrP \cdot chrP}{\beta^2 \cdot chrP + chrR} \tag{81}$$

- **Embedding similarity**, which is more robust against synonyms or paraphrases.

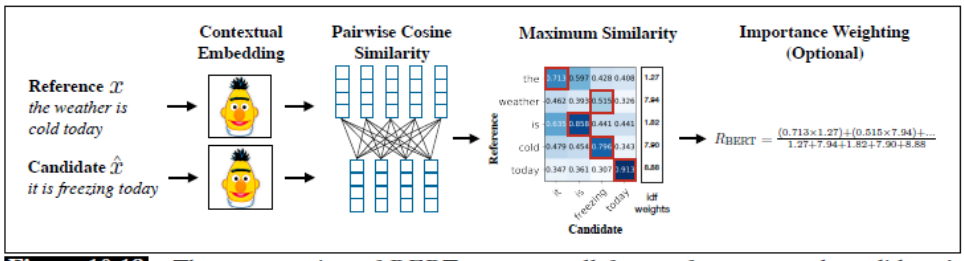
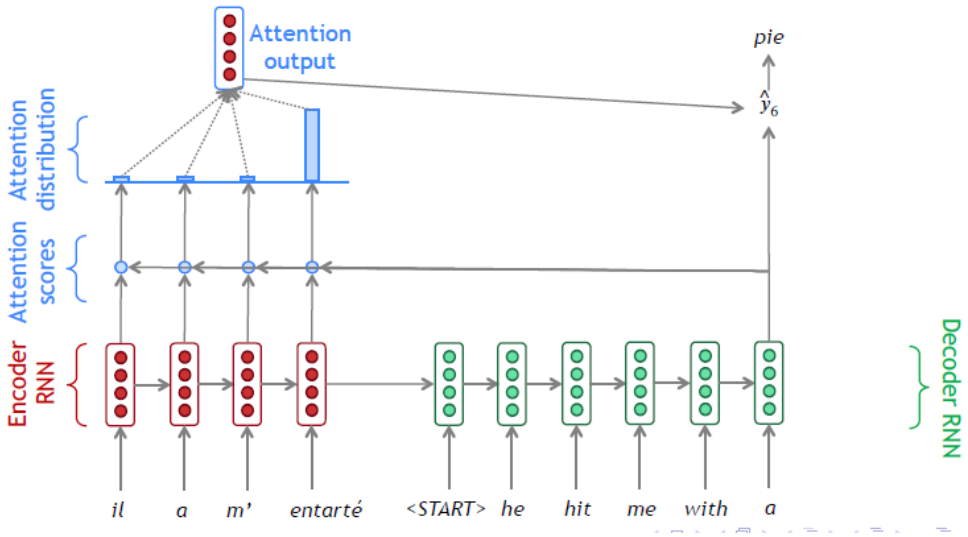


Figure 10.18 The computation of BERTSCORE recall from reference x and candidate $\hat{x}$, from Figure 1 in Zhang et al. (2020). This version shows an extended version of the metric in which tokens are also weighted by their idf values.

11.3 Attention

On each step of the decoder, one could use a direct connection to the encoder to focus on a particular part of the source sequence:



In particular:

- The encoder hidden states are $h_1 \dots h_N \in \mathbb{R}^h$.
- The decoder hidden state at step t is s_t.
- The **attention scores** at step t are computed as $e^t = [s_t^{\mathrm{T}} h_1, \dots, s_t^{\mathrm{T}} h_N] \in \mathbb{R}^N$.
- The **attention distribution** α^t used to take a weighted sum of the encoder hidden states is computed as $\alpha^t = \mathrm{softmax}(e^t) \in \mathbb{R}^N$.
- The **attention output**, which mostly contains information from the hidden states the received high attention, is computed as $a_t = \sum_{i=1}^{N} \alpha_i^t h_i \in \mathbb{R}^h$.
- The attention output and the decoder hidden state are concatenated $[a_t; s_t] \in \mathbb{R}^{2h}$.

12 Information Extraction and Question Answering

12.1 Information Extraction

Information extraction is about extracting limited kinds of semantic content from text: named entity recognition, relation extraction, event extraction, template filling. In particular:

- **Named entity recognition** is about finding each mention of entities (e.g. people, places, etc.). For example:

Type	Tag	Sample Categories	Example sentences
People	PER	people, characters	**Turing** is a giant of computer science.
Organization	ORG	companies, sports teams	The **IPCC** warned about the cyclone.
Location	LOC	regions, mountains, seas	The **Mt. Sanitas** loop is in **Sunshine Canyon**.
Geo-Political Entity	GPE	countries, states, provinces	**Palo Alto** is raising the fees for parking.
Facility	FAC	bridges, buildings, airports	Consider the **Golden Gate Bridge**.
Vehicles	VEH	planes, trains, automobiles	It was a classic **Ford Falcon**.

Figure 18.1 A list of generic named entity types with the kinds of entities they refer to.

  In particular, given a corpus of text, one has to find proper spans of text that constitute proper names and then classify the type of entity. Named entity recognition can be seen as a sequence labelling task which can be solved using three main approaches: feature-based (HMM, CRF, etc.), neural based (LSTMs), and rule-based.

- **Relation extraction** is about discerning relationships among detected entities. For example:

Relations	Types	Examples
Physical-Located	PER-GPE	**He** was in **Tennessee**
Part-Whole-Subsidiary	ORG-ORG	**XYZ**, the parent company of **ABC**
Person-Social-Family	PER-PER	**Yoko**'s husband **John**
Org-AFF-Founder	PER-ORG	**Steve Jobs**, co-founder of **Apple**...

Figure 18.10 Semantic relations with examples and the named entity types they involve.

  Relations correspond to sets of tuples over elements of a domain. Relation extraction can be solves either using a rule-based approach, or a machine learning approach.

12.2 Question Answering

Question answering focuses on questions that can be answered with simple facts expressed in short texts (factoid questions). There are two paradigms to implement question answering: an IR-based paradigm, which finds relevant documents then uses reading comprehension to read and draw and answer, and a knowledge-based paradigm, which builds a semantic (namely, logic) repersentation of the query, then queries a database of facts. In particular:

- **Knowledge-based** question answering focuses on semantic parsing to find a mapping between a text string and its logical form.
- **IR-based** question answering focuses on answering a user's question by finding short text segments from a document collection. The main steps that are involved are: question processing, document and passage retrieval, answer extraction.

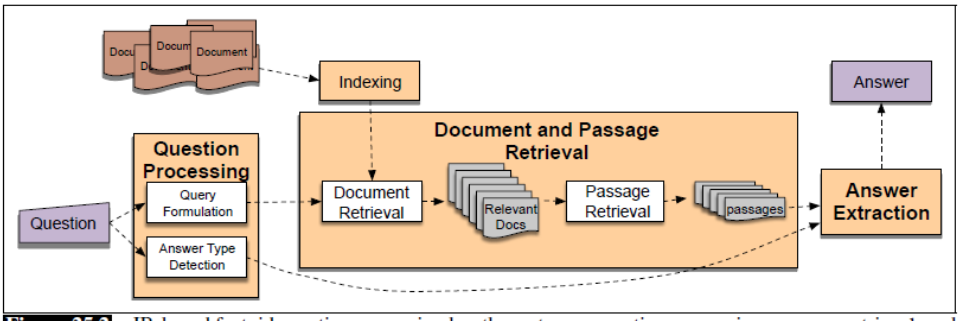


Figure 25.2 IR-based factoid question answering has three stages: question processing, passage retrieval, and answer processing.

  - The goal of **question processing** is to extract the query (i.e. a list of keywords for matching documents) and, possibly, the answer type, the focus and the question type.
  - **Document and passage retrieval** is about returning a ranked set of documents, and dividing these documents into smaller passages such as sections, paragraphs or sentences.
  - **Answer extraction** is a span labelling task: given a text, one would like to find the span of text which constitutes an answer. This type of task can be solved via supervised learning, either feature-based or neural-based. **Feature-based** answer extraction can be carried out using hand-written regular expressions:

Pattern	Question	Answer
<AP> such as <QP>	What is autism?	", developmental disorders such as autism"
<QP>, a <AP>	What is a caldera?	"the Long Valley caldera, a volcanic crater 19 miles long"

Figure 25.5 Some answer-extraction patterns using the answer phrase (AP) and question phrase (QP) for definition questions (Pasca, 2003).

Neural-based answer extraction, on the other hand, is based on neural networks. This approach works by producing good word embeddings and sentence encodings enabling to select the correct passage spans.

13 CNNs, Transformers and Contextual Word Embeddings

13.1 CNNs for NLP

Convolutional neural networks can be applied to text. Indeed, the idea behind 1D convolution is to compute vectors for every possible word subsequence of a certain length. For example, for *tentative deal reached to keep government open*, one computes the vectors for *tentative deal reached*, *deal reached to*, *reached to keep*, *to keep government*, *keep government open*:

∅	0.0	0.0	0.0	0.0
tentative	0.2	0.1	−0.3	0.4
deal	0.5	0.2	−0.3	−0.1
reached	−0.1	−0.3	−0.2	0.4
to	0.3	−0.3	0.1	0.1
keep	0.2	−0.3	0.4	0.2
government	0.1	0.2	−0.1	−0.1
open	−0.4	−0.4	0.2	0.3
∅	0.0	0.0	0.0	0.0

*

First kernel:

3	1	2	−3
−1	2	1	−3
1	1	−1	1

Second kernel:

1	0	0	1
1	0	−1	−1
0	1	0	1

Third kernel:

1	−1	2	−1
1	0	−1	3
0	2	2	1

=

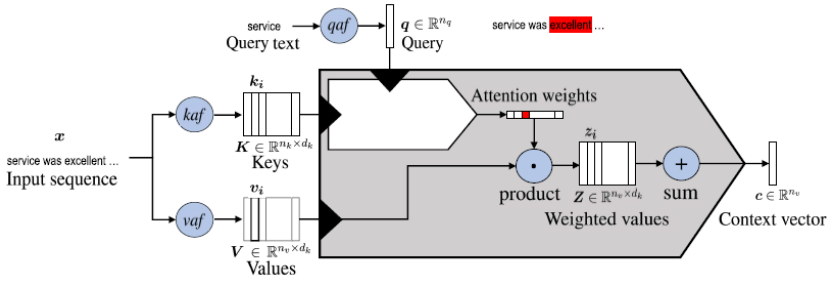
∅,t,d	−0.6	0.2	1.4
t,d,r	−1.0	1.6	−1.0
d,r,t	−0.5	−0.1	0.8
r,t,k	−3.6	0.3	0.3
t,k,g	−0.2	0.1	1.2
k,g,o	0.3	0.6	0.9
g,o,∅	−0.5	−0.9	0.1

max p	0.3	1.6	1.4

In general, CNNs are good for sentence classification but they are somewhat implausible/hard to interpret. Moreover, they are easy to parallelize on GPUs.

13.2 Transformers

Transformers have been introduced to achieve easy parallelization, like CNNs, and maintain the same sequence-to-sequence nature of RNNs. The attention mechanism in transformers uses the concepts of queries, keys and values, which are derived from the input sentence.



The attention mechanism comes in many flavours:

- Often K and V are linear projections of the same input embeddings.
- Q, K and V could also be all projections of the input using different embedding matrices (**self-attention**).
- The combination of Q and K can be a simple dot-product (**dot-product attention**).
- Different projections of Q, K and V can be used to yield multiple outputs (**multi-head attention**).

Transformers were originally introduced in 2017, were the authors proposed to apply a specific version of attention: the scaled dot-product attention:

$$\text{Attention}(Q, K, V) = \text{softmax}\left(\frac{QK^{\text{T}}}{\sqrt{d_k}}\right)V \tag{82}$$

where a scaling factor is used to prevent extremely small gradients. Moreover, masking can be used to inhibit leftward information flaw. The original paper also proposed to use multi-head, scaled dot-product attention. Q, K and V are projected h times with different, learned linear projections:

$$\text{MultiHead}(Q, K, V) = \text{concat}(head_1, \ldots, head_h)W^Q \tag{83}$$

where:

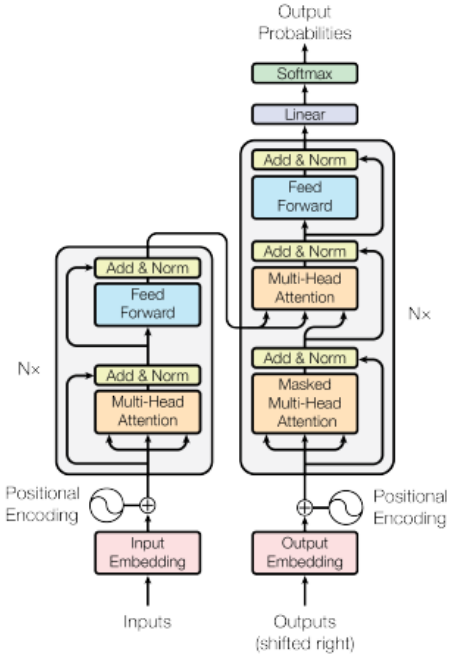
$$head_i = \text{Attention}(QW_i^Q, KW_i^K, VW_i^V) \tag{84}$$

A transformer is an RNN-free, encoder-decoder architecture entirely based on attention:

- The encoder is a stack of N layers, where each layer is composed of two sub-layers:
  - A multi-head self-attention mechanism.
  - A fully connected feed-forward network with ReLU activation, operating on each position independently.

  The encoder also uses residual connections, normalization, and positional encoding which makes up for the lack of sequential information processing.
- The decoder is also a stack of N layers, where each layer is composed of three sub-layers:
  - A masked multi-head self-attention layer.
  - A multi-head self-attention layer over the output of the encoder stack.
  - A feed-forward network.

  Masking is to ensure that predictions depend only on known outputs. Also the decoder uses positional encoding, residual connections and normalization.



Self-attention directly models relationships between all words in a sentence, regardless of their respective position. This enables learning long-range dependencies.

13.3 Contextual Word Representations

Contextual word vectors are vectors for words in context (word tokens). These vectors depends on arbitrarily long context of nearby words, and not on the context windows. Since RNNs produce context-specific word representations at each position, the idea here is to train a deep bidirectional RNN as language model, then use the context vectors as pre-trained word tokens. An example of such model is BERT.

14 Dialogue Systems and Chatbots

14.1 Dialogue Systems

Some Characteristics of human conversations are the following:

- Turn, namely each single contribution to the dialogue. A system has to know when to start/stop talking, and has to perform endpoint detection.
- Speech act, namely the action implied by a dialogue utterance.
- Grounding, namely establishing common ground.
- Dialogue structure, namely the structure followed by dialogue turns.
- Initiative, namely who controls the conversation.
- Implicature, namely a licensed inference (i.e. what inference is the hearer expected to draw? Usually, more information is communicated than present in uttered words).

In general, there exist two main types of dialogue systems: **chatbots**, which can carry on extended conversations with the goal of mimicking the unstructured conversations characteristics of the informal human-human interaction (e.g. ELIZA), and **task-oriented dialogue agents**, which use conversation with users to help complete tasks (e.g. Siri, Alexa, etc.).

14.2 Chatbots

The most important chatbot dialogue system in the field's history is ELIZA. ELIZA is designed to simulate a Rogerian psychologist (i.e. the system can assume the pose of knowing almost nothing about the world). This chatbot is a rule-based system. There exist also corpus-based chatbots, which are usually built using IR and machine-learned sequence transduction:

- **IR-based chatbots** respond to user's turn (query q) by repeating some appropriate turn (response r) from a corpus of natural text. In particular, there are two methods to do so:
  - Return the response to the most similar turn:

$$r = \text{response}\left(\underset{t \in C}{\text{argmax}} \frac{q^{\mathrm{T}} t}{\|q\|\|t\|}\right) \tag{85}$$

  - Return the most similar turn:

$$r = \underset{t \in C}{\text{argmax}} \frac{q^{\mathrm{T}} t}{\|q\|\|t\|} \tag{86}$$

  Returning the most similar turn works better in practice.

14.3 Task-Oriented Dialogue Agents

These agents are also called **frame-based dialogue systems**. A frame is a knowledge structure representing the kinds of intentions the system can extract from user sentences. A frame consists of a collection of slots, representing what the system needs to know. Each slot can take a set of possible values of a particular semantic type. A set of such frames is called **domain ontology**:

Slot	Type	Question Template
ORIGIN CITY	city	"From what city are you leaving?"
DESTINATION CITY	city	"Where are you going?"
DEPARTURE TIME	time	"When would you like to leave?"
DEPARTURE DATE	date	"What day would you like to leave?"
ARRIVAL TIME	time	"When do you want to arrive?"
ARRIVAL DATE	date	"What day would you like to arrive?"

Figure 26.9 A frame in a frame-based dialogue system, showing the type of each slot and a question used to fill the slot.

The system's goal is to fill the slots in the frame, then perform an action which can be relevant for the user. In particular, the system asks questions to the user, using question templates. If the user's response fills multiple slots, the system fills all relevant slots. Once enough information have been collected, the system can perform a necessary action and return the result. A common approach to slot-filling is rule-based, where the system consists of a large hand-designed semantic grammar (i.e. CFG where the left-hand side of each rule corresponds to semantic entities).

```
SHOW              → show me | i want | can i see|...
DEPART_TIME_RANGE → (after|around|before) HOUR |
                    morning | afternoon | evening
HOUR              → one|two|three|four...|twelve (AMPM)
FLIGHTS           → (a) flight | flights
AMPM              → am | pm
ORIGIN            → from CITY
DESTINATION       → to CITY
CITY              → Boston | San Francisco | Denver | Washington
```

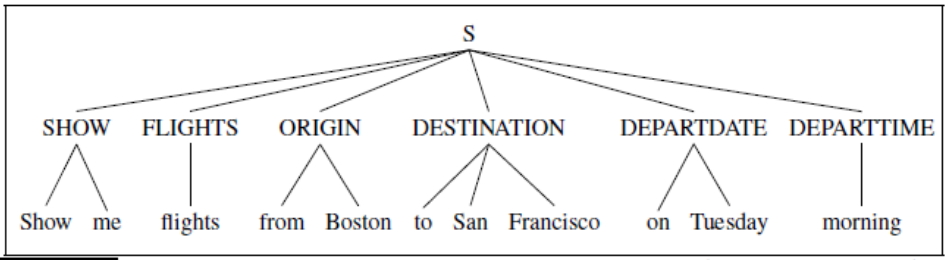


Figure 26.10 A semantic grammar parse for a user sentence, using slot names as the internal parse tree nodes.

More advanced architectures have also other components: natural language understanding, which helps extracting slot fillers using machine learning, dialogue state tracker, which maintains the current state

of dialogue, dialogue policy, which decides what to do or say next, and natural language generation, which is conditioned on context to produce more natural turns.

15 Argument Mining

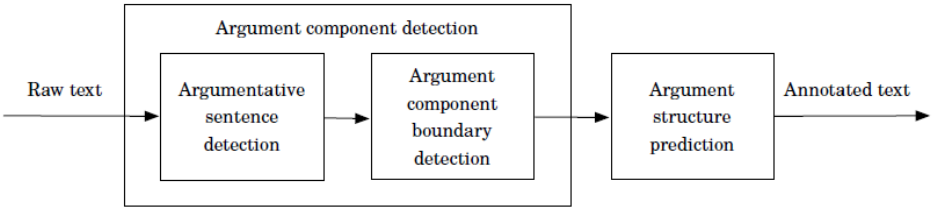
Argument mining is a sub-area of NLP that focuses on the automatic extraction of arguments from unstructured text. Natural arguments can be found in dialogues, legal documents, scientific literature, news article, user-generated Web discourse, etc. Argument mining can be used, for example, to move sentiment analysis a step forward, so to understand not only opinions, but also reasons behind them.

15.1 Data

Examples or argument mining sources are: Debatepedia (namely, a platform in which pros and cons of a specific topic are discussed), Wikipedia, annotations in datasets. The labels for such data are usually produced by humans annotators, and it is usually crucial to measure the quality of annotations. Indeed, the **inter-annotator agreement** is a measure of how well two annotators can make the same annotation decision for a certain category.

15.2 Methods

A typical argument mining problem can be divided into subsequent subtasks:



The most typical techniques used in argument mining can be applied to the following problems:

- Argument component detection/classification, which can be solved using statistical classifiers with hand-crafted features, or deep neural networks.
- Prediction of the structure of argument graphs, which can be solved using statistical classifiers with hand-crafted features, deep neural networks, symbolic approaches, or structured output machine learning methods (i.e. machine learning methods that use constraints amongst argument components).