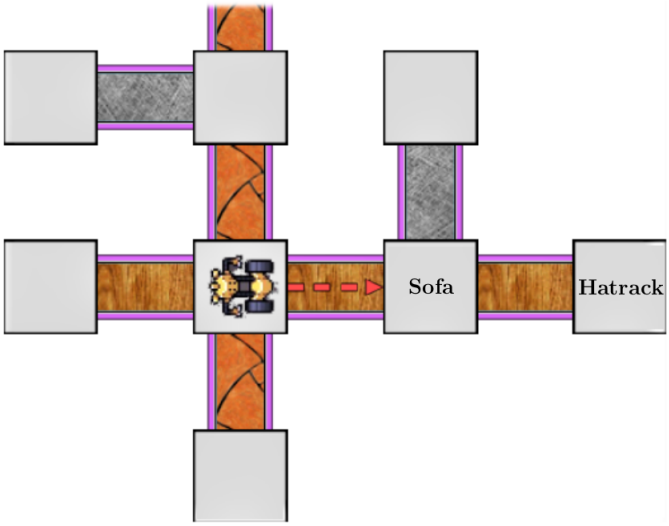
Text normalization, units, and edit distance

Herman Kamper

Robotic instruction following



We want to instruct the robot with natural language commands like:[1]

1. `Move forward to the chair and then turn left.`
2. `Go 2 spaces and check if there is a coat stand.`

The dumbest NLP solution

- Collect a dataset of (X, y) pairs, with $X = x_{1:T}$ an input instruction and y the command that the robot should execute.
- Gives a set of training pairs: $\{(X^{(n)}, y^{(n)})\}_{n=1}^{N}$
- For a new input, we find the closest input in the training set and predict its label y, i.e. one-nearest neighbour classification.

[1] Figure by Ryan Eloff.

To do this we need to figure out:

- How should we preprocess the data?
  - Should we map 2 to `two` or the other way around in input sentences?
  - Is punctuation important?
  - Can we lowercase everything?
- What is our basic modelling unit?
  - What is each x_t in the input sequence $x_{1:T}$?
  - Words, characters or maybe something in between?
- How do we measure the distance between two sequences?
  - Is the input word `forward` more similar to `forwds` or to `reverse`?

Text normalization

Text normalization is the task of converting text to a consistent, standardized form appropriate for the task at hand.

Normalization rules have to deal with things like:

- Punctuation: Can often remove, but sometimes useful, e.g. full stops indicating the end of a sentence. Need to decide whether we keep punctuation in abbreviations, numbers, dates, hyphenated words, etc. (also below).
- Abbreviations: `m.p.h.` or `miles per hour`? `Ph.D.` or `PhD`? `USA` or `US`?
- Numbers and prices: `R55,000.50` or `R55 000.50` or `R55000.50` or `R55000,50` or `fifty-five-thousand rand and fifty cents`?
- Dates: `13/04/22` or `13 April 2022`?
- URLs (`http://www.kamperh.com`), email addresses (`someone@ed.ac.uk`), hashtags (`#nlproc`), and emoticons `:)`
- Clitic contractions: `what're` or `what are` or `what 're`?
- Casing: Might want to lowercase everything, but could also be useful to keep capitalization, e.g. to identify names like `Stellenbosch` or `San Francisco`.
- Multi-word expressions: `New York-based`, `$37-a-share`

Rules are often implemented with regular expressions (J&M3, Sec. 2.1).

Task-specific

Unfortunately there isn't a single set of text normalization rules that can be applied across all settings. The normalization scheme should match your specific task. There are some standards, e.g. the Penn Treebank tokenization or NIST's tools for ASR text normalization, but you can't apply these blindly without regard to the specific NLP task.

Tokenization and units

Tokenization involves breaking up the input stream into the units we will model, i.e. constructing $x_{1:T}$ from the (normalized) input.

Say we have the following input:

```
turn right at the door
```

One obvious tokenization would be to just split at spaces between words:

```
["turn", "right", "at", "the", "door"]
```

But we could also just use characters as our modelling unit:

```
["t", "u", "r", "n", " ", "r", "i", "g", "h", ...]
```

Let's first talk about words and then return to the question of which units to use.

Words

Word tokens and types

- Word types: The set of unique words.
- Word token: An instance/occurrence of a word of a specific type. You can have multiple word tokens of the same type occurring in a text.

How many word types and tokens are in this sentence?

```
a cat and a brown dog chases a black dog
```

Word counts

In 10k sentences from the English Wikipedia dump, there are 194 207 word tokens from 24 183 types. The most frequent words are listed below.

Any word:

Count	Word	Rank
12 336	`the`	1
7384	`of`	2
6561	`and`	3
4655	`in`	4
4305	`to`	5
3322	`a`	6
1959	`is`	7
1743	`as`	8
1627	`The`	9
1483	`that`	10

Nouns:

Count	Word
419	Apollo
379	state
276	Lincoln
240	Alaska
231	time
230	Agassi
215	Alabama
179	century
170	use
153	anthropology

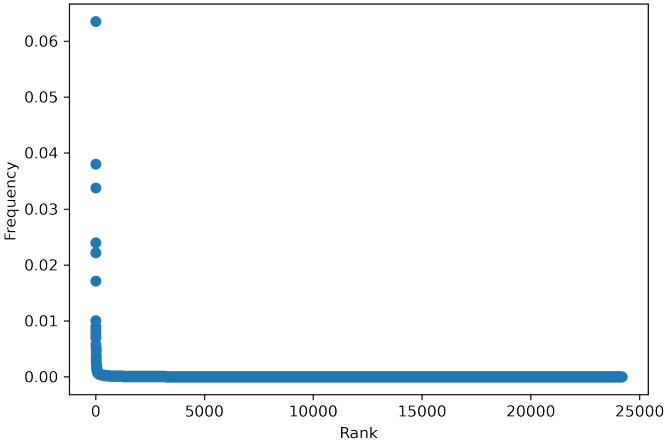
Zipf's law

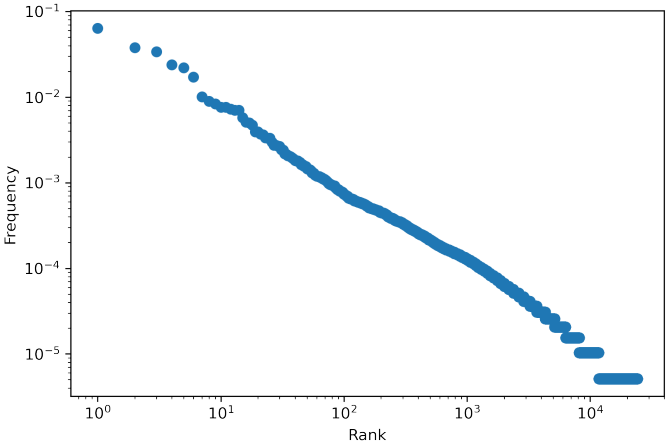
The frequency of a word is inversely proportional to its rank:

$$f \approx \frac{c}{r}$$

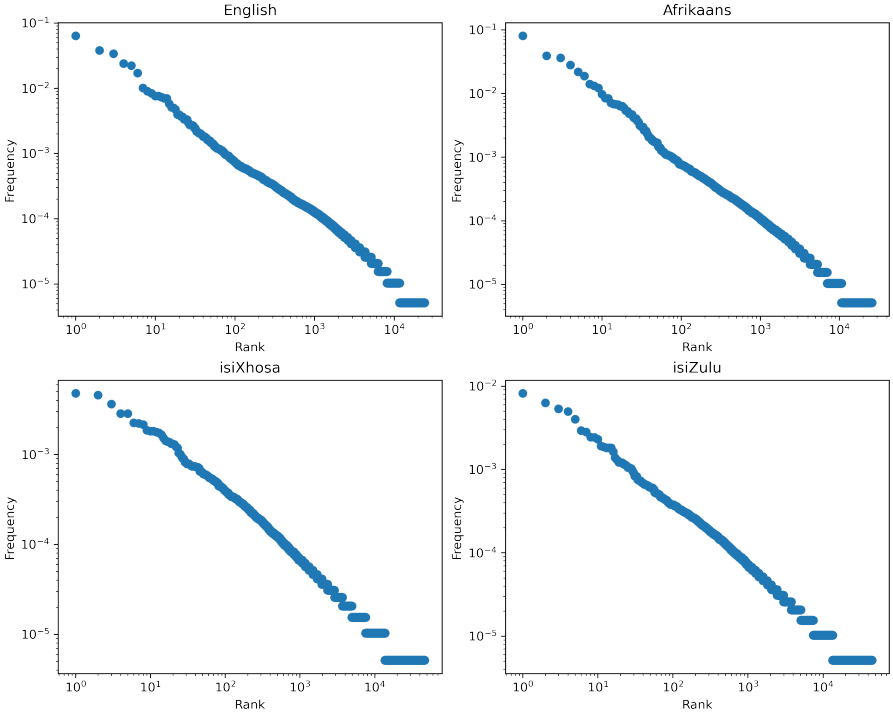
where f is the word frequency, r the rank, and c some constant.

Linear on a plot with the frequency and rank axes on $\log$ scale (why?). On English:





An empirical law that holds for all languages. On different languages from the Wikipedia dumps:



NLP challenge: Sparsity

Most words will be used infrequently, but we still need to be able to deal with them. This is a challenge when using machine learning, since your training data might only have one or two occurrences of most words (or maybe even zero!), but we still need to handle these words at test time.

What's so special about words?

You might think that the idea of a "word" is pretty obvious. But this is not really the case.

English: How many words are in `didn't`, `New York`, `high-risk`, `@stellenboschuniversity`?

Afrikaans and German: Use agglutination to form words.

- `satellietnavigasiestelsels` (af)
- `K-gemiddeldestrosvormingalgoritme` (af)
- `computerlinguistikvorlesung` (de)

isiZulu: Morphologically rich.

- `wukutholakala` = `wu+u+ku+thol+akal+a`

Chinese: Written without spaces. Characters in Chinese are meaning-bearing units (normally morphemes) that combine into words, but there isn't a standard definition of what a word is.

姚明进入总决赛
Yao Ming reaches the finals

姚明 进入 总决赛
YaoMing reaches finals

姚 明 进入 总 决赛
Yao Ming reaches overall finals

姚 明 进 入 总 决 赛
Yao Ming enter enter overall decision game

So which units should we use in our NLP system?

- We have looked at using words or characters as our units.
- With words we have sparsity problems but have a lot of structure (from a single word you get meaning).
- With characters, we loose structure (a single Latin-script character bears no meaning) but we don't suffer from sparsity (why?).
- But maybe there is something in between these two extremes where we still have some structure but less sparsity?

Subword units: Morphology

Morpheme: The smallest meaning-bearing unit of a language, e.g.

`unlikeliest` = `un`+`likely`+`est`

Morphology: Study of how words are built up from morphemes, e.g.

`de`+`salin`+`ate`+`ion` and not `ate`+`salin`+`ion`+`de`

In some languages, morphology matters a lot:

```
ru: zhenshina devochke dala knigu
en: the woman gave the girl a book

ru: zhenshine devochka dala knigu
en: the girl gave the woman a book
```

Types of morphemes

- Stems: Central morpheme giving a word its main meaning, e.g. `fox`, `cat`, `small`, `walk`.
- Affixes: Added on to give additional meaning, e.g.
  - `+s`, `+ed`
  - `un+`
  - `+bloody+`

Stems vs lemmas

- Lemma: The canonical form (dictionary form) of a set of words.
  - `am`, `are`, `is` have the lemma `be`.
  - `fly`, `flies`, `flew`, `flying` have the lemma `fly`.
  - `walk`, `walks`, `walked`, `walking` have the lemma `walk`.
  - `walker`, `walkers` have the lemma `walker`.
- Stem: The part of the word that is common to all its variants (but there are also other definitions).
  - `produce`, `production` have the stem `produc`.
  - `walk`, `walks`, `walked`, `walking`, `walker`, `walkers` have the stem `walk`.
  - Do `fly`, `flies`, `flew`, `flying` have a common stem `fl`? Or maybe only `fly` and `flying` share the stem `fly`? Decision will depend on the application.
- Lemmatization and stemming are both NLP tasks.
  - Porter stemmer: Rule-based and fast but a bit crude (J&M3, Sec. 2.4.4).

Byte-pair encoding (BPE)

Instead of morphemes, subword units can also be induced automatically from a text corpus. There are a several approaches including SentencePiece and byte-pair encoding (BPE).

We will look at BPE (probably one of the most popular). BPE first learns the units and then applies it to new data.

Token learner algorithm

- Initialization:
  - Vocabulary $\mathcal{V} \leftarrow$ unique characters in text.
  - Tokenize text into separate characters.
- for iteration $i = 1$ to K:
  - Find most frequent pair of adjacent tokens: t_L, t_R
  - Merge tokens: $t_{\text{new}} = t_L t_R$
  - Add to vocabulary: $\mathcal{V} \leftarrow \mathcal{V} \cup \{t_{\text{new}}\}$
  - Replace all occurrences of t_L, t_R in text with t_{new}.

Token segmenter

Apply on the data the merges we learned on the training data in the order we learned them.

Neural Machine Translation of Rare Words with Subword Units

Rico Sennrich and **Barry Haddow** and **Alexandra Birch**
School of Informatics, University of Edinburgh
{rico.sennrich,a.birch}@ed.ac.uk, bhaddow@inf.ed.ac.uk

BPE example

Corpus:

```
low low low low low
lower lower
newer newer newer newer newer newer
wider wider wider
new new new
```

The corpus has 19 word tokens from five word types.

- iteration $i = 1$:
  - Find most frequent pair of adjacent tokens: t_L, t_R
  - Merge tokens: $t_{\text{new}} = t_L t_R$
  - Add to vocabulary: $\mathcal{V} \leftarrow \mathcal{V} \cup \{t_{\text{new}}\}$
  - Replace all occurrences of t_L, t_R in text with t_{new}.

```
l o w _ l o w _ l o w _ l o w _ l o w _
l o w e r _ l o w e r _
n e w e r _ n e w e r _ n e w e r _ n e w e r _ n e w e r _
n e w e r _
w i d e r _ w i d e r _ w i d e r _
n e w _ n e w _ n e w _
```

- iteration $i = 2$:
  - Find most frequent pair of adjacent tokens: t_L, t_R
  - Merge tokens: $t_{\text{new}} = t_L t_R$
  - Add to vocabulary: $\mathcal{V} \leftarrow \mathcal{V} \cup \{t_{\text{new}}\}$
  - Replace all occurrences of t_L, t_R in text with t_{new}.

```
l o w _ l o w _ l o w _ l o w _ l o w _
l o w er _ l o w er _
n e w er _ n e w er _ n e w er _ n e w er _ n e w er _
n e w er _
w i d er _ w i d er _ w i d er _
n e w _ n e w _ n e w _
```

- iteration $i = 3$:
  - Find most frequent pair of adjacent tokens: t_L, t_R
  - Merge tokens: $t_{\text{new}} = t_L t_R$
  - Add to vocabulary: $\mathcal{V} \leftarrow \mathcal{V} \cup \{t_{\text{new}}\}$
  - Replace all occurrences of t_L, t_R in text with t_{new}.

```
l o w _ l o w _ l o w _ l o w _ l o w _
l o w er_ l o w er_
n e w er_ n e w er_ n e w er_ n e w er_ n e w er_
n e w er_
w i d er_ w i d er_ w i d er_
n e w _ n e w _ n e w _
```

```
Iteration: 1
5: l o w </w>
2: l o w e r </w>
6: n e w e r </w>
3: w i d e r </w>
3: n e w </w>
Merge: ("e", "r")

Iteration: 2
5: l o w </w>
2: l o w er </w>
6: n e w er </w>
3: w i d er </w>
3: n e w </w>
Merge: ("er", "</w>")

Iteration: 3
5: l o w </w>
2: l o w er</w>
6: n e w er</w>
3: w i d er</w>
3: n e w </w>
Merge: ("n", "e")

...

Iteration: 7
5: low </w>
2: low er</w>
6: new er</w>
3: w i d er</w>
3: new </w>
Merge: ("new", "er</w>")
```

Edit distance

Given two symbolic sequences, how similar are they?

The edit distance is the minimum number of (weighted) changes needed to convert one sequence to the other. Also called the Levenshtein distance.

Edit distance between `stall` and `table`:

```
s t a l l
  t a l l    # deletion
  t a b l    # substitution
  t a b l e  # insertion
```

The edit distance between `stall` and `table` is therefore 3, if we let deletions, substitutions and insertions all cost 1.

One way to determine the edit distance is to find the best alignment between the sequences.

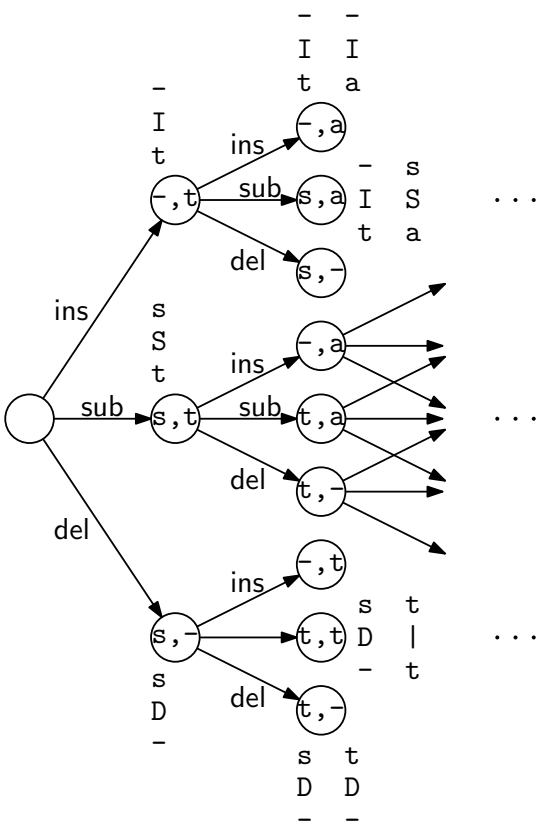
Example: Alignment costs between `stall` and `table`

Let $w_{\text{del}} = w_{\text{ins}} = 1$ and $w_{\text{sub}} = 2$. (We often use a higher penalty for substitutions. Why?) Below are example alignments between `stall` and `table` with their alignment costs. The edit distance is given by the optimal alignment with a cost of 4 in this case. There are a few optimal alignments in this case, all with a cost of 4.

```
s  t  a  -  l  -  l
S  D  |  I  |  I  D
t  -  a  b  l  e  -    # cost: 6

s  t  a  l  -  l  -
D  D  S  S  I  |  I
-  -  t  a  b  l  e    # cost: 8

s  t  a  l  l  -
D  |  |  S  |  I
-  t  a  b  l  e       # cost: 4

s  t  a  -  l  l
D  |  |  I  |  S
-  t  a  b  l  e       # cost: 4
```

Brute force alignment

Solution to calculating edit distance: Consider all possible alignments and pick the one with the lowest cost. But how many possible alignments are there?



Roughly $\mathcal{O}(3^N)$-ish. The number of alignments grow exponentially with the length of the sequences!

So rather than brute force, we use a dynamic programming algorithm: Break the problem down into simpler sub-problems and then solve recursively (Wikipedia). (The fudge does that mean?) Note that the algorithm below is guaranteed to give the optimal alignment: no approximations are used.

The edit distance algorithm

- Inputs: $x_{1:N}$ and $y_{1:M}$
- Cost matrix: $\mathbf{D} \in \mathbb{R}^{(N+1)\times(M+1)}$
- Initialization:

$$
\begin{aligned}
D_{0,0} &= 0 \\
D_{i,0} &= D_{i-1,0} + w_{\text{del}} \qquad \text{for } i = 1, 2, \ldots, N \\
D_{0,j} &= D_{0,j-1} + w_{\text{ins}} \qquad \text{for } j = 1, 2, \ldots, M
\end{aligned}
$$

- Recursion:

$$
D_{i,j} = \begin{cases} D_{i-1,j-1} & \text{if } x_i = y_j \\ \min \begin{cases} D_{i-1,j} + w_{\text{del}} \\ D_{i,j-1} + w_{\text{ins}} \\ D_{i-1,j-1} + w_{\text{sub}} \end{cases} & \text{if } x_i \neq y_j \end{cases}
$$

for $i = 1, 2, \ldots, M$ and $j = 1, 2, \ldots, N$

- Backtracking: From $D_{N,M}$ to $D_{0,0}$

Example: Align `stall` to `table`

$w_{\text{del}} = 1$, $w_{\text{ins}} = 1$ and $w_{\text{sub}} = 2$

		t	a	b	l	e
s						
t						
a						
l						
l						

		t	a	b	l	e
	0	$\leftarrow 1$	$\leftarrow 2$	$\leftarrow 3$	$\leftarrow 4$	$\leftarrow 5$
s	$\uparrow 1$	$\leftarrow\nwarrow\uparrow 2$	$\leftarrow\nwarrow\uparrow 3$	$\leftarrow\nwarrow\uparrow 4$	$\leftarrow\nwarrow\uparrow 5$	$\leftarrow\nwarrow\uparrow 6$
t	$\uparrow 2$	$\nwarrow 1$	$\leftarrow 2$	$\leftarrow 3$	$\leftarrow 4$	$\leftarrow 5$
a	$\uparrow 3$	$\uparrow 2$	$\nwarrow 1$	$\leftarrow 2$	$\leftarrow 3$	$\leftarrow 4$
l	$\uparrow 4$	$\uparrow 3$	$\uparrow 2$	$\leftarrow\nwarrow\uparrow 3$	$\nwarrow 2$	$\leftarrow 3$
l	$\uparrow 5$	$\uparrow 4$	$\uparrow 3$	$\leftarrow\nwarrow\uparrow 4$	$\nwarrow\uparrow 3$	$\leftarrow\nwarrow\uparrow 4$

Time complexity: Have to calculate $\mathcal{O}(NM)$ values in the cost matrix.

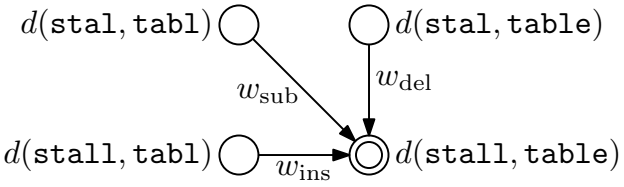
Why is this guaranteed to give the optimal alignment?

Each cell $D_{i,j}$ is the smallest cost to align $x_{1:i}$ to $y_{1:j}$. Why?

Let's say I knew the edit distances:

- $d(\texttt{stall}, \texttt{tabl})$
- $d(\texttt{stal}, \texttt{table})$
- $d(\texttt{stal}, \texttt{tabl})$

Then I could get the edit distance of $d(\texttt{stall}, \texttt{table})$. Note, this would be exact, without any approximation!



This also holds for the non-final steps. The edit distance algorithm uses this as its smallest subproblem. The solution to each subproblem is stored, and then used to solve and store the solutions to larger subproblems until we get the final solution (the best overall alignment).

More on the edit distance

Advanced edit distance

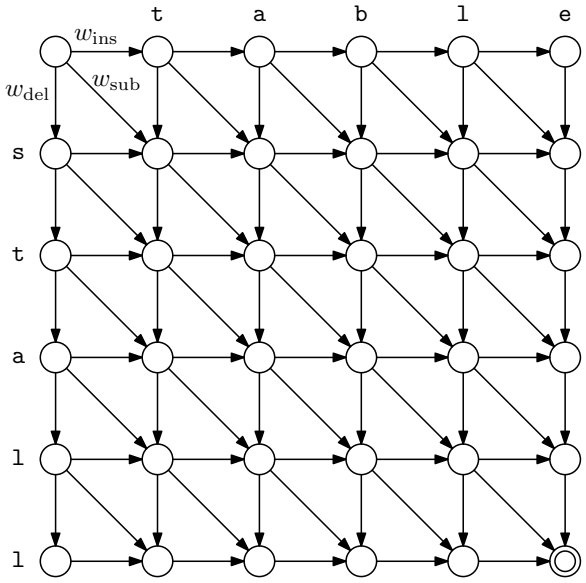
- Can give a different weight w depending on the symbols involved.
- Local alignment variants: Find subsequences that align well.
- Dynamic time warping: Align continuous signals.

Applications of edit distance

- Computational biology: Aligning sequences of nucleotides.
- Spell correction
- Speech recognition: Calculating word error rate.

Dynamic programming in general

The edit distance algorithm is an instance of dynamic programming—a more general class of algorithms. Dynamic programming can always be reduced to finding the optimal path through a directed acyclic graph. The graph in this case:



Videos covered in this note

- Edit distance (20 min)

Further reading

I have a separate note that describes dynamic programming in the more general sense.

Acknowledgements

Some of the content in this note is based on:

- The NLP course of Jan Buys at the University of Cape Town.
- The NLP course of Sharon Goldwater at the University of Edinburgh.