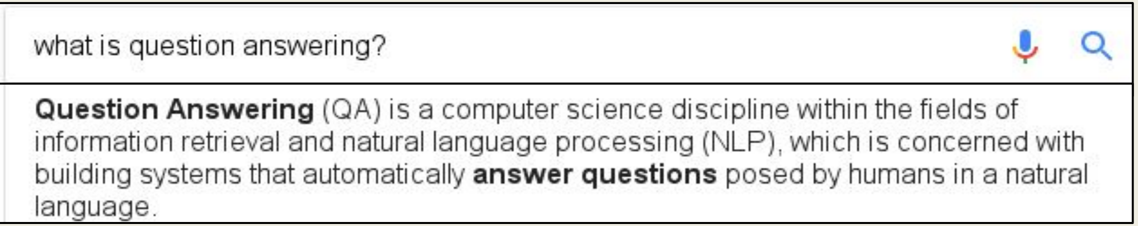
Language Models

Human Language from a Computational Perspective
April 11, 2018

Natural Language Processing

Algorithms that understand or generate human language



Automatic summarization is the process of reducing a text document with a computer program in order to create a summary that retains the most important points of the original document. Technologies that can make a coherent summary take into account variables such as length, writing style and syntax. Automatic data summarization is part of machine learning and data mining. The main idea of summarization is to find a representative subset of the data, which contains the *information* of the entire set. Summarization technologies are used in a large number of sectors in industry today. An example of the use of summarization technology is search engines such as Google. Other examples include document summarization, image collection summarization and video summarization. Document summarization, tries to automatically create a *representative summary* or *abstract* of the entire document, by finding the most *informative* sentences. Similarly, in image summarization the system finds the most representative and important (or salient) images. Similarly, in consumer videos one would want to remove the boring or repetitive scenes, and extract out a much shorter and concise version of the video.

Automatic summarization: reducing text with a computer to retain the most important points.

Statistical Language Model

How likely is each of these sentences?

PLEASE MAKE ME A CUP OF <u>COFFEE</u>

PLEASE MAKE ME A CUP OF <u>BUTTER</u>

PLEASE MAKE ME A CUP OF <u>BOTTLE</u>

PLEASE MAKE ME A CUP OF <u>DREAM</u>

PLEASE MAKE ME A CUP OF <u>PLEASE</u>

Uses of Language Models

- Typing prediction
- Spelling correction
- Speech recognition
- Many more

Algorithm

Instructions for manipulating data.

Can get parameters as **input**.

Returns an **output**.

Pseudocode

Notation to describe algorithms.

Not a programming language, but clear enough for humans.

Algorithm to find maximum

Find the largest number in a list.

$$[3, 1, 4, 16, 0, 2] \longrightarrow 16$$

$$[1, 2, 1, 1, 1] \longrightarrow 2$$

$$[-3, -2, 0, -1] \longrightarrow 0$$

Algorithm to find maximum

```
max(L):
    m ← L[1]
    i ← 2
    while i ≤ len(L):
        if L[i] > m:
            m ← L[i]
        i ← i + 1
    return m
```

- ▹ L is a list of numbers
- ▹ assign the first number to m
- ▹ assign 2 to i
- ▹ repeat while i is at most len(L)
- ▹ the i’th number is larger than m
- ▹ assign the i’th number to m
- ▹ increase i by 1
- ▹ output is the value of m

Algorithm to find maximum

Comments

```
max(L):
    m ← L[1]
    i ← 2
    while i ≤ len(L):
        if L[i] > m:
            m ← L[i]
        i ← i + 1
    return m
```

- ▹ L is a list of numbers
- ▹ assign the first number to m
- ▹ assign 2 to i
- ▹ repeat while i is at most len(L)
- ▹ the i’th number is larger than m
- ▹ assign the i’th number to m
- ▹ increase i by 1
- ▹ output is the value of m

Algorithm to find maximum

Function definition

```
max(L):                       ▷ L is a list of numbers
    m ← L[1]                  ▷ assign the first number to m
    i ← 2                     ▷ assign 2 to i
    while i ≤ len(L):         ▷ repeat while i is at most len(L)
        if L[i] > m:          ▷ the i'th number is larger than m
            m ← L[i]          ▷ assign the i'th number to m
        i ← i + 1             ▷ increase i by 1
    return m                  ▷ output is the value of m
```

Algorithm to find maximum

max(L): **Parameters**

m ← L[1]	▷ L is a list of numbers
	▷ assign the first number to m
i ← 2	▷ assign 2 to i
while i ≤ len(L):	▷ repeat while i is at most len(L)
if L[i] > m:	▷ the i'th number is larger than m
m ← L[i]	▷ assign the i'th number to m

A function can get more than one parameter, but **max** gets just one

Algorithm to find maximum

```
max(L):                         ▷ L is a list of numbers
    m ← L[1]                    ▷ assign the first number to m
    i ← 2  Variables            ▷ assign 2 to i
    while i ≤ len(L):           ▷ repeat while i is at most len(L)
        if L[i] > m:            ▷ the i’th number is larger than m
            m ← L[i]            ▷ assign the i’th number to m
        i ← i + 1               ▷ increase i by 1
    return m                    ▷ output is the value of m
```

Algorithm to find maximum

```
max(L):                         ▷ L is a list of numbers
    m ← L[1]                    ▷ assign the first number to m
    i ← 2   Assignment          ▷ assign 2 to i
    while i ≤ len(L):           ▷ repeat while i is at most len(L)
        if L[i] > m:            ▷ the i'th number is larger than m
            m ← L[i]            ▷ assign the i'th number to m
        i ← i + 1               ▷ increase i by 1
    return m                    ▷ output is the value of m
```

Algorithm to find maximum

```
max(L):                       ▷ L is a list of numbers
    m ← L[1]  Indexing        ▷ assign the first number to m
    i ← 2                     ▷ assign 2 to i
    while i ≤ len(L):         ▷ repeat while i is at most len(L)
        if L[i] > m:          ▷ the i'th number is larger than m
            m ← L[i]          ▷ assign the i'th number to m
        i ← i + 1             ▷ increase i by 1
    return m                  ▷ output is the value of m
```

Algorithm to find maximum

```
max(L):                         ▷ L is a list of numbers
    m ← L[1]                    ▷ assign the first number to m
    i ← 2                       ▷ assign 2 to i
    while i ≤ len(L):           ▷ repeat while i is at most len(L)
        if L[i] > m:            ▷ the i'th number is larger than m
            m ← L[i]            ▷ assign the i'th number to m
```

Function call

The function **len** returns the number of elements (length) of a list

Algorithm to find maximum

```
max(L):
    m ← L[1]                 ▷ L is a list of numbers
    i ← 2                    ▷ assign the first number to m
    while i ≤ len(L):        ▷ assign 2 to i
        if L[i] > m:         ▷ repeat while i is at most len(L)
            m ← L[i]         ▷ the i'th number is larger than m
        i ← i + 1            ▷ assign the i'th number to m
    return m                 ▷ increase i by 1
                             ▷ output is the value of m
```

Loop

Algorithm to find maximum

```
max(L):                      ▷ L is a list of numbers
    m ← L[1]                 ▷ assign the first number to m
    i ← 2                    ▷ assign 2 to i
    while i ≤ len(L):        ▷ repeat while i is at most len(L)
        if L[i] > m:         ▷ the i'th number is larger than m
            m ← L[i]         ▷ assign the i'th number to m
        i ← i + 1            ▷ increase i by 1
    return m                 ▷ output is the value of m
```

Condition

Algorithm to find maximum

```
max(L):
    m ← L[1]
    i ← 2
    while i ≤ len(L):
        if L[i] > m:
            m ← L[i]
        i ← i + 1
    return m
```

- ▷ L is a list of numbers
- ▷ assign the first number to m
- ▷ assign 2 to i
- ▷ repeat while i is at most len(L)
- ▷ the i'th number is larger than m
- ▷ assign the i'th number to m
- ▷ increase i by 1
- ▷ output is the value of m

Output

Algorithm to find maximum

```
max(L):  Indentation          ▷ L is a list of numbers
    m ← L[1]                  ▷ assign the first number to m
    i ← 2                     ▷ assign 2 to i
    while i ≤ len(L):         ▷ repeat while i is at most len(L)
        if L[i] > m:          ▷ the i'th number is larger than m
            m ← L[i]          ▷ assign the i'th number to m
        i ← i + 1             ▷ increase i by 1
    return m                  ▷ output is the value of m
```

Running the algorithm

```
max(L):
    m ← L[1]
    i ← 2
    while i ≤ len(L):
        if L[i] > m:
            m ← L[i]
        i ← i + 1
    return m
```

L = [3, 1, 4, 16, 0, 2]

m

i

Running the algorithm

```
max(L):
    m ← L[1]
    i ← 2
    while i ≤ len(L):
        if L[i] > m:
            m ← L[i]
        i ← i + 1
    return m
```

L = [3, 1, 4, 16, 0, 2]

m

i

Running the algorithm

```
max(L):
    m ← L[1]
    i ← 2
    while i ≤ len(L):
        if L[i] > m:
            m ← L[i]
        i ← i + 1
    return m
```

L = [3, 1, 4, 16, 0, 2]

m = 3

i

Running the algorithm

```
max(L):
    m ← L[1]
    i ← 2
    while i ≤ len(L):
        if L[i] > m:
            m ← L[i]
        i ← i + 1
    return m
```

$L = [3, 1, 4, 16, 0, 2]$

$m = 3$

$i = 2$

Running the algorithm

```
max(L):
    m ← L[1]
    i ← 2
    while i ≤ len(L):
        if L[i] > m:
            m ← L[i]
        i ← i + 1
    return m
```

L = [3, 1, 4, 16, 0, 2]

m = 3

i = 2

len(L) = 6

2 ≤ 6 ✓

Running the algorithm

max(L):
    m ← L[1]
    i ← 2
    while i ≤ len(L):
        if L[i] > m:
            m ← L[i]
        i ← i + 1
    return m

L = [3, 1, 4, 16, 0, 2]

m = 3

i = 2

L[i] = L[2] = 1

1 > 3 ✗

Running the algorithm

```
max(L):
    m ← L[1]
    i ← 2
    while i ≤ len(L):
        if L[i] > m:
            m ← L[i]
        i ← i + 1
    return m
```

L = [3, 1, 4, 16, 0, 2]

m = 3

i = 3

Running the algorithm

```
max(L):
    m ← L[1]
    i ← 2
    while i ≤ len(L):
        if L[i] > m:
            m ← L[i]
        i ← i + 1
    return m
```

$L = [3, 1, 4, 16, 0, 2]$

$m = 3$

$i = 3$

$\text{len}(L) = 6$

$3 \leq 6$ ✓

Running the algorithm

```
max(L):
    m ← L[1]
    i ← 2
    while i ≤ len(L):
        if L[i] > m:
            m ← L[i]
        i ← i + 1
    return m
```

$L = [3, 1, 4, 16, 0, 2]$

$m = 3$

$i = 3$

$L[i] = L[3] = 4$

$4 > 3$ ✓

Running the algorithm

```
max(L):
    m ← L[1]
    i ← 2
    while i ≤ len(L):
        if L[i] > m:
            m ← L[i]
        i ← i + 1
    return m
```

L = [3, 1, 4, 16, 0, 2]

m = 4

i = 3

L[i] = L[3] = 4

Running the algorithm

```
max(L):
    m ← L[1]
    i ← 2
    while i ≤ len(L):
        if L[i] > m:
            m ← L[i]
        i ← i + 1
    return m
```

L = [3, 1, 4, 16, 0, 2]

m = 4

i = 4

Running the algorithm

```
max(L):
    m ← L[1]
    i ← 2
    while i ≤ len(L):
        if L[i] > m:
            m ← L[i]
        i ← i + 1
    return m
```

$L = [3, 1, 4, 16, 0, 2]$

$m = 4$

$i = 4$

$\text{len}(L) = 6$

$4 \leq 6$ ✓

Running the algorithm

```
max(L):
    m ← L[1]
    i ← 2
    while i ≤ len(L):
        if L[i] > m:
            m ← L[i]
        i ← i + 1
    return m
```

$L = [3, 1, 4, 16, 0, 2]$

$m = 4$

$i = 4$

$L[i] = L[4] = 16$

$16 > 4$ ✓

Running the algorithm

```
max(L):
    m ← L[1]
    i ← 2
    while i ≤ len(L):
        if L[i] > m:
            m ← L[i]
        i ← i + 1
    return m
```

L = [3, 1, 4, 16, 0, 2]

m = 16

i = 4

L[i] = L[4] = 16

Running the algorithm

```
max(L):
    m ← L[1]
    i ← 2
    while i ≤ len(L):
        if L[i] > m:
            m ← L[i]
        i ← i + 1
    return m
```

L = [3, 1, 4, 16, 0, 2]

m = 16

i = 5

Running the algorithm

```
max(L):
    m ← L[1]
    i ← 2
    while i ≤ len(L):
        if L[i] > m:
            m ← L[i]
        i ← i + 1
    return m
```

$L = [3, 1, 4, 16, 0, 2]$

$m = 16$

$i = 5$

$\mathrm{len}(L) = 6$

$5 \leq 6$ ✓

Running the algorithm

```
max(L):
    m ← L[1]
    i ← 2
    while i ≤ len(L):
        if L[i] > m:
            m ← L[i]
        i ← i + 1
    return m
```

L = [3, 1, 4, 16, 0, 2]

m = 16

i = 5

L[i] = L[5] = 0

0 > 16 ✗

Running the algorithm

```
max(L):
    m ← L[1]
    i ← 2
    while i ≤ len(L):
        if L[i] > m:
            m ← L[i]
        i ← i + 1
    return m
```

L = [3, 1, 4, 16, 0, 2]

m = 16

i = 6

Running the algorithm

max(L):
    m ← L[1]
    i ← 2
    while i ≤ len(L):
        if L[i] > m:
            m ← L[i]
        i ← i + 1
    return m

L = [3, 1, 4, 16, 0, 2]

m = 16

i = 6

len(L) = 6

6 ≤ 6 ✓

Running the algorithm

```
max(L):
    m ← L[1]
    i ← 2
    while i ≤ len(L):
        if L[i] > m:
            m ← L[i]
        i ← i + 1
    return m
```

$L = [3, 1, 4, 16, 0, 2]$

$m = 16$

$i = 6$

$L[i] = L[6] = 2$

$2 > 16$

Running the algorithm

```
max(L):
    m ← L[1]
    i ← 2
    while i ≤ len(L):
        if L[i] > m:
            m ← L[i]
        i ← i + 1
    return m
```

L = [3, 1, 4, 16, 0, 2]

m = 16

i = 7

Running the algorithm

```
max(L):
    m ← L[1]
    i ← 2
    while i ≤ len(L):
        if L[i] > m:
            m ← L[i]
        i ← i + 1
    return m
```

$L = [3, 1, 4, 16, 0, 2]$

$m = 16$

$i = 7$

$len(L) = 6$

$7 \le 6$ ✗

Running the algorithm

```
max(L):
    m ← L[1]
    i ← 2
    while i ≤ len(L):
        if L[i] > m:
            m ← L[i]
        i ← i + 1
    return m
```

L = [3, 1, 4, 16, 0, 2]

m = 16

i = 7

output: 16

Finding index of maximum

Index of largest number in a list.

$[3, 1, 4, 16, 0, 2] \rightarrow 4$

$[1, 2, 1, 1, 1] \rightarrow 2$

$[-3, -2, 0, -1] \rightarrow 3$

Finding index of maximum

```
argmax(L):                    ▹ L is a list of numbers
    a ← 1                     ▹ index of the first element
    i ← 2                     ▹ index of the second element
    while i ≤ len(L):         ▹ repeat while i is at most len(L)
        if L[i] > L[a]:       ▹ i'th number is larger than a'th
            a ← i             ▹ assign i to a
        i ← i + 1             ▹ increase i by 1
    return a                  ▹ output: index of largest number
```

Back to language models

Given a list of tokens, predict the most likely token to follow.

PLEASE MAKE ME A CUP OF <u>TEA</u>

Tokenization

We represent a string:

“I’M LATE!”, HE SAID.

As a **list** of tokens:

“	I	‘M	LATE	!	“	,	HE	SAID	.

Language model algorithm

Predict the next token in the list.

PLEASE MAKE ME A CUP OF $\longrightarrow$ COFFEE

ONE TWO THREE $\longrightarrow$ FOUR

WHAT IS YOUR PHONE $\longrightarrow$ NUMBER

But the list of tokens is not enough.

We also need to know the language.

Corpora

A **text corpus** is used to analyze the distribution of words.

linguistics. It was Busa and Juilland together who worked out much of the foundations of modern corpus linguistics. If their contributions are less than well known, it is largely because corpus linguistics became closely associated with work in English only, and neither Busa nor Juilland worked on corpora of modern English. To review work in English linguistics, we need to consider the last two major groups working on corpus linguistics from the 1950s onwards.

1.5.3. Work arising from the study of English grammar

Work on English corpus linguistics started in the early 1960s, when Quirk (1960) planned and executed the construction of his ambitious Survey of English Usage (SEU). In the same year, Francis and Kucera began work on the now famous Brown corpus, a work which was to take almost two decades to complete.[24] These researchers were in a minority, but they were not universally regarded as peculiar in the field of English language studies, and others followed their lead. For example, in 1975, fourteen years after work began on the Brown corpus, Jan Svartvik started to build on the work of the SEU and the Brown corpus to construct the London-Lund corpus. The computer became the mainstay of English corpus linguistics in the 1970s. Svartvik computerised the SEU and, as a consequence, produced what, as Leech (1991: 9) said, was for a long time 'an unmatched resource for the study of spoken English'. With the appearance in the mid 1990s of large-scale corpora of spontaneous spoken English, such as the spoken section of the British National Corpus, the importance of the London-Lund corpus has faded somewhat. Even so, it is still the only corpus of spontaneous spoken English annotated with prosodic mark-up and, as such, still retains a niche in modern corpus linguistics to this day.

Another side effect of the work of the SEU was the training of academics in a tradition of corpus-based approaches to the grammatical analysis of English. Geoffrey Leech was associated with the early days of the SEU and went on to Lancaster to start a corpus research centre which has given rise to a number of well-known corpus-building projects, including perhaps most famously the Lancaster-Oslo-Bergen corpus (LOB) and more recently the British National Corpus. Sidney Greenbaum was also associated with the SEU as sometime assistant director of the project under Quirk. Greenbaum went on to succeed Quirk as director of the SEU in the mid-1980s and founded the International Corpus of English project. So the SEU was important as it spawned useful corpus resources and trained some linguists who later went on to become pioneers in the field of English corpus linguistics.

The work of Francis and Kucera, as well as that of Quirk and his disciples, inspired centres of English corpus building and exploitation beyond the United Kingdom in addition to Lund. English corpus linguistics in the tradition of the SEU steadily grew in Europe throughout the 1970s and 1980s with centres for corpus work being established across Scandinavia (e.g. Bergen, Gothenburg, Oslo), Western Europe (e.g. Berlin, Chemnitz, Nijmegen) and

Eastern Europe[25] (e.g. Leipzig, Potsdam). There is little doubt that a great deal of the current popularity of corpus linguistics, especially in studies of the English language, can be traced to this line of work. However, another related, though somewhat separate, strand of corpus work has been similarly influential in English corpus linguistics over the past forty years. That is the work of the neo-Firthians.

1.5.4. Work by neo-Firthians

J. R. Firth had a colourful life by any definition of the term. He studied in Leeds, picked up an interest in language while on military service in Afghanistan, Africa and India and went on to be professor firstly of English at Lahore and later at the School of Oriental and African Studies from 1944 (where he started as a senior lecturer in 1938, being appointed professor in 1944). His impact upon English – and more specifically British – linguistics has been notable. He was deeply influenced by the work of the anthropologist Bronislaw Malinowski and the phonetician Daniel Jones. Firth produced a series of writings in the 1930s, 1940s and 1950s which were published in a compendium format in Firth (1957). In this collection of papers Firth (1957: 27) outlines an approach to language in which social context and the social purpose of communication are paramount:[26]

> The central concept … is the context of situation. In that context are the human participant or participants, what they say, what is going on. The phonetician can find his phonetic context, and the grammarian and the lexicographer theirs.

Firth's agenda dominated much of British linguistics for the best part of a generation. As stated, its penetration beyond the United Kingdom was never anywhere near as great as it was within the United Kingdom. In America in particular it cannot be said that Firth's views ever constituted a dominant paradigm of linguistic research. Indeed, some of the most trenchant criticism of the Firthian approach to language can be found in the writing of American linguists, e.g. Langendoen (1968), though it is possible to find even stronger critical voices raised by others, e.g. Lyons (1968). Firth's place in corpus linguistics is assured, however, largely because he stated (Firth, 1957: 29) that 'Attested language … duly recorded is in the focus of attention for the linguist' and used some terminology which is used to this day in corpus linguistics. On the data side, his exhortation to study 'attested language' inspired what we will call neo-Firthian linguists, such as Halliday, Hoey and Sinclair, to work in the tradition he established.

On the terminology side, his term *collocation* is in use in modern corpus linguistics to this day (see section 3.4.4 for example). However, the popularity of that term can most easily be understood in the context of later corpus linguists, such as Sinclair, using the term. Collocation as a concept has a history

Counts table

To represent token counts, we map strings to numbers.

Some ready-made counts:

books.google.com/ngrams

,	775
THE	630
.	392
“	345
AND	339
A	337
TO	277

Algorithm to count words

Count all words in a tokenized corpus

and return a table of counts.

[I, AM, SAM, .,

SAM, I, AM.,

I, DO, NOT, LIKE,

GREEN, EGGS, AND, HAM, .]

→

I	3	LIKE	1
AM	2	GREEN	1
SAM	2	EGGS	1
.	3	AND	1
DO	1	HAM	1
NOT	1		

Algorithm to count words

```
count(L):
    C1 ← [0]
    i ← 1
    while i ≤ len(L):
        t ← L[i]
        C1[t] ← C1[t] + 1
        i ← i + 1
    return C1
```

- L is a list of tokens
- create a table of zeros
- assign 1 to i
- repeat while i is at most len(L)
- get token at position i
- increase count for t by 1
- increase i by 1
- output is the counts table

Algorithm to count words

Using a word as an index to a table

```
count(L):
    C1 ← [0]
    i ← 1
    while i ≤ len(L):
        t ← L[i]
        C1[t] ← C1[t] + 1
        i ← i + 1
    return C1
```

- ▷ L is a list of tokens
- ▷ create a table of zeros
- ▷ assign 1 to i
- ▷ repeat while i is at most len(L)
- ▷ get token at position i
- ▷ increase count for t by 1
- ▷ increase i by 1
- ▷ output is the counts table

Word counts

Example counts from *Alice's Adventures in Wonderland* (1866) by Lewis Carroll

,	775
THE	630
.	392
“	345
AND	339
A	337
TO	277

Unigram Language Model

Easiest: always predict the most frequent token:

I WISH I ,

C1 =

(**Unigram** counts)

C1[,] = 775

C1[THE] = 630

,	**775**
THE	630
.	392
“	345
AND	339
A	337
TO	277

Bigram counts

We can also count **bigrams** (pairs of words)

C2 =

,	THE	530
AND	THE	320
.	"	89
SHE	SAID	65
	…	
I	'M	20
I	DO	10

C2[AND, THE] = 320

C2[SHE, SAID] = 65

Bigram Language Model

Look only at the **last** token to predict the next:

I WISH **I 'M**

C2[I, ·] =

(**Bigram** counts starting with I)

I	**'M**	**20**
I	DO	10
I	'LL	10
I	'VE	10
I	SHOULD	8
I	MUST	7
I	THINK	7

C2[I, 'M] = 20

C2[I, DO] = 10

Trigram Language Model

Look at the **two** last tokens to predict the next:

I **WISH I COULD**

C3[WISH, I, ·] =

WISH I COULD	**20**
WISH I HAD	10

(**Trigram** counts starting with WISH I)

C3[WISH, I, COULD] = 20

C3[WISH, I, HAD] = 10

n-gram Language Model

Look at the ***n* − 1** last tokens to predict the next:

I **WISH I COULD**

C4[I, WISH, I, ·] =

I WISH I COULD	**2**
I WISH I HAD	2

(**4-gram** counts starting with I WISH I):

C3[I, WISH, I, COULD] = 2

C3[I, WISH, I, HAD] = 2

Algorithm to count *n*-grams

count(L, n):	▷ L: list of tokens, n: a number
$C \leftarrow [0]$	▷ create a table of zeros
$i \leftarrow 1$	▷ assign 1 to i
while $i \leq \text{len}(L) - n + 1$:	▷ repeat while i is at most len(L) − n + 1
$T \leftarrow L[i, \ldots, i + n - 1]$	▷ get n tokens starting at i
$C[T] \leftarrow C[T] + 1$	▷ increase count for T by 1
$i \leftarrow i + 1$	▷ increase i by 1
return C	▷ output is the counts table

Algorithm to count *n*-grams

Getting several elements from a list

```
count(L, n):                          ▷ L: list of tokens, n: a number
    C ← [0]                           ▷ create a table of zeros
    i ← 1                             ▷ assign 1 to i
    while i ≤ len(L) − n + 1:         ▷ repeat while i is at most len(L) − n + 1
        T ← L[i, ..., i + n − 1]      ▷ get n tokens starting at i
        C[T] ← C[T] + 1               ▷ increase count for T by 1
        i ← i + 1                     ▷ increase i by 1
    return C                          ▷ output is the counts table
```

Algorithm to count *n*-grams

Using an *n*-gram as an index to a table

```
count(L, n):                          ▷ L: list of tokens, n: a number
    C ← [0]                           ▷ create a table of zeros
    i ← 1                             ▷ assign 1 to i
    while i ≤ len(L) − n + 1:         ▷ repeat while i is at most len(L) − n + 1
        T ← L[i, ..., i + n − 1]      ▷ get n tokens starting at i
        C[T] ← C[T] + 1               ▷ increase count for T by 1
        i ← i + 1                     ▷ increase i by 1
    return C                          ▷ output is the counts table
```

Unigram algorithm

```
unigram(L, C1):
    return argmax(C1)
```

- L: tokens, C1: unigram counts
- token with highest count

Unigram algorithm

```
unigram(L, C1):
    return argmax(C1)
```

Function call

▹ L: tokens, C1: unigram counts

▹ token with highest count

Unigram algorithm

```
unigram(L, C1):
    return argmax(C1)
```

- L: tokens, C1: unigram counts
- token with highest count

Ignores L and always predicts the same word...

Bigram algorithm

bigram(L, C2):

k ← len(L)

t ← L[k]

return argmax(C2[t, ·])

▷ L: tokens, C2: bigram counts

▷ length of L

▷ last token in L

▷ bigram with highest count,

▷ among the bigrams starting with t

Bigram algorithm

bigram(L, C2): ▷ L: tokens, C2: bigram counts

k ← len(L) ▷ length of L

t ← L[k] ▷ last token in L

return argmax($\boxed{\text{C2[t, ·]}}$) ▷ bigram with highest count, ▷ among the bigrams starting with t

Getting part of the table

Trigram algorithm

trigram(L, C3):

  k ← len(L)

  T ← L[k − 1, k]

  return argmax(C3[T, ·])

- L: tokens, C3: trigram counts
- length of L
- last two tokens in L
- trigram with highest count,
- among the trigrams starting with T

General *n*-gram algorithm

ngram(L, n, Cn): ▹ L: tokens, Cn: *n*-gram counts

$k \leftarrow \text{len}(L)$ ▹ length of L

$T \leftarrow L[k - n + 2, ..., k]$ ▹ last n − 1 tokens in L

return argmax(Cn[T, ·]) ▹ n-gram with highest count,

▹ among the n-grams starting with T

This can replace **unigram**, **bigram** and **trigram** algorithms: just use n=1, n=2 or n=3

Text prediction algorithm

predict(L, n, Cn, m): ▹ L: tokens, Cn: *n*-gram counts,

▹ m: total wanted number of words

P ← L ▹ start with words given as input

while len(P) < m: ▹ repeat until we have m words

P[len(P) + 1] ← ngram(P, n, Cn) ▹ add next word

return P ▹ output is list of words including input

n-gram models comparison

Unigram	, ,
Bigram	THEN SHE WENT ON IT HAD BEEN RUNNING ABOUT IN HER HEAD ! THE GARDEN , WHO WAS NOW , FOR SOME OF THEM !
Trigram	ALL OF A GOOD DEAL FRIGHTENED AT THE TOP OF HER SISTER , WHO WAS GENTLY BRUSHING AWAY SOME DEAD LEAVES THAT HAD FALLEN INTO A TREE A FEW MINUTES , IT WAS THE WHITE RABBIT , WHO WAS NOW ABOUT TWO FEET HIGH .
4-gram	THE FIRST THING I 'VE GOT TO DO , SO ALICE SOON BEGAN TALKING TO HERSELF . `` DINAH 'LL MISS ME VERY MUCH TO-NIGHT , I SHOULD THINK ! '' (DINAH WAS THE CAT
5-gram	AND SO IT WAS INDEED ! SHE WAS NOW ONLY TEN INCHES HIGH , AND HER FACE BRIGHTENED UP AT THE THOUGHT THAT SHE WAS NOW ABOUT TWO FEET HIGH AND WAS GOING ON SHRINKING RAPIDLY .

Back-off

n-gram models quickly become too **sparse**.

Whenever I wish I

does not occur in *Alice in Wonderland*: cannot use 5-gram.

If no match is found, use a smaller *n*:

To predict the next token, **back-off** to 4-grams:

Whenever **I wish I could**

I wish I could	**2**
I wish I had	2

Trigram with Backoff to Bigram

```
trigram-backoff-bigram(L, C2, C3):   ▷ L: tokens,
    k ← len(L) ▷ C2: bigram counts, C3: trigram counts
    if C3[L[k − 1, k], ·]) is empty :        ▷ not found
        return argmax(C2[L[k], ·])           ▷ use bigram
    else:                                    ▷ trigram found
        return argmax(C3[L[k − 1, k], ·])    ▷ use trigram
```

Trigram with Backoff to Bigram

trigram-backoff-bigram(L, C2, C3): ▷ L: tokens,

k ← len(L) ▷ C2: bigram counts, C3: trigram counts

if C3[L[k − 1, k], ·]) is empty : ▷ not found

return argmax(C2[L[k], ·]) ▷ use bigram

else: ▷ trigram found

return argmax(C3[L[k − 1, k], ·]) ▷ use trigram

if/else condition

Trigram with Full Backoff

trigram-backoff(L, C): ▷ L is a list of tokens,

$k \leftarrow$ len(L) ▷ C is the list [C1, C2, C3]:

$i \leftarrow 3$ ▷ unigram, bigram, trigram counts

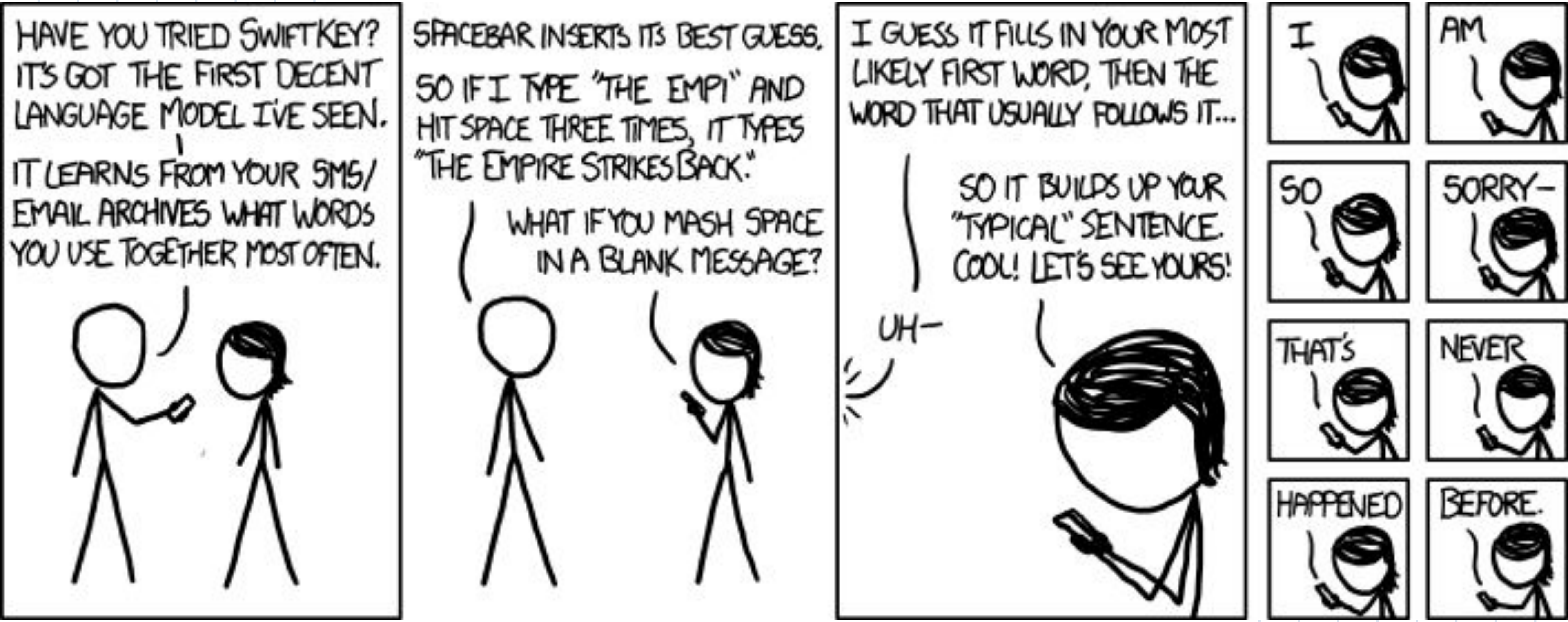
while $C[i][L[k - i + 2, ..., k], \cdot]$ is empty:

  $i \leftarrow i - 1$ ▷ *i*-gram not found, try $i - 1$

return argmax$(C[i][L[k - i + 2, ..., k], \cdot])$

References

- Google Ngram Viewer: books.google.com/ngrams
- *Alice's Adventures in Wonderland* on Wikisource: en.wikisource.org/wiki/Alice's_Adventures_in_Wonderland_(1866)
- *n*-grams: en.wikipedia.org/wiki/N-gram



xkcd.com/1068