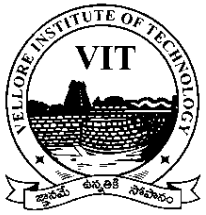


VIT-AP UNIVERSITY

Natural Language Processing

(Course Code: CSE 3015)

Module-2:Lecture-1: Naive Bayes Classifier and N-Gram Model

Gundimeda Venugopal, Professor of Practice, SCOPE

Probability Refresher

Basic Probability

❖**Probability Theory**: predicting how likely it is that something will happen.

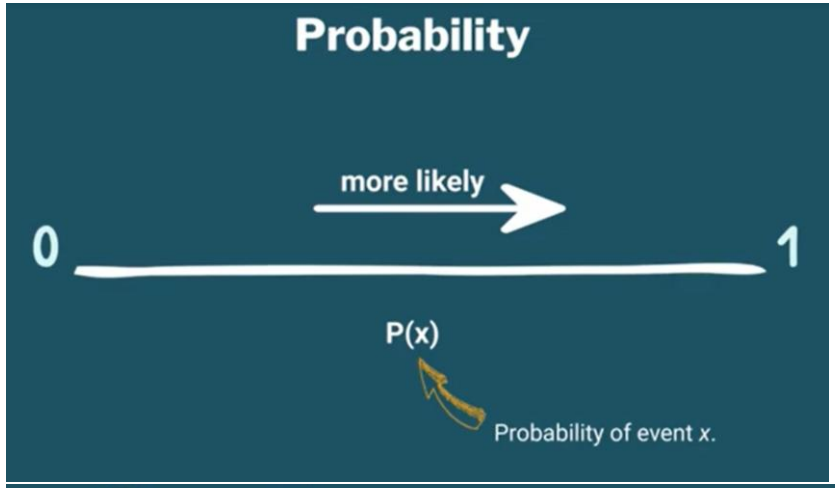
❖**Probabilities**: numbers between 0 and 1.

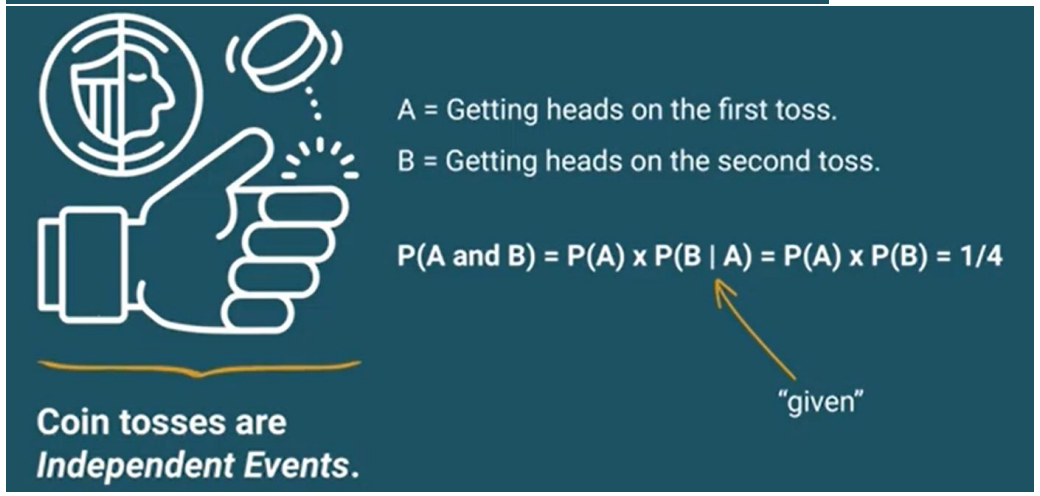
❖•**Probability Function**:

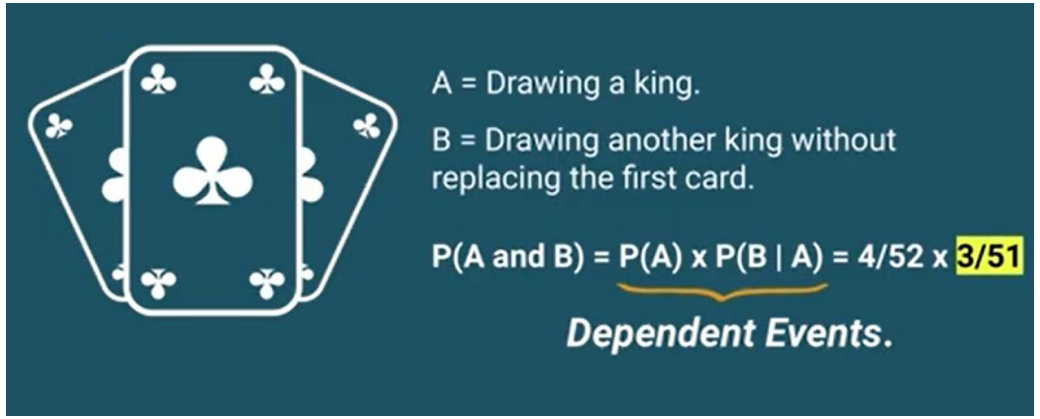
- P(A) means that how likely the event A happens.
- P(A) is a number between 0 and 1
- P(A)=1 => a certain event
- P(A)=0 => an impossible event

❖**Example**: a coin is tossed three times.

- What is the probability of 3 heads? 1/8 uniform distribution







Conditional Probability + Bayes' Theorem

Conditional Probability

$$P(A \text{ and } B) = P(A) \times P(B \mid A)$$

$$P(A \text{ and } B) = P(B \text{ and } A)$$

$$P(A \text{ and } B) = P(A) \times P(B \mid A)$$

$$P(B \text{ and } A) = P(B) \times P(A \mid B)$$

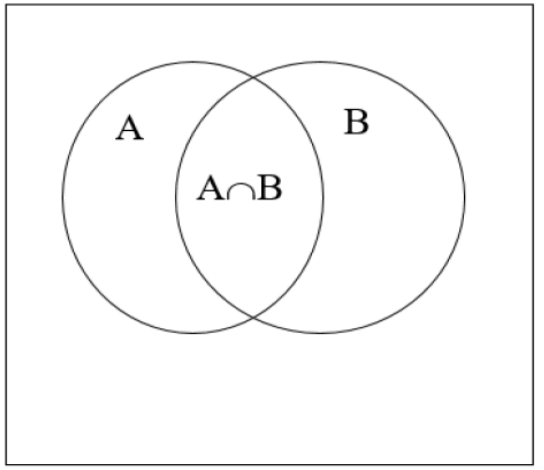
$$P(B) \times P(A \mid B) = P(A) \times P(B \mid A)$$

Bayes' Theorem

$$P(A \mid B) = \frac{P(A)P(B \mid A)}{P(B)}$$

Unconditional and Conditional Probability

- **Unconditional Probability** or **Prior Probability**
  - P(A)
  - the probability of the event A does not depend on other events.
- **Conditional Probability -- Posterior Probability -- Likelihood**
  - P(A|B)
  - this is read as the probability of A given that we know B.



$P(A|B) = P(A \cap B) / P(B)$

$P(B|A) = P(A \cap B) / P(A)$

Example:

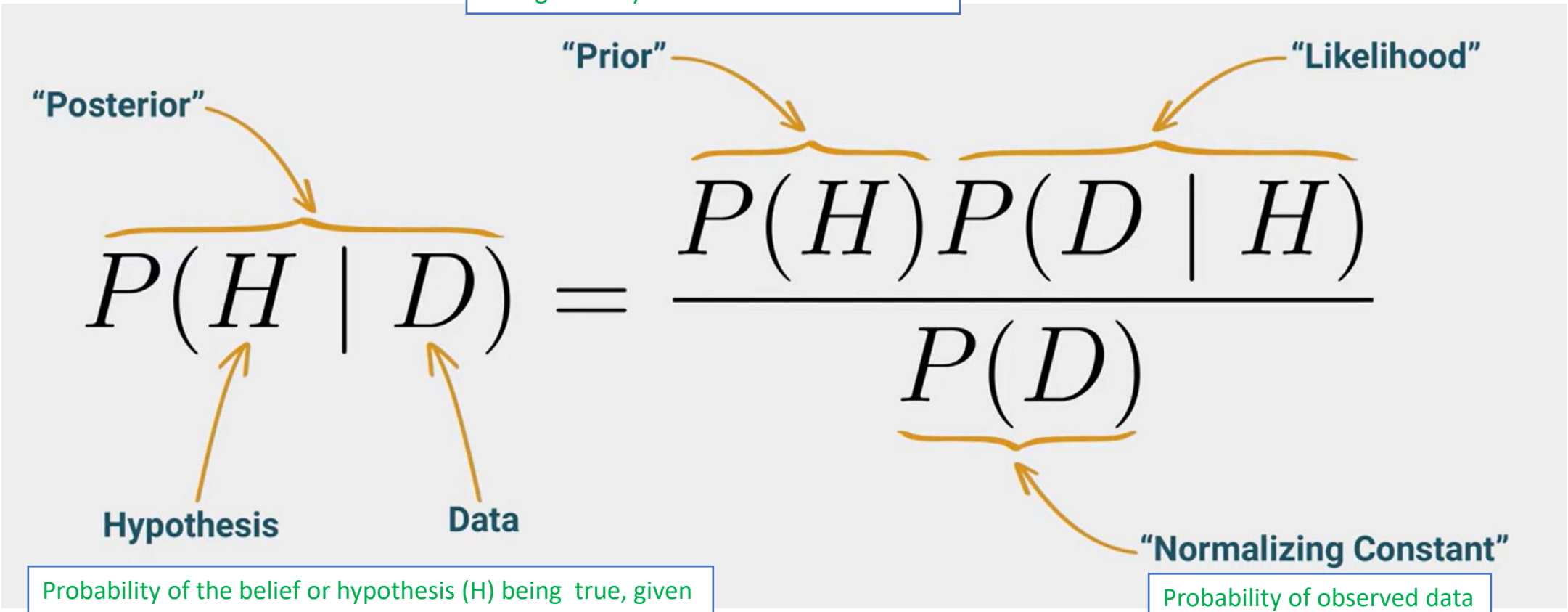
- P(put) is the probability of to see the word *put* in a text
- P(on|put) is the probability of to see the word *on* after seeing the word *put*.

- **Joint Probability**
  - $P(A \cap B)$ or P(A,B)
  - the probability of the events A and B occur together

Bayes' Theorem

Quantify the belief or hypothesis without taking into any of the observed data

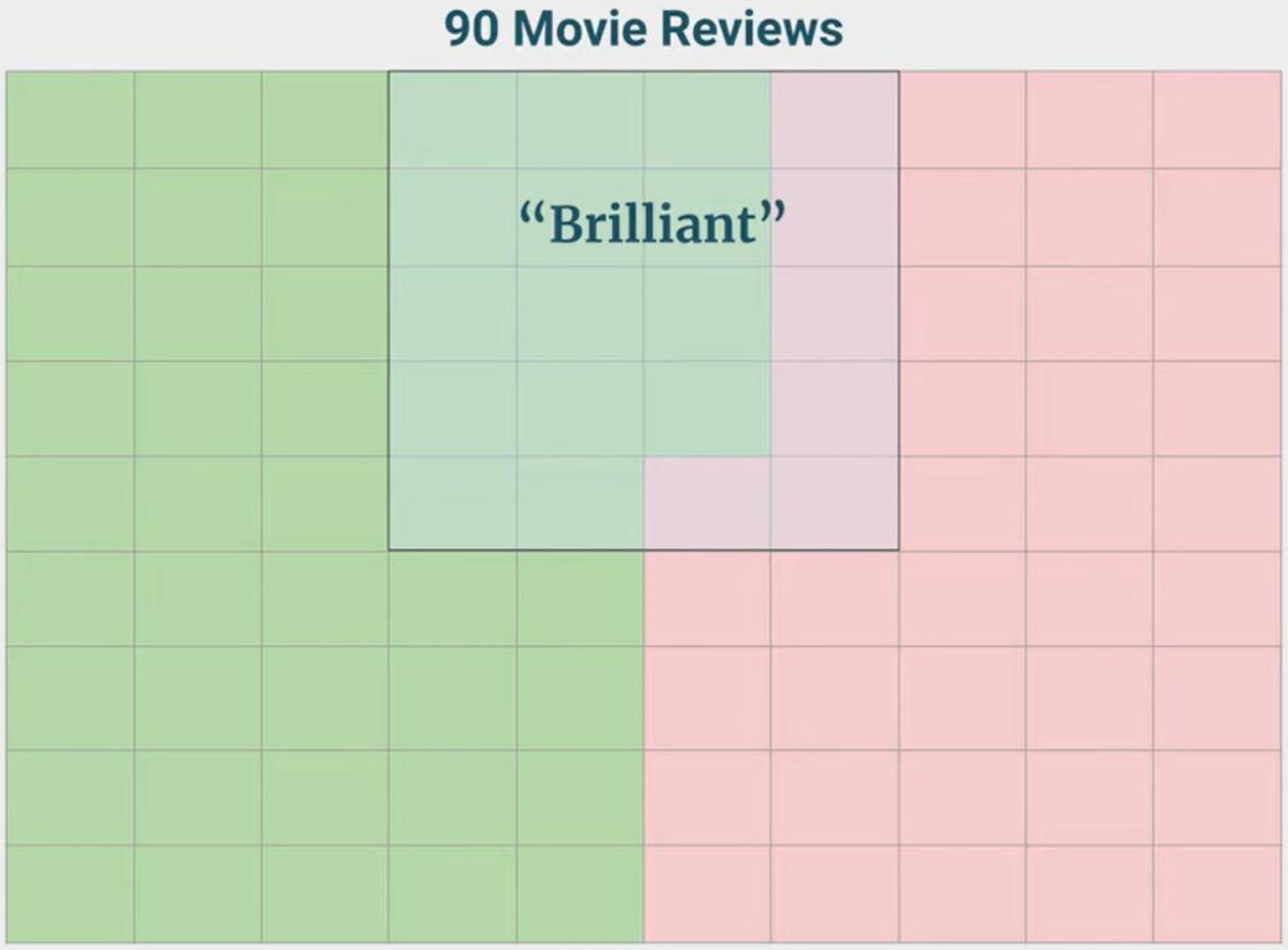
Probability of observed data, given our hypothesis or belief



Probability of the belief or hypothesis (H) being true, given some observed data (D).
H is some Hypothesis or Belief we hold
D is some body of observed data

Probability of observed data

Bayes' Theorem Example



$$P(H \mid D) = \frac{P(H)P(D \mid H)}{P(D)}$$

P(Positive | "Brilliant") ?

P(Positive) = 49/90

P("Brilliant" | Positive) = 14/49

P("Brilliant") = 20/90

$$P(\text{positive} \mid \text{"brilliant"}) = \frac{14}{49} * \frac{49/90}{20/90} = 0.7$$

Naive Bayes Text Classifier

Bayes' Theorem – Applied to Text Classification

$$P(c \mid d) = \frac{P(c)P(d \mid c)}{P(d)}$$

Class Document

$$\underset{c \in C}{\operatorname{argmax}}\ P(c \mid d) = \underset{c \in C}{\operatorname{argmax}}\ \frac{P(c)P(d \mid c)}{P(d)}$$

Specifically, find the class which *maximizes* the posterior possibility (i.e. the most probable class).

No need to normalize.

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

Most probable class.

Prior

$$\frac{\text{Number of documents of class } c}{\text{Total number of documents}}$$

Likelihood

$$P(d \mid c) = P([w_1, w_2, w_3, \ldots, w_n] \mid c)$$

Text Classification: Likelihood calculation

The likelihood is difficult to calculate

(as it stands)

$$P(d \mid c) = P([w_1, w_2, w_3, \ldots, w_n] \mid c)$$

So we make two simplifying assumptions:

1. Word order doesn't matter, so we use BOW representations.
2. Word appearances are *independent* of each other given a particular class.

"Naive"

Naive assumption: Word appearances are *independent* of each other given a particular class.

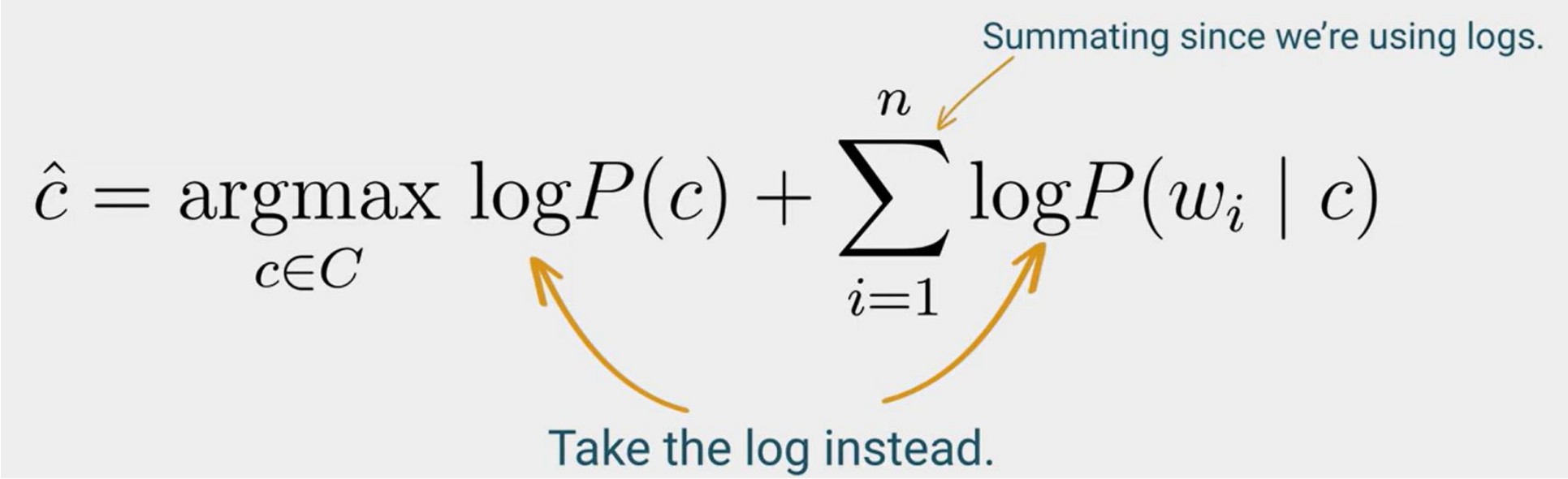
$$P(d \mid c) = P([w_1, w_2, w_3, \ldots, w_n] \mid c)$$

With the naive assumption...

$$P(d \mid c) = P(w_1 \mid c) \cdot P(w_2 \mid c) \cdot \ldots \cdot P(w_n \mid c)$$

Multiplying lots of small probabilities together

Naive Bayes Classifier



Naive Bayes Classifier: Training + Scoring

Document	Class
"Beyond the pandemic"	Medical
"Compelling jumble of philosophy"	Not medical
"Vaccine rollout enters second phase"	Medical
"Man regains sight with implant"	Medical
"My toilet is haunted"	Not medical

Priors, P(c)

P(medical) = **3/5**

P(not medical) = **2/5**

Likelihoods, $P(w_i \mid c)$

$$\frac{\text{Count of } w_i \text{ in c}}{\text{Count of } \textit{all} \text{ w in c}}$$

P(**vaccine** | medical) = $\frac{1}{13}$

Likelihoods, $P(w_i \mid c)$

$$\frac{\text{Count of } w_i \text{ in c}}{\text{Count of } \textit{all} \text{ w in c}}$$

P(**haunted** | medical) = $\frac{0}{13}$

Laplacian Smoothing

|V| is the size of the total vocabulary.

Likelihoods, $P(w_i \mid c)$

$$\frac{\text{Count of } w_i \text{ in c} + \alpha}{\text{Count of } \textit{all} \text{ w in c} + \alpha|V|}$$

Likelihoods, $P(w_i \mid c)$

$$\frac{\text{Count of } w_i \text{ in c} + \alpha}{\text{Count of } \textit{all} \text{ w in c} + \alpha|V|}$$

P(**haunted** | medical) = $\frac{0 + 1}{13 + 21}$

Find most probable class for "vaccine trial"

P(medical | S) = log P(medical) + log P(vaccine | medical) + log P(trial | medical)
= log (3/5) + log(2/34) + log(1/34) = -6.87 ← Most probable

P(not medical | S) = log P(not medical) + log P(vaccine | not medical) + log P(trial | not medical)
= log (2/5) + log(1/29) + log(1/29) = -7.65

Convert to probabilities

P(medical | S) = -6.87 ⇒ exp(-6.87) = 0.001

P(not medical | S) = -7.65 ⇒ exp(-7.65) = 0.0004

P(medical | S) = $\frac{0.001}{0.001 + 0.0004}$ = 0.71

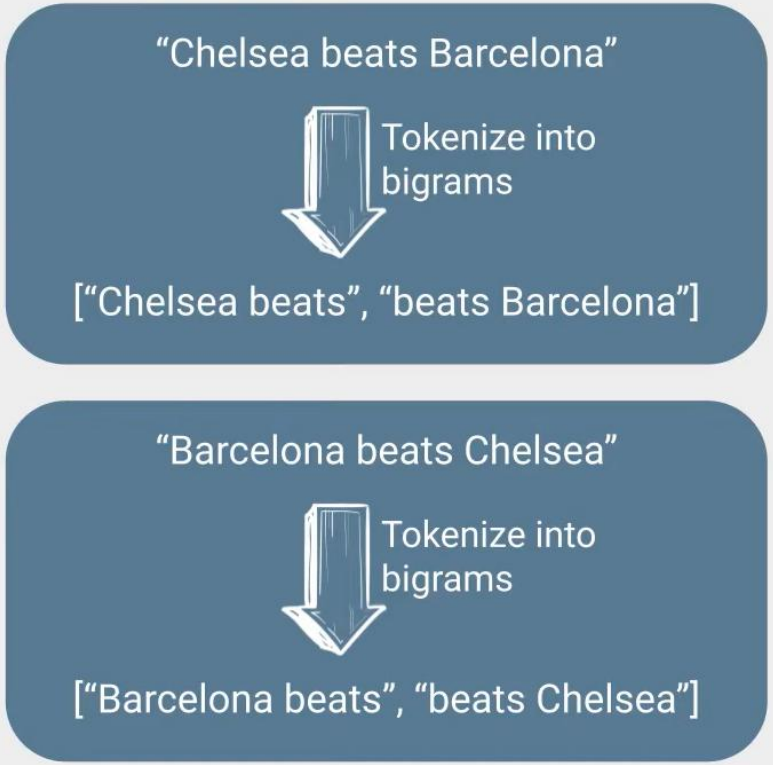
Just a rough measure.

N-grams

Chunks of continuous tokens

- 2-gram or *bigram* has two tokens per chunk
- 3-gram or *trigram* has three tokens per chunk

.
.
.



Helps capture some context our previous approach didn't.

The vocabulary is the collection of bigrams across the corpus.

N-grams

Can combine tokens (unigrams) and n-grams

e.g. tokenize as usual and keep only meaningful n-grams ("*social media*", "*los angeles*", "*witch hunt*") based on some metric.

Couple of ideas

- Look for high-frequency n-grams, or n-grams with certain POS structure, or anything flagged by NER tagger.
- Use a measure of association such as *Pointwise Mutual Information (PMI)*.

N-grams help us capture some context...

But still

- No way to handle Out-of-Vocabulary (OOV) words.
- Dimensionality increases rapidly.

N-gram exercises

Chain Rule of Probability

- ***How can we compute probabilities of entire word sequences like $w_1,w_2,\ldots,w_n$?***
  - The probability of the word sequence $w_1,w_2,\ldots,w_n$ is $P(w_1,w_2,\ldots,w_n)$.
- We can use the **chain rule of the probability** to decompose this probability:

$$\mathbf{P(w_1^n) = P(w_1)\,P(w_2|w_1)\,P(w_3|w_1^2)\ldots P(w_n|w_1^{n-1})}$$

$$\mathbf{= \prod_{k=1}^{n} P(w_k \mid w_1^{k-1})}$$

Example:

P(the man from jupiter) =

P(the) P(man|the) P(from|the man) P(jupiter|the man from)

Chain Rule of Probability and Conditional Probabilities

- The chain rule shows the link between computing the joint probability of a sequence and computing the conditional probability of a word given previous words.

- Definition of Conditional Probabilities:

$$\mathbf{P(B|A) = P(A,B) / P(A) \quad \rightarrow \quad P(A,B) = P(A)\, P(B|A)}$$

- Conditional Probabilities with More Variables:

$$P(A,B,C,D) = P(A)\, P(B|A)\, P(C|A,B)\, P(D|A,B,C)$$

- **Chain Rule:**

$$\mathbf{P(w_1 \ldots w_n) = P(w_1)\, P(w_2|w_1)\, P(w_3|w_1 w_2) \ldots P(w_n|w_1 \ldots w_{n-1})}$$

Context Sensitive Spelling Correction

The office is about fifteen minuets from my house

min·u·et *noun* \min-yə-'wet\

: a slow, graceful dance that was popular in the 17th and 18th centuries

: the music for a minuet

Use a Language Model

P(about fifteen **minutes** from) > P(about fifteen **minuets** from)

Probabilistic Language Models: Applications

Speech Recognition

- P(I saw a van) >> P(eyes awe of an)

Machine Translation

Which sentence is more plausible in the target language?

- P(**high** winds) > P(**large** winds)

Other Applications

- Context Sensitive Spelling Correction
- Natural Language Generation
- …

Web search engine / …

I saw a cat

I saw a cat on the chair
I saw a cat running after a dog
I saw a cat in my dream
I saw a cat book

Spell Correction:

- Thek office is about ten minutes from here
- P(The office is) > P(Then office is)

Completion Prediction

- Language model also supports predicting the completion of a sentence.
  - Please turn off your cell …
  - Your program does not …
- *Predictive text input* systems can guess what you are typing and give choices on how to complete it.

Probabilistic Language Modeling: Language Model

- **Goal:** Compute the probability of a sentence or sequence of words

$$P(W) = P(w_1, w_2, w_3, \ldots, w_n)$$

- **Related Task:** probability of an upcoming word:

$$P(w_4|w_1, w_2, w_3)$$

- A model that computes either of these is called a **language model**

Language Models

- A Language model is a machine learning model that uses statistical and probabilistic techniques to predict the likelihood of words or sequences of words occurring in a sentence or phrase.
- Language Models (LMs) estimate the probability of different linguistic units: symbols, tokens, token sequences.

Web search engine / ...

A good model would simulate the behavior of the real world: it would "understand" which events are in better agreement with the world, i.e., which of them are more likely.

N-Grams

❖The intuition of the n-gram model (simplifying assumption):

- instead of computing the probability of a word given its entire history, we can approximate the history by just the last few words.

❖ $P(w_n|w_1 \ldots w_{n-1}) \approx P(w_n)$ **unigram**

❖ $P(w_n|w_1 \ldots w_{n-1}) \approx P(w_n|w_{n-1})$ **bigram**

❖ $P(w_n|w_1 \ldots w_{n-1}) \approx P(w_n|w_{n-1}\, w_{n-2})$ **trigram**

❖ $P(w_n|w_1 \ldots w_{n-1}) \approx P(w_n|w_{n-1}\, w_{n-2}\, w_{n-3})$ **4-gram**

❖ $P(w_n|w_1 \ldots w_{n-1}) \approx P(w_n|w_{n-1}\, w_{n-2}\, w_{n-3}\, w_{n-4})$ **5-gram**

❖**In general, N-Gram is**

$P(w_n|w_1 \ldots w_{n-1}) \approx P(w_n|w_{n-1}\, w_{n-2}\, w_{n-3} \ldots w_{n-N+1})$ **N-gram**

Or

$$P(w_n|w_1 \ldots w_{n-1}) \approx P(w_n|w_{n-N+1}^{n-1})$$

N-Grams: computing probabilities of word sequences

Unigrams -- $P(w_1^n) \approx \prod_{k=1}^{n} P(w_k)$

Bigrams -- $P(w_1^n) \approx \prod_{k=1}^{n} P(w_k \mid w_{k-1})$

Trigrams -- $P(w_1^n) \approx \prod_{k=1}^{n} P(w_k \mid w_{k-1} w_{k-2})$

4-grams -- $P(w_1^n) \approx \prod_{k=1}^{n} P(w_k \mid w_{k-1} w_{k-2} w_{k-3})$

N-Grams: computing probabilities of word sequences (Sentences)

❖ Unigram

- P(<s> the man from Jupiter came </s>) ≈ P(the) P(man) P(from) P(jupiter) P(came)

❖ Bigram

- P(<s> the man from Jupiter came </s>) ≈ P(the |<s>) P(man | the) P(from | man) P(Jupiter | from) P(came | jupiter) P(</s>|came)

❖ Trigram

- P(<s> the man from Jupiter came </s>) ≈ P(the |<s> <s>) P(man|<s>the) P(from|the man) P(jupiter|man from) P(came|from jupiter) P(</s>|Jupiter came) P(</s>|came </s>)

N-Gram models

- In general, a n-gram model is an insufficient model of a language because languages have **long-distance dependencies**.
  - "The **computer(s)** which I had just put into the machine room **is (are)** crashing."
  - But we can still effectively use N-Gram models to represent languages.
- Which N-Gram should be used as a language model?
  - Bigger N, the model will be more accurate.
    - But we may not get good estimates for N-Gram probabilities.
    - The N-Gram tables will be more sparse.
  - Smaller N, the model will be less accurate.
    - But we may get better estimates for N-Gram probabilities.
    - The N-Gram table will be less sparse.
  - In reality, we do not use higher than Trigram (not more than Bigram).
  - How big are N-Gram tables with 10,000 words?
    - Unigram --10,000
    - Bigram – 10,000*10,000 = 100,000,000
    - Trigram – 10,000*10,000*10,000 = 1,000,000,000,000

N-Grams and Markov Models

- The assumption that the probability of a word depends only on the previous word(s) is called ***Markov assumption***
- ***Markov models*** are the class of probabilistic models that assume that we can predict the probability of some future unit without looking too far into the past.
- A bigram is called a ***first order Markov model*** (because it looks one token into the past);
- A trigram is called a ***second order Markov model***;
- In general a N-Gram is called a ***N-1 order Markov model***.

bigram

The **bigram** model, for example, approximates the probability of a word given all the previous words $P(w_n|w_{1:n-1})$ by using only the conditional probability of the preceding word $P(w_n|w_{n-1})$. In other words, instead of computing the probability

$$P(\texttt{blue}|\texttt{The water of Walden Pond is so beautifully}) \quad (3.5)$$

we approximate it with the probability

$$P(\texttt{blue}|\texttt{beautifully}) \quad (3.6)$$

When we use a bigram model to predict the conditional probability of the next word, we are thus making the following approximation:

$$P(w_n|w_{1:n-1}) \approx P(w_n|w_{n-1}) \quad (3.7)$$

Estimating N-Gram Probabilities

- Estimating n-gram probabilities is called maximum likelihood estimation (or MLE).
- We get the MLE estimate for the parameters of an n-gram model by getting counts from a corpus, and normalizing the counts so that they lie between 0 and 1.
- Estimating bigram probabilities:

$$P(w_n|w_{n-1}) = \frac{C(w_{n-1}w_n)}{\sum_w C(w_{n-1}w)} = \frac{C(w_{n-1}w_n)}{C(w_{n-1})}$$

where C is the count of that pattern in the corpus

- Estimating N-Gram probabilities:

$$P(w_n \mid w_{n-N+1}^{n-1}) = \frac{C(w_{n-N+1}^{n-1}w_n)}{C(w_{n-N+1}^{n-1})}$$

For example, to compute a particular bigram probability of a word w_n given a previous word w_{n-1}, we'll compute the count of the bigram $C(w_{n-1}w_n)$ and normalize by the sum of all the bigrams that share the same first word w_{n-1}:

$$P(w_n|w_{n-1}) = \frac{C(w_{n-1}w_n)}{\sum_w C(w_{n-1}w)} \quad (3.10)$$

We can simplify this equation, since the sum of all bigram counts that start with a given word w_{n-1} must be equal to the unigram count for that word w_{n-1} (the reader should take a moment to be convinced of this):

$$P(w_n|w_{n-1}) = \frac{C(w_{n-1}w_n)}{C(w_{n-1})} \quad (3.11)$$

The equation estimates the n-gram probability by dividing the observed frequency of a particular sequence by the observed frequency of a prefix. This ratio is called a relative frequency.

This use of relative frequencies as a way to estimate probabilities is an example of Maximum Likelihood Estimation or MLE.

Unigram Calculation Example

A mini-corpus: We augment each sentence with a special symbol <s> at the beginning of the sentence, to give us the bigram context of the first word, and special end-symbol </s>.

<s>I am Sam</s>
<s>Sam I am</s>
<s>I fly</s>

P(*w*) = *w* word count in the corpus / Total words count in the corpus including duplicates

Unigrams: I, am, Sam, fly

P(I) = 3/8 P(am)=2/8 P(Sam)=2/8 P(fly)=1/8

Note: start and end tokens are not counted (as they are augmented)

Bigram probabilities calculation: Examples

Example 1:

<s> I am Sam </s>
<s> Sam I am </s>
<s> I do not like green eggs and ham </s>

Here are the calculations for some of the bigram probabilities from this corpus

$P(\text{I}|\text{<s>}) = \frac{2}{3} = 0.67$ $P(\text{Sam}|\text{<s>}) = \frac{1}{3} = 0.33$ $P(\text{am}|\text{I}) = \frac{2}{3} = 0.67$

$P(\text{</s>}|\text{Sam}) = \frac{1}{2} = 0.5$ $P(\text{Sam}|\text{am}) = \frac{1}{2} = 0.5$ $P(\text{do}|\text{I}) = \frac{1}{3} = 0.33$

Example 2:

$$P(w_i|w_{i-1}) = \frac{c(w_{i-1}, w_i)}{c(w_{i-1})}$$

<s>I am here </s>
<s>who am I </s>
<s>I would like to know </s>

Estimating bigrams

P(I|<s>) = 2/3
P(</s>|here) =1
P(would | I) = 1/3
P(here | am) = 1/2
P(know | like) = 0

Estimating N-Gram Probabilities

Corpus: Berkeley Restaurant Project Sentences

- There are 9332 sentences in the corpus.
- Raw biagram counts of 8 words (out of 1446 words)

	i	want	to	eat	chinese	food	lunch	spend
i	5	827	0	9	0	0	0	2
want	2	0	608	1	6	6	5	1
to	2	0	4	686	2	0	6	211
eat	0	0	2	0	16	2	42	0
chinese	1	0	0	0	0	82	1	0
food	15	0	15	0	1	4	0	0
lunch	2	0	0	0	0	1	0	0
spend	1	0	1	0	0	0	0	0

Corpus Analysis

Top Bigrams

i want
to eat
want to
to spend
chinese food
eat lunch

Low probable Bigrams

chinese eat
food want
i lunch
spend food
i spend

Estimating N-Gram Probabilities

Corpus: Berkeley Restaurant Project Sentences

	i	want	to	eat	chinese	food	lunch	spend
i	5	827	0	9	0	0	0	2
want	2	0	608	1	6	6	5	1
to	2	0	4	686	2	0	6	211
eat	0	0	2	0	16	2	42	0
chinese	1	0	0	0	0	82	1	0
food	15	0	15	0	1	4	0	0
lunch	2	0	0	0	0	1	0	0
spend	1	0	1	0	0	0	0	0

- Unigram counts:

i	want	to	eat	chinese	food	lunch	spend
2533	927	2417	746	158	1093	341	278

- Normalize bigrams by unigram counts:

	i	want	to	eat	chinese	food	lunch	spend
i	0.002	0.33	0	0.0036	0	0	0	0.00079
want	0.0022	0	0.66	0.0011	0.0065	0.0065	0.0054	0.0011
to	0.00083	0	0.0017	0.28	0.00083	0	0.0025	0.087
eat	0	0	0.0027	0	0.021	0.0027	0.056	0
chinese	0.0063	0	0	0	0	0.52	0.0063	0
food	0.014	0	0.014	0	0.00092	0.0037	0	0
lunch	0.0059	0	0	0	0	0.0029	0	0
spend	0.0036	0	0.0036	0	0	0	0	0

Bigram Estimates of Sentence Probabilities

- Some other bigrams:

  P(i|<s>)=0.25

  P(english|want)=0.0011

  P(food|english)=0.5

  P(</s>|food)=0.68

	i	want	to	eat	chinese	food	lunch	spend
i	0.002	0.33	0	0.0036	0	0	0	0.00079
want	0.0022	0	0.66	0.0011	0.0065	0.0065	0.0054	0.0011
to	0.00083	0	0.0017	0.28	0.00083	0	0.0025	0.087
eat	0	0	0.0027	0	0.021	0.0027	0.056	0
chinese	0.0063	0	0	0	0	0.52	0.0063	0
food	0.014	0	0.014	0	0.00092	0.0037	0	0
lunch	0.0059	0	0	0	0	0.0029	0	0
spend	0.0036	0	0.0036	0	0	0	0	0

- Compute the probability of sentence **I want English food**

P(<s> i want english food </s>)

= P(i|<s>) P(want|i) P(english|want) P(food|english) P(</s>|food)

= 0.25*0.33*0.0011*0.5*0.68

= 0.000031

What knowledge N-Gram represent?

What do you infer?

- P(english|want) = .0011
- P(chinese|want) = .0065
- P(to|want) = .66
- P(eat | to) = .28
- P(food | to) = 0
- P(want | spend) = 0
- P (i | <s>) = .25

Data Analysis

Chinese food is 6 times more popular than English food
Language: words “want to” occurs 66% of time
Language: words “to eat” occurs 28% of time
Language: words “to food” won’t occur together.
Language: words “spend want” won’t occur together.
Language related

Practical Issues: Log Probabilities

- Since probabilities are less than or equal to 1, the more probabilities we multiply together, the ***smaller the product becomes***.
  - Multiplying enough n-grams together would result in **numerical underflow**.
- By using **log probabilities** instead of ***raw probabilities***, we get numbers that are not as small.
  - Adding in log space is equivalent to multiplying in linear space, so we combine log probabilities by adding them.
    - adding is faster than multiplying
  - The result of doing all computation and storage in log space is that we only need to convert back into probabilities if we need to report them at the end

$$\log(p_1 \times p_2 \times p_3 \times p_4) = \log p_1 + \log p_2 + \log p_3 + \log p_4$$

$$p_1 \times p_2 \times p_3 \times p_4 = \exp(\log p_1 + \log p_2 + \log p_3 + \log p_4)$$

Everything in log space

- Avoids underflow
- Adding is faster than multiplying

$$log(p_1 \times p_2 \times p_3 \times p_4) = logp_1 + logp_2 + logp_3 + logp_4$$

Handling zeros

Use smoothing

Language models applications

- Autocomplete: A language model that predicts the most likely word or sequence of words to replace an underscore in a sentence
- Automatic speech recognition (ASR)
- Machine translation: Language models are used in Google's Live Translate
- Information retrieval: Language models are used to build a model for each document, and the document with the highest probability for a query is ranked the highest
- Text generation: Language models can be used to generate text, such as essays or poems, in response to a prompt or question
- Code writing: Some language models can help programmers write code
- Sentiment analysis: Language models can be used for sentiment analysis
- DNA research: Language models can be used in DNA research
- Customer service: Language models can be used in customer service
- Chatbots: Language models can be used in chatbots
- Online search: Language models can be used in online search

What is a Language Model?

- Probability distribution over strings of text
  - how likely is a given string (observation) in a given "language"
  - for example, consider probability for the following four strings
  - English: $p_1 > p_2 > p_3 > p_4$

P_1 = P("a quick brown dog")

P_2 = P("dog quick a brown")

P_3 = P("un chien quick brown")

P_4 = P("un chien brun rapide")

A dog (French to english)

  - … depends on what "language" we are modeling
  - in most of IR we will have $p_1 == p_2$
  - for some applications we will want p_3 to be highly probable

Google N-Grams

File sizes: approx. 24 GB compressed (gzip'ed) text files

Number of tokens: 1,024,908,267,229
Number of sentences: 95,119,665,584
Number of unigrams: 13,588,391
Number of bigrams: 314,843,401
Number of trigrams: 977,069,902
Number of fourgrams: 1,313,818,354
Number of fivegrams: 1,176,470,663

Example from the 4-Gram data

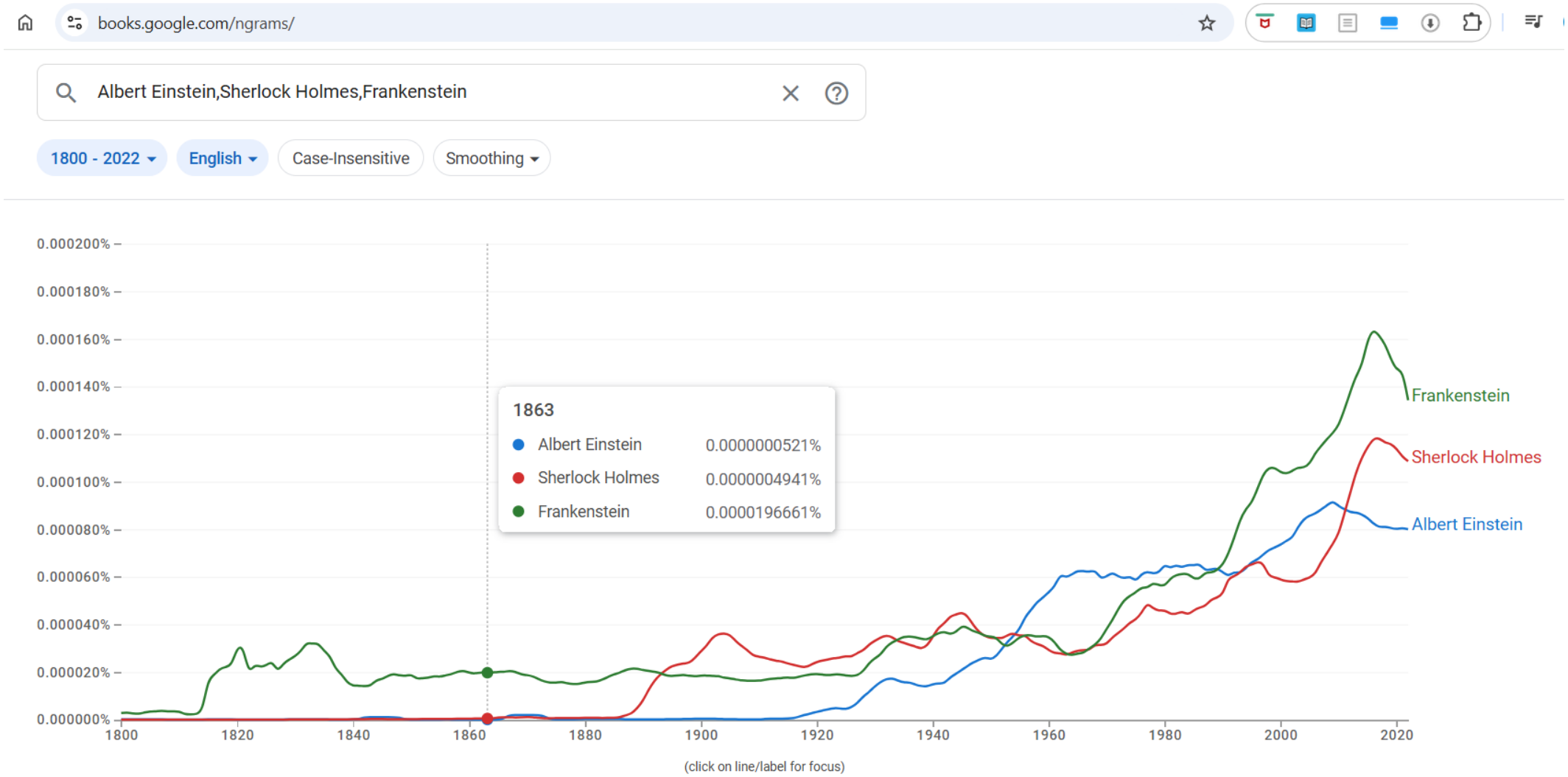
serve as the inspector 66
serve as the inspiration 1390
serve as the installation 136
serve as the institute 187
serve as the institution 279
serve as the institutional 461

3-Gram data, 2-Gram and Unigram data stats

needed to estimate probabilities.....

https://research.google/blog/all-our-n-gram-are-belong-to-you/

Google N-Gram Viewer



Raw dataset: https://storage.googleapis.com/books/ngrams/books/datasetsv3.html

https://library.mit.edu.au/c.php?g=967412&p=7032571

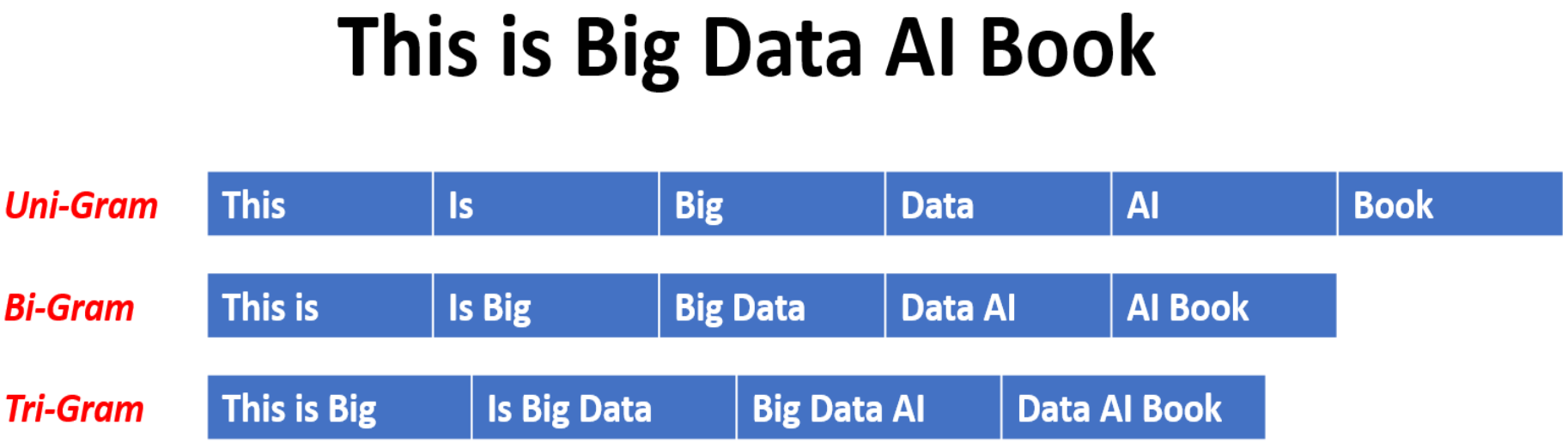
https://books.google.com/ngrams/info

N-gram model

•Given a sequence of N-1 words, an N-gram model predicts the most probable word that might follow this sequence.
•It's a probabilistic model that's trained on a corpus of text. Such a model is useful in many NLP applications including speech recognition, machine translation and predictive text input.

•An N-gram model is built by counting how often word sequences occur in corpus text and then estimating the probabilities.
•Since a simple N-gram model has limitations, improvements are often made via smoothing, interpolation and backoff

An N-gram model is one type of a Language Model (LM), which is about finding the probability distribution over word sequences



❖Consider two sentences: "There was heavy rain" vs. "There was heavy flood". From experience, we know that the former sentence sounds better. An N-gram model will tell us that "heavy rain" occurs much more often than "heavy flood" in the training corpus. Thus, the first sentence is more probable and will be selected by the model.

❖A model that simply relies on how often a word occurs without looking at previous words is called unigram.

❖If a model considers only the previous word to predict the current word, then it's called bigram.

❖If two previous words are considered, then it's a trigram model.

An n-gram model for the above example would calculate the following probability:

P('There was heavy rain') = P('There', 'was', 'heavy', 'rain') = P('There')P('was'|'There')P('heavy'|'There was')P('rain'|'There was heavy’)

❖Since it's impractical to calculate these conditional probabilities, using Markov assumption, we approximate this to a bigram model:

P('There was heavy rain') ~P('There')P('was'|'There')P('heavy'|'was')P('rain'|'heavy')

Applications of N- Gram

❖In speech recognition, input may be noisy and this can lead to wrong speech-to-text conversions. N-gram models can correct this based on their knowledge of the probabilities. Likewise, N-gram models are used in machine translation to produce more natural sentences in the target language.

❖When correcting for spelling errors, sometimes dictionary lookups will not help. For example, in the phrase "in about fifteen mineuts" the word 'minuets' is a valid dictionary word but it's incorrect in this context. N-gram models can correct such errors.

❖N-gram models are usually at word level. It's also been used at character level to do stemming, that is, separate the root word from the suffix. By looking at N-gram statistics, we could also classify languages or differentiate between US and UK spellings.

❖In general, many NLP applications benefit from N-gram models including part-of-speech tagging, natural language generation, word similarity, sentiment extraction and predictive text input.

❖The best way to evaluate a model is to check how well it predicts in end-to-end application testing. This approach is called extrinsic evaluation but it's time consuming and expensive.

❖An alternative approach is to define a suitable metric and evaluate independent of the application. This is called intrinsic evaluation. This doesn't guarantee application performance but it's a quick first step to check algorithmic performance.

What are some limitations of N-gram models?

❖A model trained on the works of Shakespeare will not give good predictions when applied to another genre. We need to therefore ensure that the training corpus looks similar to the test corpus.

❖There's also the problem of Out of Vocabulary (OOV) words. These are words that appear during testing but not in training. One way to solve this is to start with a fixed vocabulary and convert OOV words in training to UNK pseudo-word.

❖In one study, when applied to sentiment analysis, a bigram model outperformed a unigram model but the number of features doubled. Thus, scaling N-gram models to larger datasets or moving to a higher N needs good feature selection techniques.

❖N-gram models poorly capture longer-distance context. It's been shown that after 6-grams, performance gains are limited. Other language models such as cache LM, topic-based LM and latent semantic indexing do better.

What software tools are available to do N-gram modelling?

- In Python, NTLK has the function nltk.utils.ngrams(). A more comprehensive package is nltk.lm. Outside NLTK, the *ngram* package can compute n-gram string similarity.

Example Code

```
s = """
    Natural-language processing (NLP) is an area of
    computer science and artificial intelligence
    concerned with the interactions between computers
    and human (natural) languages.
"""
```

If we want to generate a list of bi-grams from the above sentence, the expected output would be something like below

```
[
    "natural language",
    "language processing",
    "processing nlp",
    "nlp is",
    "is an",
    "an area",
    ...
]
```

```python
import re

def generate_ngrams(s, n):
    # Convert to lowercases
    s = s.lower()

    # Replace all none alphanumeric characters with spaces
    s = re.sub(r'[^a-zA-Z0-9\s]', ' ', s)

    # Break sentence in the token, remove empty tokens
    tokens = [token for token in s.split(" ") if token != ""]

    # Use the zip function to help us generate n-grams
    # Concatentate the tokens into ngrams and return
    ngrams = zip(*[token[i:] for i in range(n)])
    return [" ".join(ngram) for ngram in ngrams]
```

Applying the above function to the sentence, with n=5, gives the following output:

```
>>> generate_ngrams(s, n=5)
['natural language processing nlp is',
 'language processing nlp is an',
 'processing nlp is an area',
 'nlp is an area of',
 'is an area of computer',
 'an area of computer science',
 'area of computer science and',
 'of computer science and artificial',
 'computer science and artificial intelligence',
 'science and artificial intelligence concerned',
 'and artificial intelligence concerned with',
 'artificial intelligence concerned with the',
 'intelligence concerned with the interactions',
 'concerned with the interactions between',
 'with the interactions between computers',
 'the interactions between computers and',
 'interactions between computers and human',
 'between computers and human natural',
 'computers and human natural languages']
```

The above function makes use of the zip function, which creates a generator that aggregates elements from multiple lists (or iterables in genera). The blocks of codes and comments below offer some more explanation of the usage:

```
# Sample sentence
s = "one two three four five"

tokens = s.split(" ")
# tokens = ["one", "two", "three", "four", "five"]

sequences = [tokens[i:] for i in range(3)]
# The above will generate sequences of tokens starting
# from different elements of the list of tokens.
# The parameter in the range() function controls
# how many sequences to generate.
#
# sequences = [
#   ['one', 'two', 'three', 'four', 'five'],
#   ['two', 'three', 'four', 'five'],
#   ['three', 'four', 'five']]
```

```
bigrams = zip(*sequences)
# The zip function takes the sequences as a list of inputs
# (using the * operator, this is equivalent to
# zip(sequences[0], sequences[1], sequences[2]).
# Each tuple it returns will contain one element from
# each of the sequences.
#
# To inspect the content of bigrams, try:
# print(list(bigrams))
# which will give the following:
#
# [
#   ('one', 'two', 'three'),
#   ('two', 'three', 'four'),
#   ('three', 'four', 'five')
# ]
#
```

```
>>>from nltk.tag import UnigramTagger
>>>from nltk.tag import DefaultTagger
>>>from nltk.tag import BigramTagger
>>>from nltk.tag import TrigramTagger
# we are dividing the data into a test and train to evaluate our taggers.
>>>train_data = brown_tagged_sents[:int(len(brown_tagged_sents) * 0.9)]
>>>test_data = brown_tagged_sents[int(len(brown_tagged_sents) * 0.9):]
>>>unigram_tagger = UnigramTagger(train_data,backoff=default_tagger)
>>>print unigram_tagger.evaluate(test_data)   // Score the accuracy of the tagger
0.826195866853
>>>bigram_tagger = BigramTagger(train_data, backoff=unigram_tagger)
>>>print bigram_tagger.evaluate(test_data) // Score the accuracy of the tagger
0.835300351655
>>>trigram_tagger = TrigramTagger(train_data,backoff=bigram_tagger)
>>>print trigram_tagger.evaluate(test_data) // Score the accuracy of the tagger
0.83327713281
```

What is Backoff Tagging? It is one of the most important features of **SequentialBackoffTagger** as it allows to combine the taggers together. The advantage of doing this is that if a tagger doesn't know about the tagging of a word, then it can pass this tagging task to the next backoff tagger. If that one can't do it, it can pass the word on to the next backoff tagger, and so on until there are no backoff taggers left to check.

Using NLTK

- Instead of using pure Python functions, we can also get help from some natural language processing libraries such as the Natural Language Toolkit (NLTK). In particular, nltk has the ngrams function that returns a generator of n-grams given a tokenized sentence.

```
import re
from nltk.util import ngrams


s = s.lower()
s = re.sub(r'[^a-zA-Z0-9\s]', ' ', s)
tokens = [token for token in s.split(" ") if token != ""]
output = list(ngrams(tokens, 5))
```

References

- NLP Demystified 5: Basic Bag-of-Words and Measuring Document Similarity
- Lecture 9 : N-Gram Language Models by Pawan Goyal IIT Kharagpur NPTEL
- NLP Demystified 8: Text Classification With Naive Bayes (+ precision and recall)