Word Completion

Thomas Graf

Stony Brook University
lin120@thomasgraf.net

LIN 220, Spring 2019
Lecture 2

Word completion: a simple problem(?)

- When you write something on your phone,
  it automatically suggests words while you're typing them.
- Seems easy, but it's **fairly intricate**.

Key Insights/Skills

- transition probabilities with n-grams
- "More than 1, but less than 5."
- efficient data structures: tries/prefix trees

Attempt 1: simple lookup

- To make suggestions for completions, we only need an English/German/…dictionary.

```python
# load English dictionary from nltk package
from nltk.corpus import words
# and see if "test" is in the list
"test" in words.words()
>>> True
```

- Given word **w**, the set of possible completions for **w** consists of all listed words that start with **w**:

$$\text{completions}(\mathbf{w}) = \{\mathbf{w'} \mid \mathbf{w'} \text{ is a word of English, and } \mathbf{w'} \text{ starts with } \mathbf{w}\}$$

Attempt 1 [cont.]

- completions(**w**) is easily computed in Python.

```python
def completions(word, wordlist):
    """Return set of all known completions for word."""
    return {comp for comp in wordlist
            if comp.startswith(word)}
```

```python
completions("testing", words.words())
>>> {"testing", "testingly"}
```

Evaluating attempt 1

- Is this a **good solution?**

Pro	**Contra**

```
    completions("test", words.words())
    >>> {"test", "testa", "testable", "testacean", ...}
```

Evaluating attempt 1

- Is this a **good solution?**

Pro	**Contra**
simple	

```
completions("test", words.words())
>>> {"test", "testa", "testable", "testacean", ...}
```

Evaluating attempt 1

- Is this a **good solution?**

Pro	Contra
simple	too many completions

```
    completions("test", words.words())
    >>> {"test", "testa", "testable", "testacean", ...}
```

Evaluating attempt 1

- Is this a **good solution?**

Pro	Contra
simple	too many completions unlikely completions

```
    completions("test", words.words())
    >>> {"test", "testa", "testable", "testacean", ...}
```

Improving attempt 1

- We can limit the number of suggestions, if we use a **list** instead of a **set**.

```python
def completions(word, wordlist):
    """Return list of all known completions for word."""
    return [comp for comp in wordlist
            if comp.startswith(word)]

completions("test", words.words())[:3]
>>> ["test", "testa", "testable"]
```

- But this still includes unlikely completions like *testa*.
- We need **probabilities**!

Probability = frequency (?)

- We only want to show the **most likely completions**.
- But what is most likely?
- **Naive proposal:** the most frequent word!

Probability = frequency (?)

- We only want to show the **most likely completions**.
- But what is most likely?
- **Naive proposal:** the most frequent word!

Um, but how do we figure out what is most common?

How to find common words

- **Task:** determine which words are common
- **Solution:**
  1. Collect sufficiently large sample of texts
  2. For each word (**type**), count how often it occurs in the entire sample (= its number of **tokens**).
  3. Calculate the **frequency** of the word in the sample:

$$\text{freq}(\textit{word}, \textit{sample}) = \frac{\text{number of tokens of } \textit{word}}{\text{word length of whole } \textit{sample}}$$

Types vs tokens

We have to distinguish **word types** (*a*, *the*, *Mary*, *red*, . . .) from their **word tokens**, which are the instances of a specific word type. For instance, the type “word” has 4 tokens in this box.

How to find common words

- **Task:** determine which words are common
- **Solution:**
  1. Collect sufficiently large sample of texts
  2. For each word (**type**), count how often it occurs in the entire sample (= its number of **tokens**).
  3. Calculate the **frequency** of the word in the sample:

$$\text{freq}(\textit{word}, \textit{sample}) = \frac{\text{number of tokens of } \textit{word}}{\text{word length of whole } \textit{sample}}$$

Types vs tokens

We have to distinguish **word types** (*a*, *the*, *Mary*, *red*, . . .) from their **word tokens**, which are the instances of a specific word type. For instance, the type "word" has 4 tokens in this box.

How to find common words

- **Task:** determine which words are common
- **Solution:**
  1. Collect sufficiently large sample of texts
  2. For each word (**type**), count how often it occurs in the entire sample (= its number of **tokens**).
  3. Calculate the **frequency** of the word in the sample:

$$\text{freq}(\textit{word}, \textit{sample}) = \frac{\text{number of tokens of } \textit{word}}{\text{word length of whole } \textit{sample}}$$

Types vs tokens

We have to distinguish **word types** (*a*, *the*, *Mary*, *red*, ...) from their **word tokens**, which are the instances of a specific word type. For instance, the type "word" has 4 tokens in this box.

Example calculation

Sample: 1000 words long

Words: be, bed, bee, bell

Type	be	bed	bee	bell
Tokens	**13**	**2**	**0**	**3**

$$\text{freq}(\text{be}) = \frac{\mathbf{13}}{1000} = 1.3\% \qquad \text{freq}(\text{bee}) = \frac{\mathbf{0}}{1000} = 0.0\%$$

$$\text{freq}(\text{bed}) = \frac{\mathbf{2}}{1000} = 0.2\% \qquad \text{freq}(\text{bell}) = \frac{\mathbf{3}}{1000} = 0.3\%$$

Example calculation

Sample: 1000 words long

Words: be, bed, bee, bell

Type	be	bed	bee	bell
Tokens	**13**	**2**	**0**	**3**

$$\text{freq}(\text{be}) = \frac{\mathbf{13}}{1000} = 1.3\% \qquad \text{freq}(\text{bee}) = \frac{\mathbf{0}}{1000} = 0.0\%$$

$$\text{freq}(\text{bed}) = \frac{\mathbf{2}}{1000} = 0.2\% \qquad \text{freq}(\text{bell}) = \frac{\mathbf{3}}{1000} = 0.3\%$$

Ordered completions for *be*:

Example calculation

Sample: 1000 words long

Words: be, bed, bee, bell

Type	be	bed	bee	bell
Tokens	**13**	**2**	**0**	**3**

$$\text{freq(be)} = \frac{\mathbf{13}}{1000} = 1.3\% \qquad \text{freq(bee)} = \frac{\mathbf{0}}{1000} = 0.0\%$$

$$\text{freq(bed)} = \frac{\mathbf{2}}{1000} = 0.2\% \qquad \text{freq(bell)} = \frac{\mathbf{3}}{1000} = 0.3\%$$

Ordered completions for *be*: be

Example calculation

Sample: 1000 words long

Words: be, bed, bee, bell

Type	be	bed	bee	bell
Tokens	**13**	**2**	**0**	**3**

$$\text{freq(be)} = \frac{\mathbf{13}}{1000} = 1.3\% \qquad \text{freq(bee)} = \frac{\mathbf{0}}{1000} = 0.0\%$$

$$\text{freq(bed)} = \frac{\mathbf{2}}{1000} = 0.2\% \qquad \text{freq(bell)} = \frac{\mathbf{3}}{1000} = 0.3\%$$

Ordered completions for *be*: be, bell

Example calculation

Sample: 1000 words long **Words:** be, bed, bee, bell

Type	be	bed	bee	bell
Tokens	**13**	**2**	**0**	**3**

$$\text{freq(be)} = \frac{\mathbf{13}}{1000} = 1.3\% \qquad \text{freq(bee)} = \frac{\mathbf{0}}{1000} = 0.0\%$$

$$\text{freq(bed)} = \frac{\mathbf{2}}{1000} = 0.2\% \qquad \text{freq(bell)} = \frac{\mathbf{3}}{1000} = 0.3\%$$

Ordered completions for *be*: be, bell, bed

Example calculation

Sample: 1000 words long

Words: be, bed, bee, bell

Type	be	bed	bee	bell
Tokens	**13**	**2**	**0**	**3**

$$\text{freq}(\text{be}) = \frac{\mathbf{13}}{1000} = 1.3\% \qquad \text{freq}(\text{bee}) = \frac{\mathbf{0}}{1000} = 0.0\%$$

$$\text{freq}(\text{bed}) = \frac{\mathbf{2}}{1000} = 0.2\% \qquad \text{freq}(\text{bell}) = \frac{\mathbf{3}}{1000} = 0.3\%$$

Ordered completions for *be*: be, bell, bed, bee

Types of corpora

- **Corpus** = large, structured collections of texts

  mono-/multilingual just one language, or many?

  annotated not just text, but additional annotations (e.g. tags for part of speech, syntax trees)

Some common corpora in Python's NLTK

Brown 1 million words, tagged, 500 samples across 15 genres (fiction, news paper, . . .)

Gutenberg 1.8 million words, 18 classic texts of fiction

Penn 40k words, tagged and parsed

Reuters 1.3 million words, news documents

Switchboard 36 phone calls, fully transcribed and parsed

Wordlist 960k words (no repetitions) and 20k affixes for 8 languages

Getting probabilities from the corpus

```python
from nltk.corpus import brown
from collections import Counter

# load Brown corpus as sequence of words
brown_text = brown.words()
# total number of words = length of text
total = len(brown_text)
# calculate counts
brown_counts = Counter(brown_text)
# convert counts to frequencies
for word in brown_counts:
    brown_counts[word] /= total
```

- Alright, we have frequencies for each word.
  Now what?

Ordering completions by frequency

- For a good user experience, completions should appear in **descending order of probability**.

```python
def completions(word, counts):
    """Return set of all known completions for word.

    The completions are sorted by frequency,
    in descending order.
    """
    comps = [comp for comp in counts
             if comp.startswith(word)]
    return sorted(comps,
                  key=counts.get,
                  reverse=True)
```

```python
completions("test", words.words(), brown_counts)
>>> ["test", "testimony", "tested", "testing", ...]
```

Summary for revised attempt 1

1. **Needed resources:** corpus
2. Compute frequencies for all words in corpus
3. Look up possible completions for user input
4. Sort completions by their frequency

Great, we're done, right?! Not quite. . .

Probability $\neq$ word frequency

- The probability of a word isn't fixed, it varies by **context**.

Example

	tested	**testing**	**testimony**
I have	hi	low	mid
I have been	hi	hi	low
I have the	low	low	hi

- The frequency of words is not enough,
  we need frequencies of sequences of words $\Rightarrow$ **n-grams**

Defining n-grams

n-gram a contiguous sequence of n words

n	Name	Example
1	unigram	John
2	bigram	John to
3	trigram	John to be
4	4-gram	John to be in
5	5-gram	John to be in the

Example

String

John and Marie are not Bill and Sue

Defining n-grams

n-gram a contiguous sequence of n words

n	Name	Example
1	unigram	John
2	bigram	John to
3	trigram	John to be
4	4-gram	John to be in
5	5-gram	John to be in the

Example

String

John and Marie are not Bill and Sue

Defining n-grams

n-gram a contiguous sequence of n words

n	Name	Example
1	unigram	John
2	bigram	John to
3	trigram	John to be
4	4-gram	John to be in
5	5-gram	John to be in the

Example

String

John and Marie are not Bill and Sue

Defining n-grams

n-gram a contiguous sequence of n words

n	Name	Example
1	unigram	John
2	bigram	John to
3	trigram	John to be
4	4-gram	John to be in
5	5-gram	John to be in the

Example

String

John and Marie are not Bill and Sue

Defining n-grams

n-gram a contiguous sequence of n words

n	Name	Example
1	unigram	John
2	bigram	John to
3	trigram	John to be
4	4-gram	John to be in
5	5-gram	John to be in the

Example

String

John and Marie **are not** Bill and Sue

Defining n-grams

n-gram a contiguous sequence of n words

n	Name	Example
1	unigram	John
2	bigram	John to
3	trigram	John to be
4	4-gram	John to be in
5	5-gram	John to be in the

Example

String

John and Marie are not Bill and Sue

Defining n-grams

n-gram a contiguous sequence of n words

n	Name	Example
1	unigram	John
2	bigram	John to
3	trigram	John to be
4	4-gram	John to be in
5	5-gram	John to be in the

Example

String

John and Marie are not Bill and Sue

Defining n-grams

n-gram a contiguous sequence of n words

n	Name	Example
1	unigram	John
2	bigram	John to
3	trigram	John to be
4	4-gram	John to be in
5	5-gram	John to be in the

Example

String

John and Marie are not Bill and Sue

Defining n-grams

n-gram a contiguous sequence of n words

n	Name	Example
1	unigram	John
2	bigram	John to
3	trigram	John to be
4	4-gram	John to be in
5	5-gram	John to be in the

Example

String

John and Marie are not Bill and Sue

Defining n-grams

n-gram a contiguous sequence of n words

n	Name	Example
1	unigram	John
2	bigram	John to
3	trigram	John to be
4	4-gram	John to be in
5	5-gram	John to be in the

Example

String

John and Marie are not Bill and Sue

Defining n-grams

n-gram a contiguous sequence of n words

n	Name	Example
1	unigram	John
2	bigram	John to
3	trigram	John to be
4	4-gram	John to be in
5	5-gram	John to be in the

Example

String

John and Marie are not Bill and Sue

Defining n-grams

n-gram a contiguous sequence of n words

n	Name	Example
1	unigram	John
2	bigram	John to
3	trigram	John to be
4	4-gram	John to be in
5	5-gram	John to be in the

Example

String

John and Marie are not Bill and Sue

Defining n-grams

n-gram a contiguous sequence of n words

n	Name	Example
1	unigram	John
2	bigram	John to
3	trigram	John to be
4	4-gram	John to be in
5	5-gram	John to be in the

Example

String

John and Marie are not Bill and Sue

Defining n-grams

n-gram a contiguous sequence of n words

n	Name	Example
1	unigram	John
2	bigram	John to
3	trigram	John to be
4	4-gram	John to be in
5	5-gram	John to be in the

Example

String

John and Marie are not Bill and Sue

Defining n-grams

n-gram a contiguous sequence of n words

n	Name	Example
1	unigram	John
2	bigram	John to
3	trigram	John to be
4	4-gram	John to be in
5	5-gram	John to be in the

Example

String

John and Marie are not Bill and Sue

Defining n-grams

n-gram a contiguous sequence of n words

n	Name	Example
1	unigram	John
2	bigram	John to
3	trigram	John to be
4	4-gram	John to be in
5	5-gram	John to be in the

Example

String

John and Marie are not Bill and Sue

Defining n-grams

n-gram a contiguous sequence of n words

n	Name	Example
1	unigram	John
2	bigram	John to
3	trigram	John to be
4	4-gram	John to be in
5	5-gram	John to be in the

Example

String

John and Marie are not Bill and Sue

Defining n-grams

n-gram a contiguous sequence of n words

n	**Name**	**Example**
1	unigram	John
2	bigram	John to
3	trigram	John to be
4	4-gram	John to be in
5	5-gram	John to be in the

Example

String

John and Marie are not Bill and Sue

Defining n-grams

n-gram a contiguous sequence of n words

n	Name	Example
1	unigram	John
2	bigram	John to
3	trigram	John to be
4	4-gram	John to be in
5	5-gram	John to be in the

Example

String

John and Marie are not Bill and Sue

Defining n-grams

n-gram a contiguous sequence of n words

n	Name	Example
1	unigram	John
2	bigram	John to
3	trigram	John to be
4	4-gram	John to be in
5	5-gram	John to be in the

Example

String

John and Marie are not Bill and Sue

Defining n-grams

n-gram a contiguous sequence of n words

n	**Name**	**Example**
1	unigram	John
2	bigram	John to
3	trigram	John to be
4	4-gram	John to be in
5	5-gram	John to be in the

Example

String

John and Marie are not Bill and Sue

Defining n-grams

n-gram a contiguous sequence of n words

n	Name	Example
1	unigram	John
2	bigram	John to
3	trigram	John to be
4	4-gram	John to be in
5	5-gram	John to be in the

Example

String

John and Marie are not Bill and Sue

Defining n-grams

n-gram a contiguous sequence of n words

n	**Name**	**Example**
1	unigram	John
2	bigram	John to
3	trigram	John to be
4	4-gram	John to be in
5	5-gram	John to be in the

Example

String

John and Marie are not Bill and Sue

Frequencies for n-grams

Frequencies can be computed for n-grams, too.

Example: Calculating Bigram Frequencies

- **String**

  when buffalo buffalo buffalo buffalo buffalo buffalo

- **Bigram token list**

- **Bigram counts and frequencies**

Frequencies for n-grams

Frequencies can be computed for n-grams, too.

Example: Calculating Bigram Frequencies

- **String**

  when buffalo buffalo buffalo buffalo buffalo buffalo

- **Bigram token list**

  when buffallo,

- **Bigram counts and frequencies**

Frequencies for n-grams

Frequencies can be computed for n-grams, too.

Example: Calculating Bigram Frequencies

- **String**

  when buffalo buffalo buffalo buffalo buffalo buffalo

- **Bigram token list**

  when buffallo, buffalo buffalo,

- **Bigram counts and frequencies**

Frequencies for n-grams

Frequencies can be computed for n-grams, too.

Example: Calculating Bigram Frequencies

- **String**

  when buffalo buffalo buffalo buffalo buffalo buffalo

- **Bigram token list**
  when buffallo, buffalo buffalo, buffalo buffalo,

- **Bigram counts and frequencies**

Frequencies for n-grams

Frequencies can be computed for n-grams, too.

Example: Calculating Bigram Frequencies

- **String**

  when buffalo buffalo buffalo buffalo buffalo buffalo

- **Bigram token list**
  when buffallo, buffalo buffalo, buffalo buffalo, buffalo buffalo,

- **Bigram counts and frequencies**

Frequencies for n-grams

Frequencies can be computed for n-grams, too.

Example: Calculating Bigram Frequencies

- **String**

  when buffalo buffalo buffalo buffalo buffalo buffalo

- **Bigram token list**
  when buffallo, buffalo buffalo, buffalo buffalo, buffalo buffalo, buffalo buffalo,

- **Bigram counts and frequencies**

Frequencies for n-grams

Frequencies can be computed for n-grams, too.

Example: Calculating Bigram Frequencies

- **String**

  when buffalo buffalo buffalo buffalo buffalo buffalo

- **Bigram token list**
  when buffallo, buffalo buffalo, buffalo buffalo, buffalo buffalo, buffalo buffalo, buffalo buffalo
- **Bigram counts and frequencies**

Frequencies for n-grams

Frequencies can be computed for n-grams, too.

Example: Calculating Bigram Frequencies

- **String**

  when buffalo buffalo buffalo buffalo buffalo buffalo

- **Bigram token list**

  when buffallo, buffalo buffalo, buffalo buffalo, buffalo buffalo, buffalo buffalo, buffalo buffalo

- **Bigram counts and frequencies**

  1. when buffalo: 1 $\Rightarrow \frac{1}{6} = 16.7\%$
  2. buffalo buffalo: 5 $\Rightarrow \frac{5}{6} = 83.3\%$

Adapting the strategy

1. **Needed resources:** corpus
2. Convert corpus to list of n-gram tokens
3. Compute frequencies for all n-grams
4. Look up possible completions for user input
5. Look at previous $n-1$ words.
6. Sort completions by n-gram probability

Example

- **Trigram frequencies**

bus is late	30%	train is late	15%
bus is lovely	25%	train is lovely	8%
bus is lazy	10%	train is lazy	2%

- **Input**

  I will text you if the train is l

- **Sorted completions**

Adapting the strategy

1. **Needed resources:** corpus
2. Convert corpus to list of n-gram tokens
3. Compute frequencies for all n-grams
4. Look up possible completions for user input
5. Look at previous $n - 1$ words.
6. Sort completions by n-gram probability

Example

- **Trigram frequencies**

bus is late	30%	train is late	15%
bus is lovely	25%	train is lovely	8%
bus is lazy	10%	train is lazy	2%

- **Input**

  I will text you if the train is l

- **Sorted completions**

  late

Adapting the strategy

1. **Needed resources:** corpus
2. Convert corpus to list of n-gram tokens
3. Compute frequencies for all n-grams
4. Look up possible completions for user input
5. Look at previous $n - 1$ words.
6. Sort completions by n-gram probability

Example

- **Trigram frequencies**

bus is late	30%	train is late	15%
bus is lovely	25%	train is lovely	8%
bus is lazy	10%	train is lazy	2%

- **Input**

  I will text you if the train is l

- **Sorted completions**

  late, lovely

Adapting the strategy

1. **Needed resources:** corpus
2. Convert corpus to list of n-gram tokens
3. Compute frequencies for all n-grams
4. Look up possible completions for user input
5. Look at previous $n-1$ words.
6. Sort completions by n-gram probability

Example

- **Trigram frequencies**

bus is late	30%	train is late	15%
bus is lovely	25%	train is lovely	8%
bus is lazy	10%	train is lazy	2%

- **Input**

  I will text you if the train is l

- **Sorted completions**

  late, lovely, lazy

How it is done: The easy generalization step

```
def bigrams(text):
    """Convert text to list of bigram tokens."""
    return [text[n:n+2] for n in range(len(text) - 1)]


brown.words()[:5]
>>> ["The", "Fulton", "County", "Grand", "Jury"]
bigrams(brown.words())[:3]
>>> [["The", "Fulton"], ["Fulton", "County"],
     ["County", "Grand"]]
```

```
brown_bigrams = bigrams(brown.words())
total = len(brown_bigrams)
brown_bicounts = Counter(brown_bigrams)
for bigram in brown_bicounts:
    brown_bicounts[bigram] /= total
```

How it is done: The trickier part

```
bigram_completions(word, previous_word, counts):
# set of all compatible bigrams
comps = [comp for comp in counts
         if comp[:-1] == previous_word and
            comp[-1].startswith(word)]
# sort the bigram completions
ordered_ngrams = sorted(comps,
                        key=counts.get,
                        reverse=True)
# only keep last word of each bigram
return [ngram[-1] for ngram in ordered_ngrams]
```

Linguistic evaluation

- We now use the local context to choose word completions.

The n-Gram Hypothesis

One can reliably predict the next word based on the **preceding $n-1$ words**.

- The n-gram hypothesis is **not quite true**, though.

Context matters a lot

- Words do not exist in a vacuum.

Example

The word *hypothyroidism* is rarely heard or seen,
unless you're an endocrinologist.

- Word choices depend greatly on genre, target audience, age of the speaker, and so on.

Common fixes

- Use frequencies from the appropriate genre (**BUT:** must be able to reliably determine genre first)
- Learn directly from the use case (e.g. analyze all text messages on phone)

Long-distance dependencies in language

- Word choice can be influenced by words that are very far away.

Subject-verb agreement

- The key to the cabinet **is** on the table.
- The keys to the cabinets **are** on the table.
- The key to the cabinets **is**/**are** on the table.
- The keys to the cabinet **is**/**are** on the table.

- Psycholinguistic observation: humans get those “wrong” too
- Potential fix: **larger n-grams**

How long can n-grams be?

- It is tempting to move to longer and longer n-grams in order to handle long-distance dependencies.
- But this has **two problems**:

  data sparsity longer n-grams require too much data

  storage needs longer n-grams require lots of storage
- Data sparsity is much more severe than storage needs.

Sparse data: A simple calculation

Words	**bigrams**	**trigrams**	**5-grams**	**6-grams**
10	100	1000	10,000	100,000
100	10,000	1,000,000	10,000,000,000	**1,000,000,000,000**
10,000	10^8	$\mathbf{10^{12}}$	10^{20}	10^{24}
25,000	6.3×10^8	1.6×10^{13}	9.7×10^{21}	2.4×10^{26}

Some comparison values

4.3×10^{17}	number of seconds since the Big Bang
5×10^{22}	number of stars in observable universe
10^{24}	milliliters of water in the Earth's oceans
8.8×10^{26}	diameter of observable universe, in meters
10^{80}	number of atoms in observable universe

- **Conclusion:** with large n, most n-grams are **never encountered** in a corpus ⇒ frequency 0

Things get worse: A more realistic estimate

- The Linux dictionary american-english-insane has 650,000 entries.
- This makes the numbers much worse.
  Can you guess how many 5-grams there are then?

Things get worse: A more realistic estimate

- The Linux dictionary american-english-insane has 650,000 entries.
- This makes the numbers much worse.
  Can you guess how many 5-grams there are then?

$$\text{116 octillion} \approx \mathbf{10^{29}}$$

Things get worse: A more realistic estimate

- The Linux dictionary american-english-insane has 650,000 entries.
- This makes the numbers much worse.
  Can you guess how many 5-grams there are then?

$$\text{116 octillion} \approx \mathbf{10^{29}}$$

10^{29} is larger than the number of shotglasses it takes to drain the Earth's oceans over 2000 times.

Trick 1: Stemming and lemmatization

- Removing inflectional markers reduces number of words
- Two solutions:
  - stemming is quick and dirty
  - lemmatization is accurate but complex

stemming cut off word ends that look like inflection

Example

- cats ⇒ cat
- tasks ⇒ task (noun and verb)
- asking ⇒ ask
- meeting ⇒ meet (**noun and verb**)
- staging ⇒ stag

Trick 1: Stemming and lemmatization [cont.]

lemmatization stemming with context information

Example

- cats ⇒ cat
- tasks ⇒ task (noun and verb)
- asking ⇒ ask
- meeting ⇒ meet (**only verb**)
- staging ⇒ stag/stage

Evaluation

- Stemming/lemmatization reduces the number of words.
- But we still have at least 10,000 words and thus 10^{20} 5-grams.

Trick 2: POS tagging

- Every word has a part of speech (POS).
- If we know the POS of a word, we can use that for estimating probabilities.

Example

- **User input:** an
- **Preceding words:** to
- **Completions:** an (Det), annoint (V)
- **Suggestion:** to annoint
- **Reasoning:** even though *to an* is more common than *to annoint*, *to V* is more common than *to Det*.

Trick 3: Statistics

- **Backoff Method**
  If an n-gram has frequency 0, use the frequency of the corresponding $(n-1)$-gram.
- **Good-Turing Smoothing**
  Change frequency from 0 to a very low value while lowering high frequency values.

Evaluation

- These tricks solve the issue of n-grams with 0% frequency.
- But they do not solve the basic problem that large n-gram models are incredibly data hungry.

Another problem: Storage

- While storage space is increasing with each year,
  you don't want to waste GB on storing n-gram frequencies.
- We want a storage solution that is
  - compact,
  - easy to expand,
  - **fast to search**.

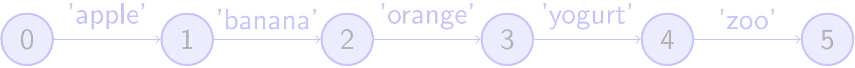
Searching through a list

List search is a much studied problem.

Simplest Algorithm: Linear Search

- Start at beginning of list.
- Move from left to right until the wanted item has been found.
- If no match is found before end of list, item is not in list.

Searching for yogurt



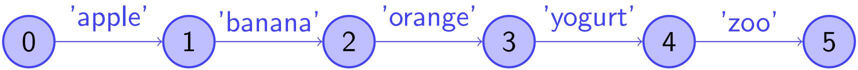
Searching through a list

List search is a much studied problem.

Simplest Algorithm: Linear Search

- Start at beginning of list.
- Move from left to right until the wanted item has been found.
- If no match is found before end of list, item is not in list.

Searching for yogurt



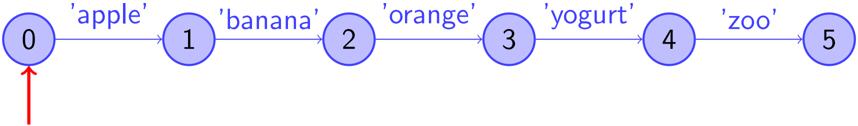
Searching through a list

List search is a much studied problem.

Simplest Algorithm: Linear Search

- Start at beginning of list.
- Move from left to right until the wanted item has been found.
- If no match is found before end of list, item is not in list.

Searching for yogurt



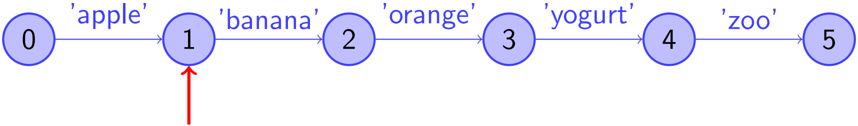
Searching through a list

List search is a much studied problem.

Simplest Algorithm: Linear Search

- Start at beginning of list.
- Move from left to right until the wanted item has been found.
- If no match is found before end of list, item is not in list.

Searching for yogurt



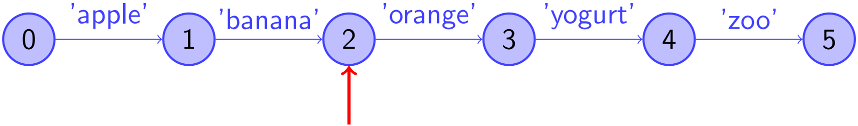
Searching through a list

List search is a much studied problem.

Simplest Algorithm: Linear Search

- Start at beginning of list.
- Move from left to right until the wanted item has been found.
- If no match is found before end of list, item is not in list.

Searching for yogurt



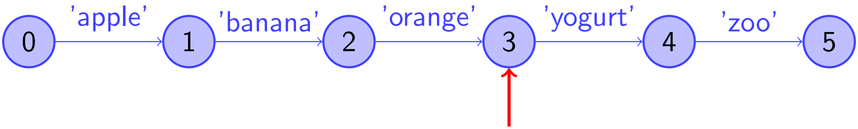
Searching through a list

List search is a much studied problem.

Simplest Algorithm: Linear Search

- Start at beginning of list.
- Move from left to right until the wanted item has been found.
- If no match is found before end of list, item is not in list.

Searching for yogurt



Evaluating linear search

- Two criteria for evaluating algorithms: correctness and efficiency
- An algorithm is **correct** if and only if it is both sound and complete.

sound	algorithm only returns correct solutions
complete	algorithm can find all correct solutions

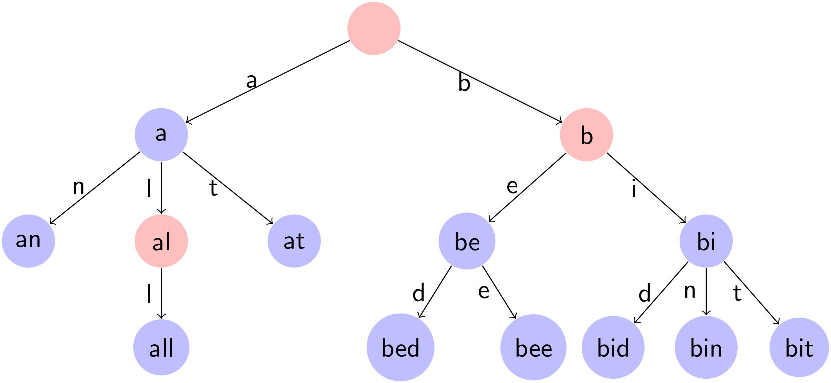
- Linear search is correct.
  - if item is not in list, linear search will not claim otherwise
  - if item is in list, linear search will find it
- But it is also slow.
  - In a list with n items, it may take n steps to find item.
  - And we have tons of n-grams!

Improving on lists: Prefix trees/Tries

Prefix Trees: The Basic Idea

Exploit the fact that many items share a common start

Example: Prefix tree for word list



Searching through a prefix tree

Query: Is word w in our dictionary?
Algorithm: follow branches that spell w; do we finish in accepting ($=$ blue) node?

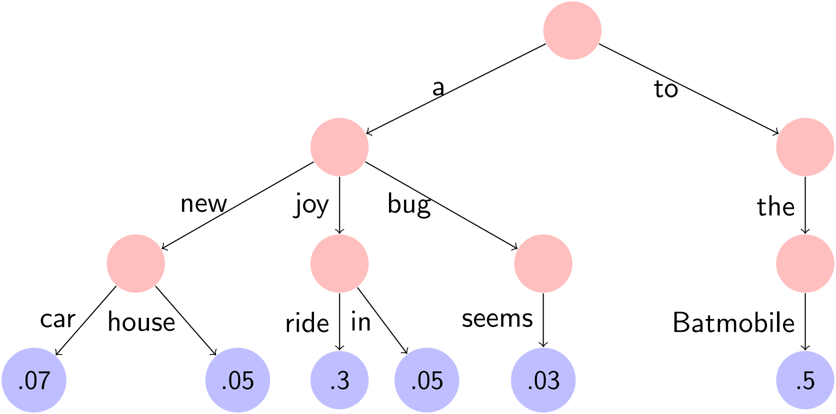
Example

1. **Query:** bit
   Search Path: b-i-t, blue ⇒ accept
2. **Query:** al
   Search Path: al, red ⇒ reject
3. **Query:** bits
   Search Path: b-i-t, stuck ⇒ reject

Prefix tree for n-grams

- A prefix tree for a list of n-grams works exactly the same way, except that each arc is **labeled with a word** instead.

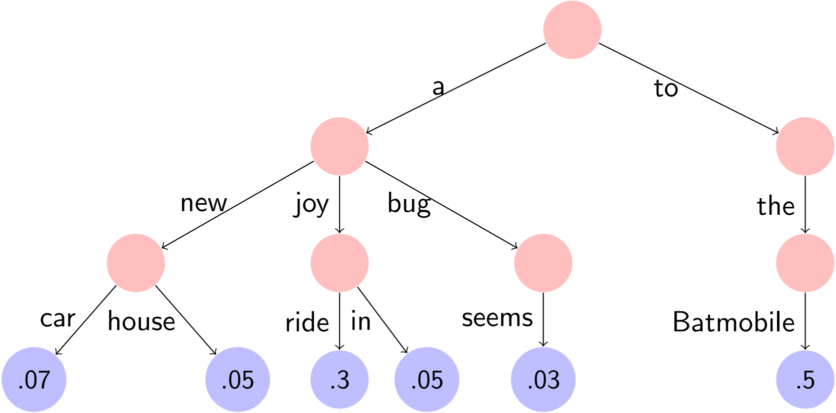
Example: Prefix tree for trigram frequencies



A mixed prefix tree

- Prefix trees can store frequencies for all $m < n$
- useful for backoff methods

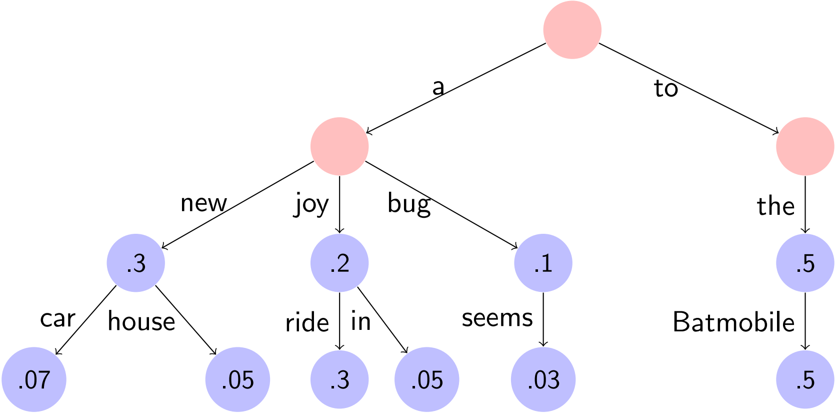
Example: Prefix tree for trigram frequencies



A mixed prefix tree

- Prefix trees can store frequencies for all $m < n$
- useful for backoff methods

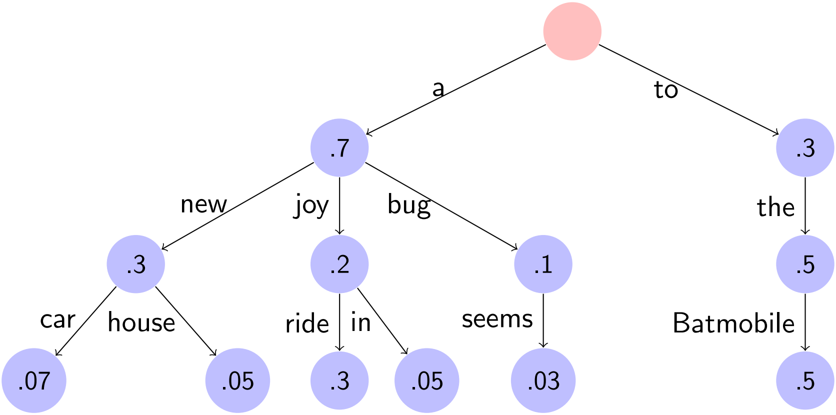
Example: Prefix tree for bigram and trigram frequencies



A mixed prefix tree

- Prefix trees can store frequencies for all $m < n$
- useful for backoff methods

Example: Prefix tree for unigram, bigram and trigram frequencies



Excursus: It's the software, stupid!

- We usually think of technological progress in terms of faster hardware, more memory, and so on.
- A good **algorithm trumps hardware** improvements.

Example

- Even the fastest current gen processor will take a long time to do linear search on a list with 18 quintillion items.
- With prefix trees it is almost instantaneous (and incremental!).
- They are also more memory efficient,
  as long as your items have lots of overlap.
- **BUT:** They are harder to implement.

Yet another problem: bad input

- What if a user makes a mistake?

Input	**Intended?**
trian	train, trying, Brian, . . .
your	your, you’re
@rdvark	aardvark

- This requires a spellchecker.
  But more on that next time. . .

Summary

A good word completion model needs **lots of components**:

- corpora from various genres
- n-grams for limited context information
- sophisticated statistics to deal with sparse data
- a fast and compact data structure for storage
- extensions:
  - stemming/lemmatizer
  - POS tagging
  - spell checker
  - efficient storage

We have barely scratched the surface of any of those.

New concepts

- corpora (common words, what types there are)
- n-grams (word = unigram)
- types VS tokens
- data sparsity
- smoothing and backoff
- stemming, lemmatization
- POS tagging

New Python concepts

```python
sorted(some_list,
       # sort in reverse?
       reverse=True,
       # what property to use for sorting
       key=some_function)

range(n)  # all numbers from 0 to n-1, in order

some_string.startswith(x)  # does some_string start with x?
```