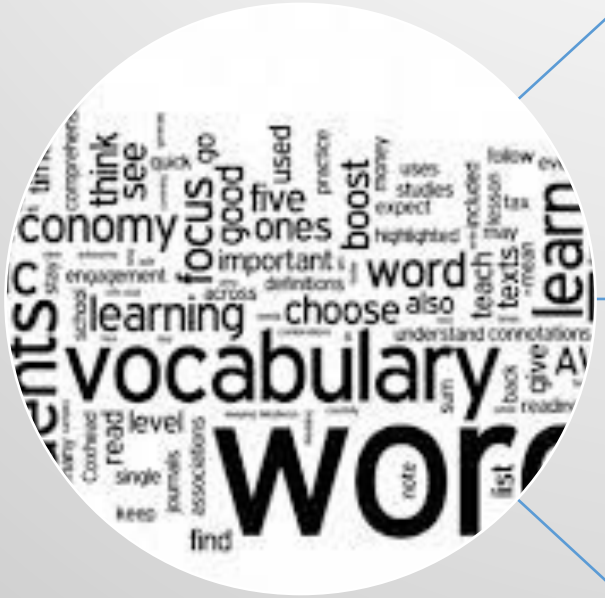
Natural Language Processing

Dr. Karen Mazidi

Part Two: Words



Topics

- ngram language models
- iterators and generators

Quizzes

- no quiz on this material

Homework

- Portfolio: Ngram language model

Before the movie begins,
please turn off your _____.

N grams

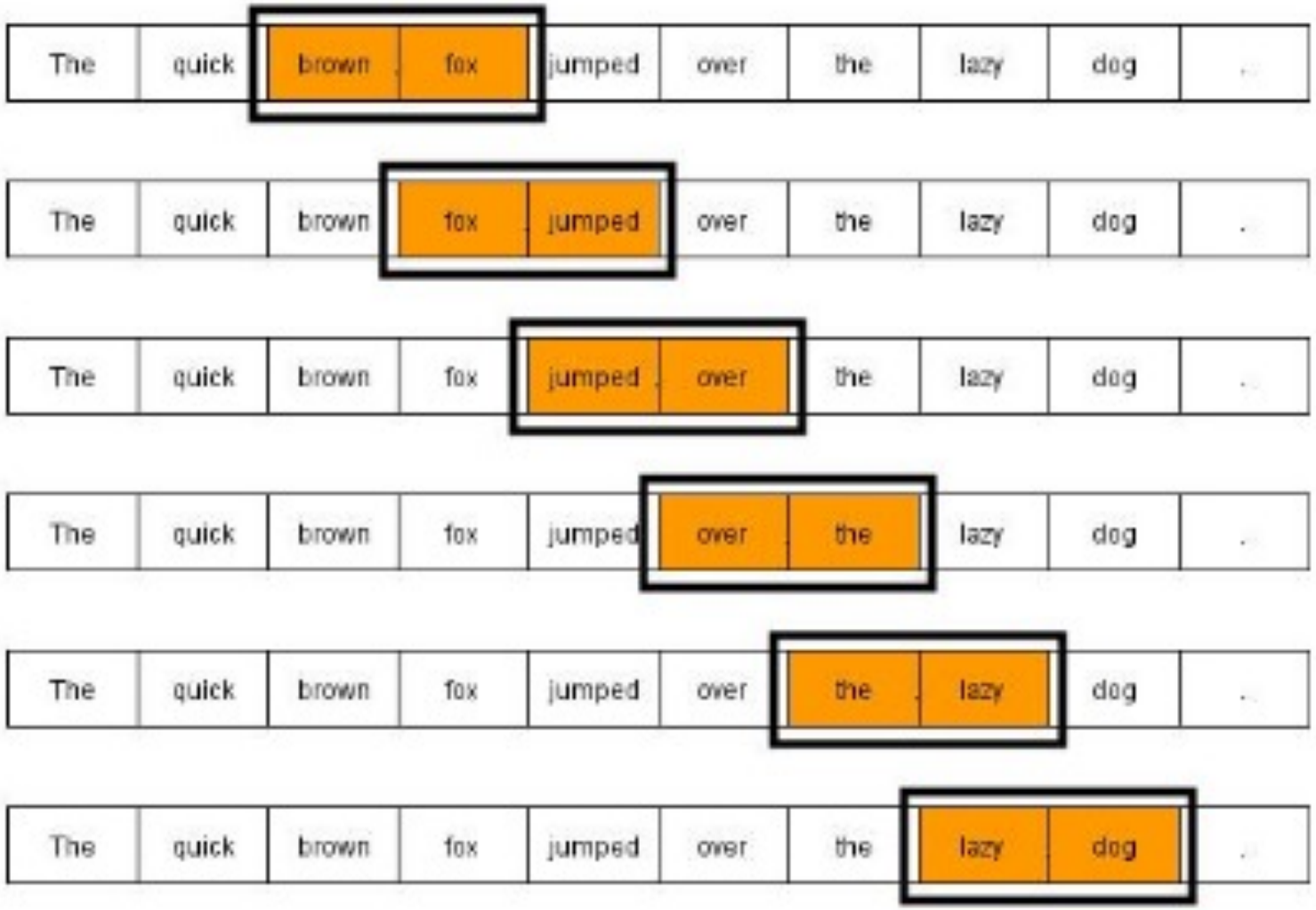
- An n-gram is a sliding window of size n over text

The cat in the hat

- Unigrams:
  - [the, cat, in, the, hat]
- Bigrams:
  - [the cat, cat in, in the, the hat]
- Trigrams:
  - [the cat in, cat in the, in the hat]
- Sometimes start and stop symbols are added

N grams

- Create a probabilistic model of language
- Choice of corpus influences the language model



Getting n-grams from text using Python

- See Notebook 8-1
- Using Python:

```
raw_text = "jack be nimble. jack be quick. jack jump over \
       the candlestick."

from nltk import word_tokenize
unigrams = word_tokenize(raw_text)

bigrams = [(unigrams[k], unigrams[k+1]) for k in range(len(unigrams)-1)]
```

Getting n-grams from text using NLTK

```
from nltk import word_tokenize
from nltk.util import ngrams

raw_text = "jack be nimble. jack be quick. jack jump over \
the candlestick"

unigrams = word_tokenize(raw_text)
bigrams = list(ngrams(unigrams, 2))

unigram_dict = {t:unigrams.count(t) for t in set(unigrams)}
bigram_dict = {b:bigrams.count(b) for b in set(bigrams)}
```

N grams form a language model

What does that mean?

```
jack be nimble.
jack be quick.
jack jump over the candlestick.
```

N-grams form a probabilistic model

- P(w1) = count of w1 / number of tokens in text
- P(w1, w2) = P(w1)P(w2|w1)

$$P(w_1, w_2) = P(w_1)P(w_2|w_1) = \frac{C(w_1)}{\Sigma_n C(w_n)} \frac{C(w_1 w_2)}{C(w_1)}$$

- P(jack, be)???
- P(jack, be) = P(jack)P(be|jack)
- = 3/14 * 2/3 = 14%

- What is the probability of ‘jack jump’?

Probability of a sequence of n words

- Sequence of 4 words:

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

- Make the Markov assumption to only look at previous word

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

which we can generalize to:

$$\Pi_{k=1}^{n} P(w_k|w_{k-1})$$

Create dicts of counts

```
from nltk import word_tokenize
from nltk.util import ngrams

raw_text = "jack be nimble. jack be quick. jack jump over \
the candlestick"

unigrams = word_tokenize(raw_text)
bigrams = list(ngrams(unigrams, 2))

unigram_dict = {t:unigrams.count(t) for t in set(unigrams)}
bigram_dict = {b:bigrams.count(b) for b in set(bigrams)}
```

The unigram dictionary entry for 'jack' is:

```
'jack': 3
```

The bigram dictionary entry for 'jack be' is:

```
('jack', 'be'): 2
```

Python Counter object

- dictionary subclass for counting hashable objects
- elements are stored as the dictionary keys and counts are stored as dictionary values
- can be faster, esp. on Windows computers

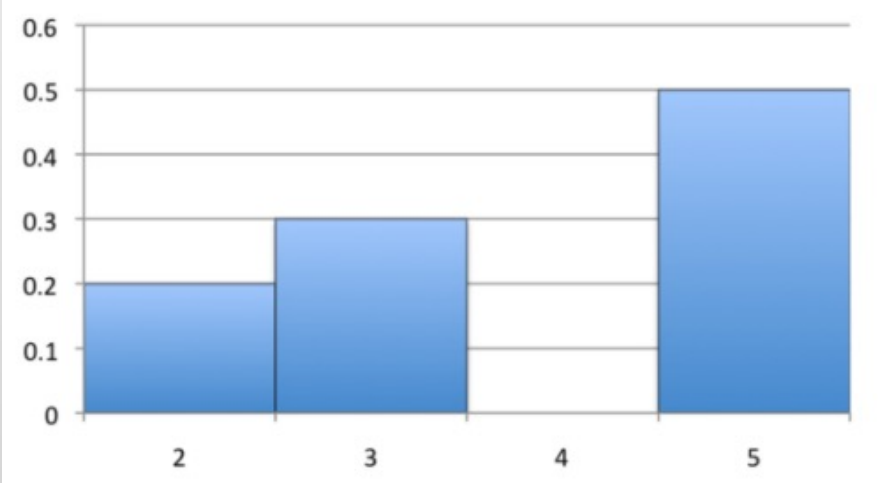
```
from collections import Counter
bigram_dict = dict(Counter(bigrams))
```

MLE

- maximum likelihood estimate – estimating the parameters of a probability distribution given some observed data
- The probabilities in an n-gram language model can be viewed as maximum likelihood estimates

smoothing

- Sparsity problem: not every possible n-gram will be in our dicts
- Smoothing fills in 0 values with a bit of probability mass



LaPlace smoothing

- Unsmoothed probability:

$$P(w_i) = \frac{C(w_i)}{N}$$

- Where N is the total number of unigrams
- Smoothed probability

$$P(w_i) = \frac{C(w_i) + 1}{N + V}$$

- Where V is the vocabulary size
- Add-1 aka LaPlace smoothing

Modified Good-Turing smoothing

- replace zero counts/probabilities with counts/probabilities of words that occur only once

$$P(w_0) = \frac{N_1}{N}$$

```python
import math

def compute_prob(text, unigram_dict, bigram_dict, N, V):
# N is the number of tokens
# V is the number of unique tokens, the vocabulary size

    unigrams_test = word_tokenize(text)
    bigrams_test = list(ngrams(unigrams_test, 2))

    p_gt = 1
    p_laplace = 1
    p_log = 0  # sometimes used to prevent overflow

    for bigram in bigrams_test:
        n = bigram_dict[bigram] if bigram in bigram_dict else 0
        n_gt = bigram_dict[bigram] if bigram in bigram_dict else 1/N
        d = unigram_dict[bigram[0]] if bigram[0] in unigram_dict else 0
        if d == 0:
            p_gt = p_gt * (1 / N)
        else:
            p_gt = p_gt * (n_gt / d)
        p_laplace = p_laplace * ((n + 1) / (d + V))
        p_log = p_log + math.log((n + 1) / (d + V))

    print("\nprobability with simplified Good-Turing is %.5f" % (p_gt))
    print("probability with laplace smoothing is %.5f" % p_laplace)
    print("log prob is %.5f == %.5f" % (p_log, math.exp(p_log)))
```

Using n-grams to generate language

- Create probability dicts

```
u_probs = {t:unigrams.count(t)/len(unigrams) for t in set(unigrams)}
b_probs = {b:bigrams.count(b)/unigrams.count(b[0]) for b in set(bigrams)}
```

Using n-grams to generate language

- A naïve approach

```
def naive_gen(start_word, u_probs, b_probs):
    phrase = [start_word]
    while phrase[-1] != '.':
        candidate_next = {k:b_probs[k] for k in b_probs
                          if k[0] == phrase[-1]}
        candidate_next = sorted(candidate_next.items(),
                         key=lambda x:x[1], reverse=True)
        if not candidate_next:
            break
        phrase += [candidate_next[0][0][1]]  # [0] = first bigram
        print(phrase)

    return phrase
```

Using n-grams to generate language

```
naive_gen('jack', u_probs, b_probs)
'jack be quick .'
```

- trigrams work better than unigrams
- higher n-grams work better still
- for best results, need a large corpus

NLTK

```
import nltk
nltkText = nltk.Text(unigrams)
nltkText.generate()
```

The output is:

```
be nimble . . jack jump over the candlestick over the candlestick jack
be quick . jack be quick . . jack be nimble . the candlestick be quick
. over the candlestick the candlestick . be nimble . jack jump over
the candlestick quick . jack be nimble . jack be quick . jack be quick
. . . jack jump over the candlestick jack be nimble . over the
```

NLTK generate over a larger corpus

```
from nltk.book import *
text4.generate()
```

The generated text is:

```
occur , and especially the truth that democratic government has innate
capacity to govern its affairs aright through the Province ceded , by
any timid forebodings of evil were not to overtake them while I
possess the property of the peaks and the exercise of free and firm on
the farm , in view that the best of my countrymen will ever find me
ready to confer their benefits on countless generations yet to make
its promise for all generations ." , remains essentially unchanged .
cost of the Rocky Mountains . abuses of an ever - expanding American
dream
```

pickle

- way to save Python data structures as binary files

```
import pickle

# pickle a dictionary
with open('my_pickle', 'wb') as handle:
    pickle.dump(my_dict, handle)

# unpack
with open('my_pickle', 'rb') as handle:
    new_dict = pickle.load(handle)
```

metrics

- How do we know we have a good language model?
- Choose n
- Evaluation:
  - Extrinsic: human annotators
  - Intrinsic: an internal metric like perplexity
- Perplexity (PP) is the inverse probability of seeing the words we observe, normalized by the number of words
- Low perplexity is ideal

$$PP(W) = P(w_1 w_2 \ldots w_N)^{-\frac{1}{N}}$$

Fun with n-grams

- https://books.google.com/ngrams
- Download n-grams for your application: https://www.ngrams.info/
- Software solutions pre-built: https://aclweb.org/aclwiki/Language_modeling_software
- More info on smoothing: https://web.stanford.edu/~jurafsky/slp3/3.pdf

iterators and generators

- A Python iterator is a container object with a countable number of elements that implements the iterator protocol
- Iterator objects implement these methods:
- `__iter__()` returns the iterator object
- `__next__()` returns the next item or `Stopiteration`

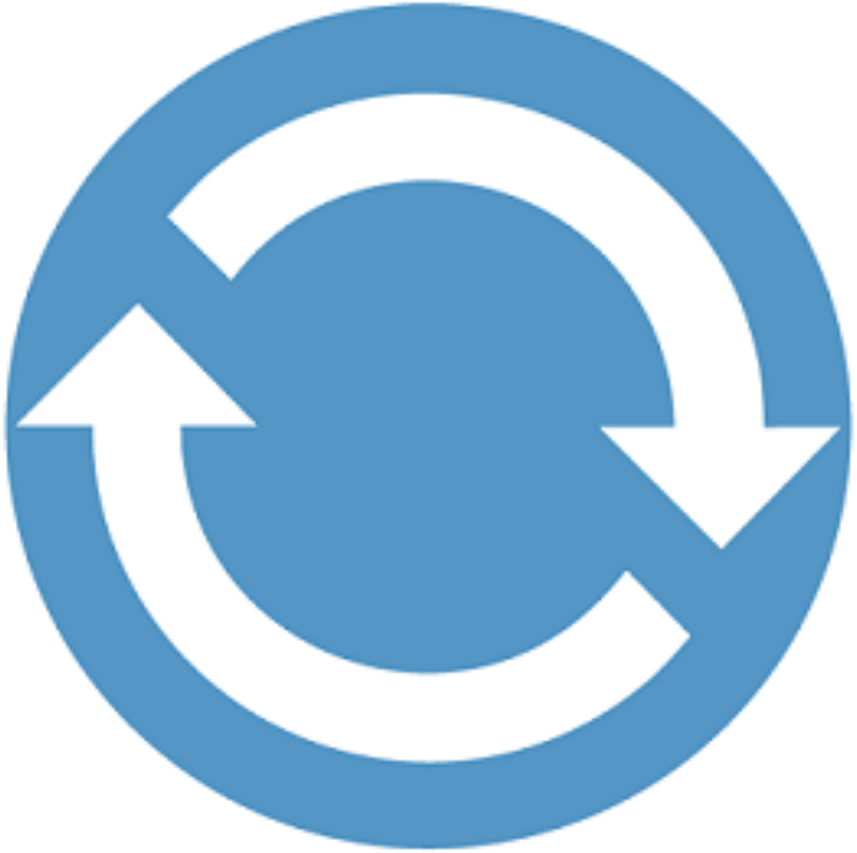
```
for fruit in ['apple', 'banana', 'kiwi']:
    print("I want a", fruit)
```

```
I want a apple
I want a banana
I want a kiwi
```

The above code worked because Python is sending the **next** item to our variable *fruit*.

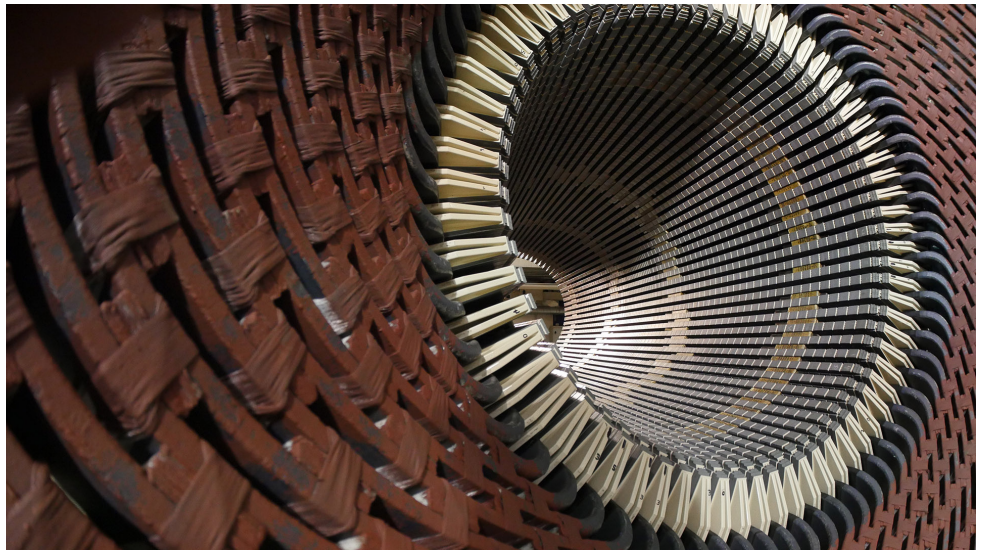
Create an iterator class

```
class MyNumbers:
  def __iter__(self):
    self.a = 1
    return self

  def __next__(self):
    if self.a <= 20:
      x = self.a
      self.a += 1
      return x
    else:
      raise StopIteration

myclass = MyNumbers()
myiter = iter(myclass)

for x in myiter:
  print(x)
```



generators

- A generator is an iterator, but not vice versa
- A generator is a special type of function that does not return a single value; instead it returns an iterator object with a sequence of values
- Instead of a return statement, a yield statement returns the next value
- NLTK ngrams() returns a generator object



```
for ngram in ngrams('apple banana cucumber dill'.split(), 3):
    print(ngram)
```

```
('apple', 'banana', 'cucumber')
('banana', 'cucumber', 'dill')
```

Writing a generator

Fibonacci – iterative version

```
def fib(max):
    numbers = []
    a, b = 0, 1
    while a < max:
        numbers.append(a)
        a, b = b, a + b
    return numbers

fib(5)
```

```
[0, 1, 1, 2, 3]
```

Fibonacci generator

```
def fib_gen(max):
    a, b = 0, 1
    while a < max:
        yield a
        a, b = b, a + b

fib_gen(5)
```

```
<generator object fib_gen at 0x7f8fb43df200>
```

```
for item in fib_gen(5):
    print(item)
```

```
0
1
1
2
3
```

Why use generators?

- If you have a huge amount of data, generators will take up less memory by streaming the data.

N-gram Code Examples

- GitHub:
  - Notebook 8-1
  - Notebook 8-2

Essential points to note

- An n-gram language model is a probabilistic model of a corpus
- N-gram models used:
  - spelling correction
  - machine translation
  - speech recognition
  - auto suggestion
  - features in machine learning applications

To Do

- No quiz
- Portfolio N-grams: Language identification using n-grams

Next topic

Corpora

CFG Context-Free Grammar