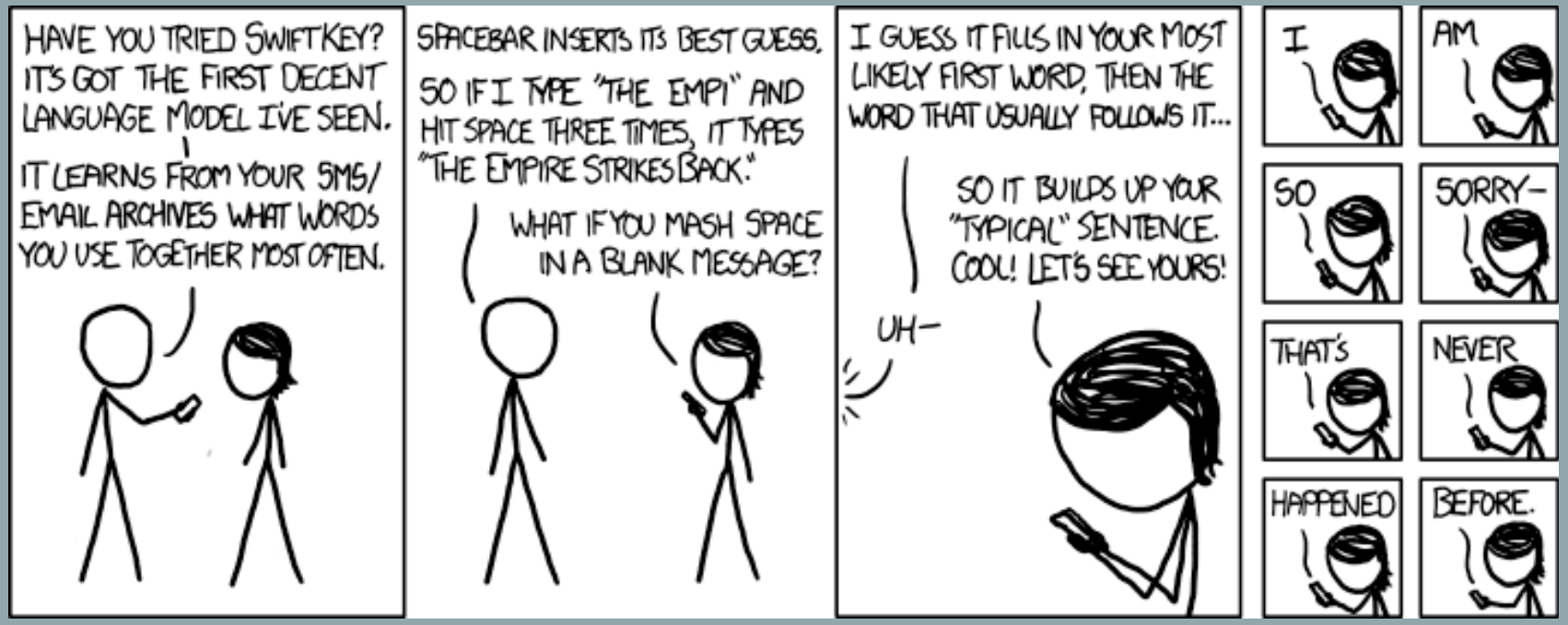
LING-362

Introduction to Natural Language Processing

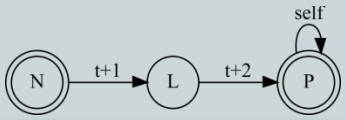
N-grams and language Models II



Homework assignment - structure

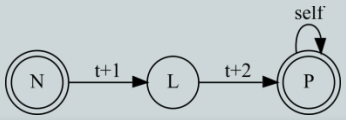
- For the morphology homework:
  - We used a machine-readable dictionary to keep code clean
  - Python script reads in lexical resource (use open and .read(), filename from argparse)
  - Creates an object to handle task: transducer
  - Applies method of object to each input, returns outputs

- Typical workflow for many tools!



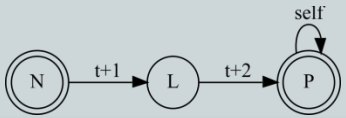
Quick reminder: probabilities

- If two events A, B, are **independent**, then:
  - p(A|B) = p(A)
- For independent events, the **chain rule** applies:
  - p(A&B) = p(A)*p(B)
- For example:
  - P(⚀ & ⚃) = 1/6 * 1/6 = 1/36



Quick reminder: N-gram models

- A very simple (and efficient) way of modeling context is using **n-grams**
  - Decide on a useful context size, often 3 words
  - Save analyses not of individual words, but of words given the previous 2 words – **trigram model**



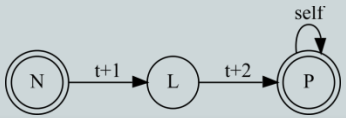
Building our own!

[Wikimedia]

- Consider the following texts, by Charles Dickens*:

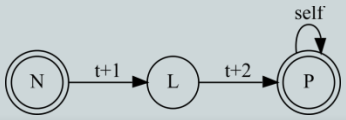
bigram model	trigram model	4-gram model
depose to linger yet, pointing upward ! ' are melting from me, pointing upward	signature under such circumstances, Mr. Mell, formerly poor pinched usher to my Middlesex magistrate	that I stole into the next street, and open a chemist's shop? Whether he could

*sort of



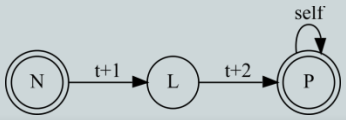
DIY natural language generation!

- Let's create Dickensian stories
- We need:
  - Dickensian training data and a way to read it
  - Random selection
  - A mechanism to choose best output for a story of a certain length
- Download:
  - *ngrams.py*
  - *dickens.txt*



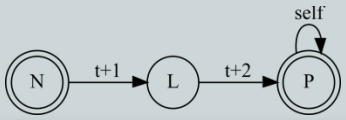
Reminder – reading files

```
parser = argparse.ArgumentParser()
parser.add_argument("file")

options = parser.parse_args()
training_file = options.file

a = open(training_file).read()

with open(training_file,'r') as f:

    training_data = f.read()

...

counts = get_counts(context_length, training_data)
```



Learning the frequencies

- To track frequencies of n-grams, we'll need a dictionary tracking counts like this:
  - Key: "In", "the"
    - Value: another dictionary:
      - Key: "beginning", value: 2
      - Key: "end", value: 5
      - …
  - The function get_counts() should build this dictionary



Learning the frequencies

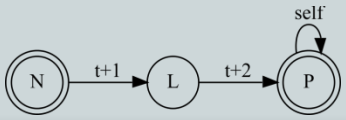
- It looks like we could use a **list** for the key:

```
freqs[["in","the"]] = {"end":5, "beginning": 2}
```

- Actually, this is **forbidden**:
  - Dictionary keys must be **immutable**
  - List values could be changed:
    - We could change the second list member of [“in”, “the”]

```
my_key = ["in","the"]
freqs[my_key] = 5
my_key[0] = "by"
```

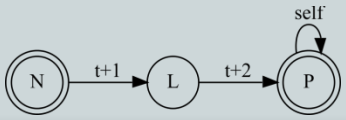
    - Dictionary would be broken



Tuples: not quite Lists

- Python has a cousin data-type to lists: **tuples**
  - Tuples are like lists, but they are **immutable**
  - Once defined they can't be changed
  - Look a lot like lists in **round** brackets:

```
# This is a list:
a = ["in", "the"]

# This is a tuple:
b = ("in", "the")
```

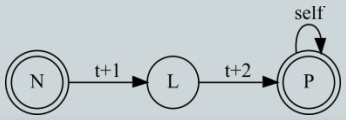


Tuples: not quite Lists

- In practice, tuples are used:
  - When an **immutable** data type is required
  - In contexts where something like a list:
    - Has a specified, invariable length
    - Each position has a predictable meaning

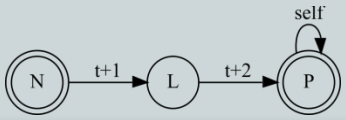
- Example: person's height, weight and age:

```
# Don't need to append or change these:
stats = (175, 72, 43)
```



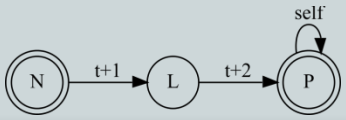
get_counts()

```
def get_counts(context_length, training_text):

    counts = {}

    tokens = word_tokenize(training_text)
    for i in range(len(tokens) - context_length):
        context = []
        next_token = tokens[i + context_length]
        for j in range(context_length):
            context.append(tokens[i + j])

        # Add 1 to frequency or create new dictionary item for this tuple
        if tuple(context) in counts:
            if next_token in counts[tuple(context)]:
                counts[tuple(context)][next_token] += 1
            else:
                counts[tuple(context)] = {next_token: 1}
        else:
            counts[tuple(context)] = {next_token: 1}

    return counts
```



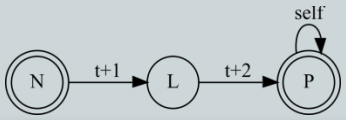
generate_from_file(context_length,training_file,output_length=10)

```
first_tokens = choice(counts.keys())  # Choose a random first context
output_list = list(first_tokens)
current_context = first_tokens

for i in range(output_length):
    next_context = max(counts[current_context], key=counts[current_context].get)
    temp = list(current_context)
    temp.pop(0)  # Remove first token in previous context
    temp.append(next_context)  # Add new token for the next context
    next_token = temp[-1]
    next_context = tuple(temp)

    current_context = next_context

    output_list.append(next_token)

print(" ".join(output_list))
```



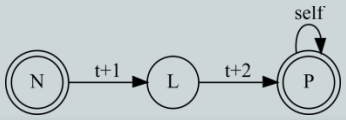
Spot the genre

these changes into our other midmarket power forms . Thanks again for your help on this . Carol St. Clair EB 3889 713-853-3989 (Phone) 713-646-3393 (Fax) carol.st.clair @ enron.com All , Please see the attached Interconnect Agreement with Questar . Transwestern will own and operate the interconnect . Questar may be able to purchase material , but some of



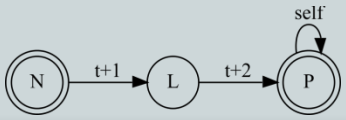
Spot the genre

Furies , and I 'll be as good as my word ; but speciously for Master Fenton . Well , on went he for a search , and away went I for foul clothes . But mark the sequel , Master Brook-I suffered the pangs of three several deaths : first , an intolerable fright to be detected with a jealous rotten bell-wether



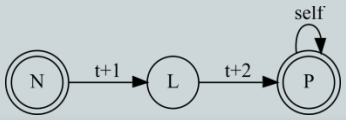
Spot the genre

ambush in her system , ready , at the corner of the street , with his great kite at his back , a very monument of human misery . My aunt went on with a quiet enjoyment , in which there was very little affectation , if any ; drinking the warm ale . 'She 's the most ridiculous of mortals . But



Homework

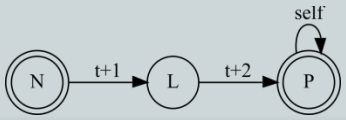
- Due by end of Friday, October 22
- Longer project - improve the n-gram generator
  1. Add an argparse argument to set context length (**default** : 2). Note that parameters default to string, so use **int()** [2 pts]
  2. Add an argument to set the starting ngram: [6 pts]
     - User can specify to start the story with "The woman"
     - Note parameter may contain spaces so surround it with double quotes in your configuration, for example:
       - *> python ngrams.py --starter "The woman"*



Homework

2. (ctd.)
   - Tokenize the starter n-gram via word_tokenize() and feed it to the generator as a tuple instead of: choice(counts.keys())
   - If the starter n-gram is not specified, use choice() as before
   - If it's too short or long (compared to context length), print a warning and quit:

```
import sys
...
if ... :
    print("some sensible warning")
    sys.exit(0)
```



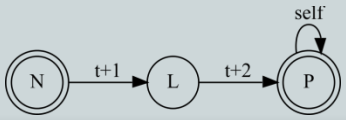
Homework

3. Handle n-grams not in the language model [4 pts]
   - It's possible the user wants to start with "Twas brillig", but that's not in the training data
   - Check to make sure the input as a tuple, e.g. (**"Twas"**, **"brillig"**), is a known key in the ***counts*** dictionary
   - If not, print the user's starter n-gram followed by a **period**; then continue generating from a random starter via choice()

> python ngrams.py -s "Twas brillig" dickens.txt

Output:

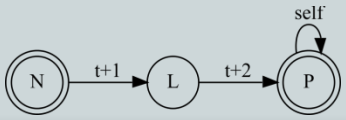
Twas brillig . It would be much more easy to be born a Jackson



Homework

Extra credit challenges: [2 pt each]

- If starter n-gram is longer than context length (e.g. the user specifies context = 2, but to start with “The young man”), print the first words until only the right amount of tokens remains (in this example print the initial ‘The’, then start generating with the bigram ‘young man’, if it’s in *counts*)
- Check whether a user-specified initial n-gram ends in a punctuation token; if so, proceed as before but **without** adding the period.

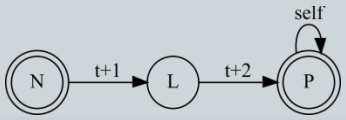


More about language models

- For predictive modeling of language, n-gram models can be very effective
- We can calculate the probability of any sequence of words, given training data:

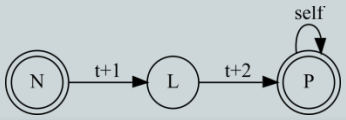
$$P(w1,w2 \ldots wk) = P(w1)*P(w2|w1)*P(w3|w1,w2)\ldots$$

$$= \prod_{k=1}^{n} P(w_k|w_1 \ldots w_{k-1})$$



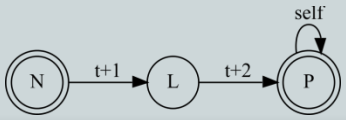
How good will our model be?

- To measure the quality of a model, we'd like to know how well it **fits** a certain data set
- What kind of language is easy to model?
  - If the language we are modeling has only one word, it's perfectly predictable:
    - *Spam spam spam spam spam ….*
  - If it has 1,000,000 different equiprobable words, it's very hard to model
  - If it has 1,000,000 different words but…
    - *Spam spam spam spam spam furry spam*



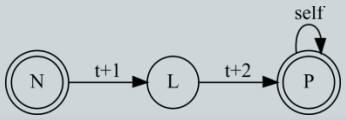
How good will our model be?

- How can we measure this predictability?
- We can use a given model to **assign** a probability to some data
  - We know how to get $P(w_1, w_2, \ldots w_n)$
  - As we use the chain rule, the probability will get very small (each multiplication creates a tiny fraction)
  - We take the -N^{th} root for N such multiplications – this is a measure called…



Perplexity

- For many purposes, we will want to know what the likelihood of a sequence of tokens is:
  - Evaluate likelihood of suggested translations / NLG
  - Recognize text type (does this look like a newspaper article?)
  - Recognize dialect/variety/non-native language
  - Predict performance in domain adaptation
  - … and much more!
- We need to measure how ‘surprising’ a text is given some training data for comparison



Perplexity (PP)

- Measured in general:

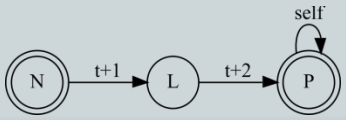
$$PP(text) = \sqrt[N]{\frac{1}{P(w_1 \ldots w_N)}}$$

- For a bigram model:

$$PP(text) = \sqrt[N]{\prod_{i=1}^{N} \frac{1}{P(w_i|w_{i-1})}}$$

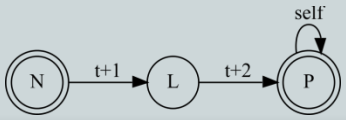
- For a trigram model:

$$PP(text) = \sqrt[N]{\prod_{i=1}^{N} \frac{1}{P(w_i|w_{i-1})P(w_{i-1}|w_{i-2})}}$$



Perplexity (PP)

- Example: *spam language*
  - PP(spam*10) = $\sqrt[10]{\frac{1}{(1*1*..*1)}} = 1$
- Example: *million word language*
  - PP(a,b,c..j)= $\sqrt[10]{\frac{1}{(0.0000001*..0.0000001)}} = 1000000$
- Example: *furry spam language*
  - PP(spam..furry,spam)= $\sqrt[10]{\frac{1}{(0.999*..0.0000001*0.999)}} = 3.98$



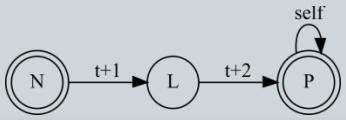
Exercise – make up a text

- Can you make up a sentence that is:
  - Bad English but rates low on perplexity
  - Good English but rates high on perplexity

- For this text, using A. **unigrams** B. **bigrams**:

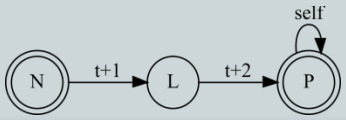
If you go out in the woods today
You're sure of a big surprise.
If you go out in the woods today
You'd better go in disguise.

For every bear that ever there was
Will gather there for certain, because
Today's the day the teddy bears have their picnic.



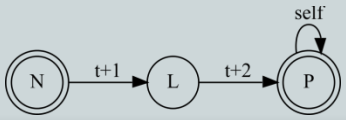
Unigram example

- Good English, high model perplexity:
  - When I venture into the forest tonight
    I'll be very surprised…
- Bad English, low perplexity:
  - If bears woods picnic you you you today woods…



Bigram example

- Good English, high model perplexity:
  - Had I gone strolling through the forest I should have worn a disguise
- Bad English, low perplexity:
  - You'd better a big surprise if you sure of a disguise. the woods today's the day the woods today



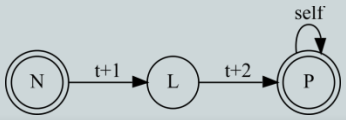
Can we always know $P(w_i)$?

- A problem with perplexity (and language models in general) is knowing $P(w_i)$
- Words can be formed productively:
  - ***dancerliness***
  - ***slacktivism***
  - …

$$PP(text) = \sqrt[N]{\prod_{i=1}^{N} \frac{1}{P(w_i|w_{i-1})}}$$

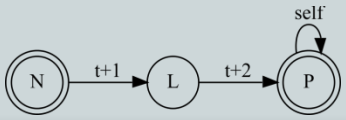
- These words are **out-of-data** or **out-of-vocabulary** ("OOV" words)

  ?? How can we assign them a probability **??**



Smoothing

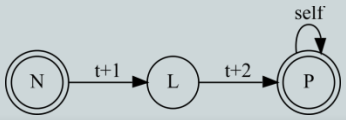
- We can assign some small frequency to each word in the text we're evaluating
- Simplest option: add 1
- This is called: **Laplace Smoothing**
  - **Pro:** very simple to do
  - **Con:** overestimates likelihood of OOV items –
    - in reality the likelihood of any particular OOV item is **not** half that of an item that occurs once (1+1 = 2*1)
    - error is compounded in n-gram models (each OOV item has likelihood of combining with other items…)
- we can take another small number (**δ smoothing**), but which?



Smoothing

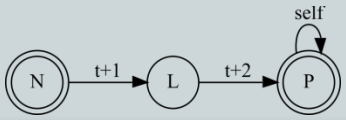
- A better entry-level smoothing algorithm is to estimate the likelihood of rare items
  - How often does a new item occur?

  - Every time a new item comes along, it's unique – a ***hapax legomenon*** (Greek: said once)
  - To estimate the likelihood of a 'surprise' we can check how often we were surprised in the past



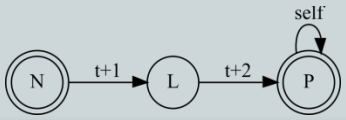
Smoothing

- **Good-Turing Discounting**:
  - Likelihood of each OOV item = hapax / N
  - Similar insights in productivity studies (Baayen 2009) – likelihood of novel word formation
- To smooth, we first check how many unique items are in the **training data** / data size
  - Assign this small fraction to each OOV item we meet in the **test data**
  - **Discount** the probability of other data so that sum is **1**



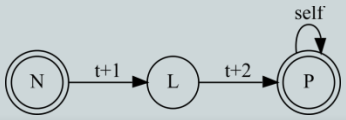
How to get perplexity in practice?

- We will need:
  - Some **training data** again, to base our model on
  - Some **test data** to calculate perplexity for
  - Calculate probabilities for all possible training sequences
  - Assign probabilities for product of text occurrences
  - Add **smoothing** in unknown cases



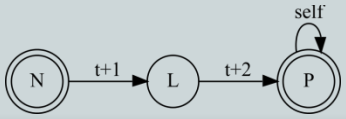
Perplexity – training data

```
import nltk

# Some data to make a model out of
# We can read this from a file too!
text  = """Mary had a little lamb,
His fleece was white as snow,
And everywhere that Mary went,
The lamb was sure to go.
… ."""

tokens = nltk.word_tokenize(text)
```



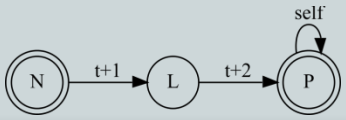
Perplexity – a unigram model

```
def unigram(tokens):
    # Token list in, model out…
    model = {}
    for tok in tokens:
        # Check if we've seen this token before
        if tok in  model:  # If we have, increase it's frequency by 1
            model[tok] += 1
        else:  # If we haven't, assign a frequency of 1
            model[tok] = 1

    for word in model:
        # Normalize probabilities so they sum up to 1
        model[word] = model[word]/float(len(model))  # Python 2 compat.
    return model
```



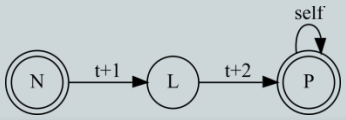
Perplexity – computing

```
def compute_perplexity(data, model):

    data = nltk.word_tokenize(data)
    perplexity = 1  # Initialize value: starting is P = 1
    N = 0
    for word in data:
        N += 1
        if word not in model:
            model[word] = 0.00001  # Rudimentary delta smoothing
        perplexity = perplexity * (1/model[word])
    perplexity = perplexity ** (1/float(N))  # ** means power
    return perplexity
```



Perplexity – real examples

```
text0 = "a little lamb had Mary"
text1 = "This is a story about a lamb with white fleece who went to school."
text2 = "On the other hand if we just talk about cheeseburgers etc. the model will be more perplexed!"
text3 = "Mais si on n'a pas des mots anglais c'est plus mauvais"

model = unigram(tokens)
print("All vocab in: " + str(compute_perplexity(text0, model)))
print("Some vocab overlap: " + str(compute_perplexity(text1, model)))
print("Same language: " + str(compute_perplexity(text2, model)))
print("French: " + str(compute_perplexity(text3, model)))
```



Perplexity – real examples

Output:

All vocab in: 30.6394384791

Some vocab overlap: 507.341574812

Same language: 16556.6079309

French: 100000.0