

N-grams in Python

Linguistics 445/515
Autumn 2008

Calculating n-grams

We want to take a practical task, i.e., using n-grams for natural language processing, and see how we can start implementing it in Python.

Some of today will move rather fast

- You may not have been able to come up with the final program on your own ...
- But I want you to be able to understand how it works

2

Defining the problem

Here's what I want us to do today:

- Input a text file → we already know how to do this.
- Output a file containing each bigram from the text, together with its frequency.

For today, we'll assume that a "word" is anything between white spaces.

- Although, we'll also take a quick look at regular expression processing to change that

3

Skeleton of the Inputting

```
# Declare file to be worked with
textfile = "furniture.txt"
```

```
# Open file & read first line
file = open(textfile, 'r')
line = file.readline()
```

```
while line:
    line = line.rstrip()

    # do stuff here

    line = file.readline()
file.close()
```

4

What Next?

Okay, we're able to read in a file line-by-line. Now, we need to do something ...

- Tokenize each line into words
- Create bigrams from the individual words
- Store the bigrams

For the moment, let's pretend we're working with unigrams, and so we'll do the following:

- Tokenize each line into words
- Store the unigrams [we'll use dictionaries for this]

5

Tokenization

A simple way to tokenize everything is to use `split`, which takes a string, splits it on whitespaces, and returns every non-whitespace item in a list.

```
while line:
    line = line.rstrip()
    # tokenize the text:
    tokens = line.split()
    ...
    # do stuff here
    ...
    line = file.readline()
```

6

Side topic: modules

That's a pretty simplistic tokenizer. Maybe someone's written a better one at some point ...

In fact, I wrote a slightly better tokenizer a couple years ago, and I've put it in a file called `useful.py`

- Q: How can you use my better tokenizer?
- A: With the python `import` statement, which allows you to import *modules*

7

import

At the top of your file, include the line:

- `from useful import tokenize`
- This says: from the module `useful` import the function `tokenize`

And then when we need the tokenizer, we can call it:

```
while line:
    line = line.rstrip()
    # tokenize the text:
    tokens = tokenize(line)
    # do stuff here
    line = file.readline()
```

8

Unigrams

Thus far, we've: read in a file and tokenized each line. Thus, we have access to unigrams.

```
while line:
    line = line.rstrip()
    # tokenize the text:
    tokens = tokenize(line)

    # loop over unigrams:
    for word in tokens:
        print word

    line = file.readline()
```

9

Data storage

Great! We have unigrams. ... And now we want to store them somewhere, along with their associated frequencies

- A list isn't quite right: once we store a word, how do we access it in the list when we encounter it again in a text?
- Plus, the data structure we want should store each word (type) and have the word "point to" its associated count

Python dictionaries are what we want ...

10

Dictionaries

Python has a **dictionary** data type, which is kind of like a set (unordered, no repeat keys), but is technically a hash which maps from one item (key) to another (value)

- It is an unordered set of *key:value* pairs

```
>>> d1 = {} # empty dictionary
>>> d1 = {'a':1, 'b':2, 'c':3} # dictionary with 3 keys
>>> x = d1['a'] # x = 1
>>> del d1['c'] # d1 = {'a':1, 'b':2}
>>> d1['d'] = 4 # d1 = {'a':1, 'd':4, 'b':2}
>>> k = d.keys() # k = ['a','d','b']
```

11

More on dictionaries

- To test whether an item is in a dictionary, use one of the following:

```
d1.has_key('a')
'a' in d1
```

- To iterate over the items in a dictionary, you have lots of options:

```
for alpha, num in d1.iteritems():
    print alpha + '\t' + str(num)
for alpha in d1.keys():
    print alpha + '\t' + str(d1[alpha])
for alpha in d1:
    print alpha + '\t' + str(d1[alpha])
```

12

Using a dictionary to store unigrams

So, let's create a dictionary `Unigrams`, which we'll use to store each unigram

- Creation of dictionary: `Unigrams = {}` (think about where to put this in the program)
- Adding each word to the dictionary: `Unigrams[word] = 1` (not quite right)

```
Unigrams = {}
while line:
    line = line.rstrip()
    # tokenize the text:
    tokens = tokenize(line)

    # loop over unigrams:
    for word in tokens:
        Unigrams[word] = 1
```

13

Adding each unigram

We really want to do the following:

- Make `Unigrams[word] = 1` if we've never seen this word before
- Make `Unigrams[word] = Unigrams[word] + 1` if we have seen this before
 - Use an if statement!

```
# loop over unigrams:
for word in tokens:
    if word in Unigrams:
        Unigrams[word] += 1
    else:
        Unigrams[word] = 1
```

14

Outputting unigrams to a file

- Iterate over the unigrams in the dictionary
- Write out each unigram, along with its count

```
# Write unigrams to output file:
output_file = open('unigrams.txt', 'w')
for unigram in Unigrams:
    count = Unigrams[unigram]
    output_file.write(str(count)+'\t'+unigram+'\n')
output_file.close()
```

15

From unigrams to bigrams

Now, let's extend the analysis from unigrams to bigrams.

- What do we need to do to make this happen? (Aside from nicely renaming our dictionary to `Bigrams` and so forth).

```
# loop over words in input:
for word in tokens:
    # something more needs to happen here:
    bigram = word
    if bigram in Bigrams:
        Bigrams[bigram] += 1
    else:
        Bigrams[bigram] = 1
```

16

Calculating bigrams

One thing novices sometimes do is to tokenize each line into tokens and then loop over tokens from 1 to $i-1$, taking pairs of words from the list

- Problem: doesn't account for bigrams between lines

Better solution:

- Keep track of the previous word
- Before we start looping over the input, include these lines:

```
# initialize variables:
Bigrams = {}
prev_word = "START"
```

17

Calculating bigrams (cont.)

```
# loop over words in input:
for word in tokens:
    # concatenate words to get bigram:
    bigram = prev_word + ' ' + word
    if bigram in Bigrams:
        Bigrams[bigram] += 1
    else:
        Bigrams[bigram] = 1
    # change value of prev_word
    prev_word = word
```

18

Regular expressions (for better tokenizing)

[If we have time, we'll cover these slides ... but we probably won't]

We mentioned that we can do better tokenization if we have regular expressions

- The module `re` provides regular expression functionality

The first thing to note is that python provides a raw string notation which is easier to use if you need to match a backslash

- `"\\section"` matches `\section`
- `r"\\section"` also matches `\section`

19

re.compile

The first step is to let python compile the regular expression into an internal format that it can use

```
import re
p = re.compile('ab*')
```

```
# can also pass flags to compile:
p = re.compile('ab*', re.IGNORECASE)
```

`p` is now a regular expression object, but we still have to know how to use it

20

Performing matches

- `match` — determines if the RE matches at the *beginning* of the string
- `search` — determines if the RE matches anywhere in the string
 - `group` — find the part of the string which matched
 - `start`, `end`, `span` — find the coverage of the match within the string

```
s = "this is my string #123"
p = re.compile('[0-9]+') # or '\d+'

print p.match(s) # None
m = p.search(s)
print m.group() # '123'
print m.span() # (19, 22)
```

21

Finding all matches

- `findall` — returns a list of matches
- `finditer` — allows you to iterate over all matches

```
s = "45 is a number, not number 2, but this is my string #123"
p = re.compile('\d+')

p.findall(s) # ['45', '2', '123']
iterator = p.finditer(s)
for match in iterator:
    print match.span() # (0, 2) \n (27, 28) \n (53, 56)
```

22

Putting it into one command (i.e., no compile)

If you are only going to use the RE once, it might be more readable to put it into one line

- It's kind of annoying to use `compile` and then `search`

```
s = "this is my string #123"
m = re.search('[0-9]+', s)
print m.group()
```

23

Splitting strings

We can use REs to tell us how to split up text

- Can specify how many splits to make, too
- Here, `\W` refers to non-word characters

```
>>> p = re.compile(r'\W+')
>>> p.split('This is a test, short and sweet, of split().')
['This', 'is', 'a', 'test', 'short', 'and', 'sweet', 'of', 'split', '']
>>> p.split('This is a test, short and sweet, of split().', 3)
['This', 'is', 'a', 'test, short and sweet, of split().']
```

24

Capturing parentheses

We can use capturing parentheses to tell us exactly what the items were that we split on

```
>>> p = re.compile(r'\W+')
>>> p2 = re.compile(r'(\W+)')
>>> p.split('This... is a test.')
['This', 'is', 'a', 'test', '']
>>> p2.split('This... is a test.')
['This', '...', 'is', ' ', 'a', ' ', 'test', '.', '']
```

25

Search and replace

Often we want to not only find a regular expression, but replace it with something else—we use `sub` for that

```
>>> p = re.compile('blue|white|red')
>>> p.sub('colour', 'blue socks and red shoes')
'colour socks and colour shoes'
>>> p.sub('colour', 'blue socks and red shoes', count=1)
'colour socks and red shoes'
```

26

Greediness

One very important thing to know about this RE matching is that it is *greedy*: it'll match the longest possible thing that it can

```
>>> s = '<html><head><title>Title</title>'
>>> len(s)
32
>>> print re.match('<.*>', s).span()
(0, 32)
>>> print re.match('<.*>', s).group()
<html><head><title>Title</title>
```

There is a way to do non-greedy matches, but I'll let you read the documentation for that.

27

REs for tokenization

Remember that program `useful.py` which tokenized text?

```
def tokenize(line):
    list = line.split()
    tokens = []
    for item in list:
        while re.match('\W', item):
            # non-alphanumeric item at beginning of item
            tokens.append(item[0])
            item = item[1:]
        # to maintain order, we use temp
        temp = []
        while re.search('\W$', item):
            # non-alphanumeric item at end of item
            temp.append(item[-1])
            item = item[:-1]
```

28

```
# Contraction handling
if item == "can't":
    tokens.append("can")
    tokens.append("n't")
# other n't words:
elif re.search("n't", item):
    tokens.append(item[:-3])
    tokens.append(item[-3:])
# other words with apostrophes ('s, 'll, etc.)
elif re.search("'", item):
    wordlist = item.split("'")
    tokens.append(wordlist[0])
    tokens.append("'" + wordlist[1])
# no apostrophe, i.e., normal word:
else:
    tokens.append(item)
tokens.extend(temp)
return tokens
```

29