Computer Science Principles

CHAPTER 6 – MACHINE LEARNING AND STRING MANIPULATION

Announcements

Read Chapter 6 in the Conery textbook (Explorations in Computing)

Acknowledgement: These slides are revised versions of slides prepared by Prof. Arthur Lee, Tony Mione, and Pravin Pawar for earlier CSE 101 classes. Some slides are based on Prof. Kevin McDonald at SBU CSE 101 lecture notes and the textbook by John Conery.

Machine Learning

Machine learning is a branch of computer science consisting of algorithms and techniques for "teaching" computers to recognize patterns, make predictions, detect trends, and the like.

- You can do lots of interesting things with machine learning and artificial intelligence
- See https://experiments.withgoogle.com/collection/ai for some examples you can play with online

One well-known application of machine learning is *spam filtering*

- As a user flags emails as spam, over time the software learns how to identify spam itself, flagging spam emails automatically

We will develop a spam filter that uses *word frequencies*

Machine Learning

For example, if the word “diet” appears in 63 of 500 emails flagged by the user as spam, the probability of a spam email containing “diet” is 63/500 = 0.126 or 12.6%

Such frequencies will help the software learn how to detect spam and calculate a probability that a particular email is spam

First we will cover some concepts that we will need to build the spam filter

Strings Revisited

Unlike lists, strings are **immutable** objects, which means you cannot change them

For example, if you try to execute the code given below:

```
fruit = 'apple'
fruit[0] = 'A'
```

Then you get the following error:

```
TypeError: 'str' object does not support item assignment
```

In other words, you can't assign a new value to an existing string by changing the contents of the string

- We can only replace an entire string

Strings Revisited

We need to use methods to create a new string based on an existing string

Some handy string methods include:

- **upper**: changes all the characters to uppercase
- **lower**: changes all the characters to lowercase
- **capitalize**: capitalizes the word

For each of these methods, the method makes a copy of the string, leaving the original unchanged

```
name = 'suny korea'
new_name = name.upper()
```

name will still be '**suny korea**', but **new_name** will be '**SUNY KOREA**'

Strings Revisited

Another example:

```
name = 'SUNY Korea'
new_name = name.lower()
```

new_name will be **'suny korea'**. The **name** variable remains unchanged.

One last example:

```
name = 'suny korea'
new_name = name.capitalize()
```

new_name will be **'Suny korea'**. The **name** variable remains unchanged.

Splitting Strings

- A very useful string method in Python is **split()**
- The method splits a string into smaller substrings, using the space character to separate “words” (but the words could actually have any characters in them)
- The substrings are placed inside of a list, which the **split** method returns

Example:

```
school = 'Stony Brook Univ'
parts = school.split()
```

parts will be the list **['Stony', 'Brook', 'Univ']**

Splitting Strings

- In fact, the **split** method will use any **whitespace** characters to split a string into parts
- Whitespace characters include spaces, tabs and newlines
- In Python, a newline is denoted **\n** and a tab is **\t**
  - These are examples of **escape sequences**, which use a **backslash** to denote special characters

Example:

```
line = 'To be or not to be,\nthat is\tthe question.'
words = line.split()
```

words will be **['To', 'be', 'or', 'not', 'to', 'be,', 'that', 'is', 'the', 'question.']**

Text Files

Files come in two general formats: **plain text** files and **binary** files

- A (plain) text file is a simple file whose contents can be read by a basic text editor
- .py and .txt files are examples of text files
- Everything not a text file (images, videos, MP3s, compiled programs, etc.) is called a **binary** file because the file has a specific structure

In this course we will only look at how to work with text files

Text Files

Files give us a convenient way to provide input to a program so that we don't have to type the input over and over

Programs that work with files need to perform three basic tasks:

1. **Open** the file
2. **Read** data from and/or **write** data to the file
3. **Close** the file so that other programs can access it

Let's see how these tasks are handled in Python

Reading Files in Python

To open a file in Python we need to give its location on the disk

- For example, suppose we have a file named "words.txt" in a folder named **CSE101** and that **CSE101** is in a folder named **Classes**
- Let's further assume that our program (the .py file) is saved in the folder named **Classes**
- Our program would refer to the file's name as **filename = "CSE101/words.txt"**

The slash is called a **separator** and forms part of the **path** to the file on the disk

- For example, assuming **Classes** is a folder within another folder (**…**), the following would be part of the path:
- **"…/Classes/CSE101/words.txt"**

Reading Files in Python

Once we have a file's path, we can open the file for reading using:

```
f = open(filename)
```

The function returns a file object and you can set this to whatever variable you like instead of **f**

To read a single line of text at a time, we can repeatedly call the **readline()** function:

```
line = f.readline()     # reads first line
line = f.readline()     # reads second line
```

and so on...

When we are done with the file, we type **f.close()** to close it

Reading Files in Python

Usually in programming we need to process an entire file, not just part of it

For this reason Python has a simpler syntax we can use when we need to process an entire file

To read a file's entire contents line-by-line, we can write this for loop:

```
for line in open(filename):
. . .
```

The advantage of this syntax is that we don't even need to make a separate variable (like f, from an earlier example) to point to the file and we don't need to manually close it

Example: Getting File Size

The function below takes the name of a file as an argument and returns the number of characters in the file

```
def filesize(filename):
    nchars = 0
    for line in open(filename):
        nchars += len(line)
    return nchars
```

Example usage:

```
size = filesize('../PythonLabs/data/email/good.txt')
```

See filesize.py

Counting Words in a File

The Unix/Linux family of operating systems has a command called **wc**, which gives a count of how many words are in a file

Consider a **wc** function in Python that performs the same task

Our **wc** function will return three values (in this order):

- The number of lines in the file
- The number of words in the file
- The number of characters in the file

This means we need to count three quantities

- Therefore, three counters (variables) will be needed, each initialized to zero

Counting Words in a File

Python provides a convenient means for initializing multiple variables to the same value via a **multiple target** assignment.

- Instead of writing three separate assignment statements, we can collapse them into one
- Example: **nlines = nwords = nchars = 0**

To return three values from our function we will return a **tuple**

A tuple is like a list in that it contains several values

- Unlike a list, a tuple is immutable (i.e., its contents cannot be changed, just as a string is immutable)

Example: wc() Function

```
def wc(filename):
    nlines = nwords = nchars = 0
    for line in open(filename):
        nlines += 1
        nwords += len(line.split())
        nchars += len(line)
    return nlines, nwords, nchars
```

We can perform a tuple assignment to save the multiple values returned by the **wc** function:

```
lines, words, chars = wc('../PythonLabs/data/email/good.txt')
```

See wc.py

Dictionaries (next)

In Python, a **dictionary** is a type of collection where we can index (access) an element in the collection using a name instead of an integer index (as in a list)

We create a dictionary using curly braces, { }, but we access the values using square brackets [ ]

To create an empty dictionary, we type this:

```
dictionary_name = {}
```

To insert or update a value stored in a dictionary, we give the **key** for the value and the **value** itself

Dictionaries

Suppose we want a **distances** dictionary to represent the number of feet in a single yard, fathom, furlong, or mile

We might *initialize* these values as follows:

```
distances['yard'] = 3
distances['fathom'] = 6
distances['furlong'] = 660
distances['mile'] = 5280
```

distances is now: **{'fathom': 6, 'furlong': 660, 'mile': 5280, 'yard': 3}**

- The strings, **'fathom'**, **'furlong'**, **'mile'** and **'yard'** are the *keys* of the dictionary.
- **6**, **660**, **5280** and **3** are the *values* of the dictionary.

Dictionaries

distances is: **{'fathom': 6, 'furlong': 660, 'mile': 5280, 'yard': 3}**

We look up a value in the dictionary by giving its key:

- **distances['fathom']** has the value 6

Suppose we wanted to know how many feet are in 10 furlongs

- The code **10 * distances['furlong']** would give us the answer (6,600)

Now how many miles is 10 furlongs?

- **(10 * distances['furlong']) / distances['mile']**

Dictionaries

In programming, a dictionary is considered an *unordered* collection

- This means that the concept of sorting really doesn't apply naturally to dictionaries (unlike a real dictionary, which is definitely sorted!)

Another important fact is that only immutable types can be used as the keys

- So you can use **integers**, **strings** and **floating-point numbers** for keys
- This makes dictionaries a little more flexible than lists in that regard
  - Note that lists can only use integers as an index (key)

Dictionaries

Here is another example of a dictionary where we map Arabic numerals to Roman numerals:

```
roman = { 1: 'I', 5: 'V', 10: 'X', 50: 'L', 100: 'C'}
```

- Note that 1, 5, 10, etc. are not indexes as with a list, but are rather keys

If we try to use a key that is not in the dictionary, the program will crash

So, first we should use the **in** operator to check if the value is in the dictionary

- An example is given on the next slide

Dictionaries

```
number = 10
if number in roman:
   print(roman[number])
else:
   print('Numeral not recognized.')
```

Another option is to use the **get** method for dictionaries

- If we don't know if a key is in the dictionary, we can use the **get** method instead of **[]** to retrieve a value
- We must provide an argument that says what should be used as the value if the key is not found. An example:

```
res = roman.get(number, 'Numeral not recognized')
print(res)
```

Word Frequencies for Spam Filtering

Getting back to our original problem, we want to build a program that will do basic spam filtering

- Part of the solution will include counting how many times each word appears in the input email message

We can define a dictionary called **count** to serve this purpose:

```
count = {}
```

To increment the count for a word, we can use the **+= 1** notation

- Suppose the variable **word** has the string we want to increment the count of
- We can write this: `count[word] += 1`

Word Frequencies for Spam Filtering

But what if we aren't sure the string stored in **word** is already in the dictionary?

- Code like **`count[word] += 1`** will cause the program to crash if there is no value associated with **word** that we can add 1 to

We should first check using either the if-based approach we saw earlier, or use the **`setdefault`** method, which avoids the need to use an if statement

Both of these techniques are most easily understood by example

Word Frequencies for Spam Filtering

Option 1: Use an if statement:

```
if word not in count:
   count[word] = 1
else:
   count[word] += 1
```

Option 2: Use the **setdefault** method:

- The **setdefault** method can take the place of the if statement
- We will set **count[word]** to 0 **only if** the string inside **word** is not a key in the dictionary yet

```
count.setdefault(word, 0)
count[word] += 1
```

Word Frequencies for Spam Filtering

Now we need to get the individual words from the input file so that we can use our **count** dictionary to count how many times each word appears in the file

We can use **split** method:

```
for line in open('../PythonLabs/data/text/quote1.txt'):
    words = line.split()
```

quote1.txt contains this text:

```
If you have no confidence in self,
    you are twice defeated in the race of life.
With confidence, you have won even before you have
started.
-- Marcus Tullius Cicero (106 BC -- 43 BC)
```

Word Frequencies for Spam Filtering

quote1.txt contains this text:

```
If you have no confidence in self,
   you are twice defeated in the race of life.
With confidence, you have won even before you have started.
-- Marcus Tullius Cicero (106 BC -- 43 BC)
```

Calling **split** *on each line* of the text yields these lists:

```
['If', 'you', 'have', 'no', 'confidence', 'in', 'self,']
['you', 'are', 'twice', 'defeated', 'in', 'the', 'race', 'of', 'life.']
['With', 'confidence,', 'you', 'have', 'won', 'even', 'before', 'you', 'have', 'started.']
['--', 'Marcus', 'Tullius', 'Cicero', '(106',  'BC', '--', '43', 'BC)']
```

Word Frequencies for Spam Filtering

So far this is looking pretty good. Let's insert each word into a dictionary now and keep track of the counts:

```
count = {}
for line in open('../PythonLabs/data/text/quote1.txt'):
    words = line.split()
    for word in words:
        count.setdefault(word, 0)
        count[word] += 1
```

The **count** dictionary will now contain a count of every word in the file

- The full dictionary is shown on the following slide

Word Frequencies for Spam Filtering

```
{'If': 1, 'you': 4, 'have': 3, 'no': 1, 'confidence': 1,
'in': 2, 'self,': 1, 'are': 1, 'twice': 1, 'defeated': 1,
'the': 1, 'race': 1, 'of': 1, 'life.': 1, 'With': 1, 'confidence,': 1,
'won': 1, 'even': 1, 'before': 1, 'started.': 1, '--': 2,
'Marcus': 1, 'Tullius': 1, 'Cicero': 1,
'(106': 1, 'BC': 1, '43': 1, 'BC)': 1}
```

Do you see any problems with this?

Word Frequencies for Spam Filtering

```
{'If': 1, 'you': 4, 'have': 3, 'no': 1, 'confidence': 1,
'in': 2, 'self,': 1, 'are': 1, 'twice': 1, 'defeated': 1,
'the': 1, 'race': 1, 'of': 1, 'life.': 1, 'With': 1, 'confidence,': 1,
'won': 1, 'even': 1, 'before': 1, 'started.': 1, '--': 2,
'Marcus': 1, 'Tullius': 1, 'Cicero': 1,
'(106': 1, 'BC': 1, '43': 1, 'BC)': 1}
```

Do you see any problems with this?

- How can we improve how we count words?

Word Frequencies for Spam Filtering

We have a few problems:

- Punctuation is causing issues: "confidence" and "confidence," (with a comma) are seen as separate words
- This means they are treated as different keys in the dictionary
- We might have the same word appearing in the text with different capitalization (not shown in this example)

To solve these two problems we will convert all words to lowercase, which is easy, and we will *strip out* all punctuation marks

Stripping Strings

The **strip** method in Python will let us delete from the beginning and end of a string any characters from the string that we don't want

For example, **s1.strip('0123456789')** removes all numerals at the start or end of string **s1**

Here's a concrete example:

```
s1 = '2/13/2193. Astronauts living on Mars base: 4,920'
s2 = s1.strip('0123456789')
```

s2 will contain **'/13/2193. Astronauts living on Mars base: 4,'**

s1 will remain unchanged

Stripping Strings

Because stripping punctuation is a common operation in text processing, Python has it built-in through the **string** module:

```
import string

s1 = 'Good morning!'
s2 = s1.strip(string.punctuation)
```

s2 will contain **'Good morning'**

Word Frequencies for Spam Filtering

With these programming capabilities at hand, we can write a function **wf** that will create a dictionary of word frequencies for us

- It will rely on a helper function **tokenize** that will split a string into a list of lowercase words with punctuation marks stripped from each lowercase string

In programming, the word **tokenize** means to process an input string, splitting or dividing it into its constituent parts (or substrings)

- These substrings are the **tokens**

Let's take a look at the **tokenize** function

Word Frequencies for Spam Filtering

```
import string

def tokenize(s):
    tokens = []
    for x in s.split():
        tokens.append(x.strip(string.punctuation).lower())
    return tokens
```

Let's break down this code:

- **s** is a string – this could be a line from the file
- **x** is a word taken from the string (via **split**)
- **x** is stripped of its punctuation, converted to lowercase, and then appended to the **tokens** list

Word Frequencies for Spam Filtering

Let's see an example of **tokenize**:

```
res = tokenize('With confidence, you have won even before you have started.')
```

res will contain the list:

```
['with', 'confidence', 'you', 'have', 'won', 'even', 'before', 'you', 'have', 'started']
```

Now we can look at the completed **wf** function, on the next slide

This function will not be explicitly used in implementing our spam filter, but looking at it will give us a sense of how to work with dictionaries in an effective manner

Word Frequencies for Spam Filtering

```
def wf(filename):
    count = {}
    for line in open(filename):
        for word in tokenize(line):
            count.setdefault(word, 0)
            count[word] += 1
    return count
```

Let's break down this code on the next few slides

Word Frequencies for Spam Filtering

```
def wf(filename):
    count = {}
    for line in open(filename):
        for word in tokenize(line):
            count.setdefault(word, 0)
            count[word] += 1
    return count
```

Create an empty dictionary to hold the word frequencies

Word Frequencies for Spam Filtering

```
def wf(filename):
    count = {}
    for line in open(filename):
        for word in tokenize(line):
            count.setdefault(word, 0)
            count[word] += 1
    return count
```

Read every line from the file

Word Frequencies for Spam Filtering

```
def wf(filename):
    count = {}
    for line in open(filename):
        for word in tokenize(line):
            count.setdefault(word, 0)
            count[word] += 1
    return count
```

Tokenize the line and then process each token (a word) one at a time

Word Frequencies for Spam Filtering

```
def wf(filename):
    count = {}
    for line in open(filename):
        for word in tokenize(line):
            count.setdefault(word, 0)
            count[word] += 1
    return count
```

If `count[word]` is uninitialized, set it to 0

Word Frequencies for Spam Filtering

```
def wf(filename):
   count = {}
   for line in open(filename):
      for word in tokenize(line):
         count.setdefault(word, 0)
         count[word] += 1
   return count
```

Add 1 to the number of times this word is in the file

Word Frequencies for Spam Filtering

Let's see an example of **wf** for the quote1.txt file

```
res = wf('./quote1.txt')
```

res will contain:

```
{'if': 1, 'you': 4, 'have': 3, 'no': 1, 'confidence': 2,
'in': 2, 'self': 1, 'are': 1, 'twice': 1, 'defeated': 1,
'the': 1, 'race': 1, 'of': 1, 'life': 1, 'with': 1, 'won': 1,
'even': 1, 'before': 1, 'started': 1, '': 2, 'marcus': 1,
'tullius': 1, 'cicero': 1, '106': 1, 'bc': 2, '43': 1}
```

Note that the problems we encountered before have been fixed (e.g., “confidence”)

Spamicity (aka Spaminess)

Now that we have a way of counting the number of occurrences of each word in a file, we can use it to help us calculate the probability that an email is spam

Suppose we know that the word “secret” appears in 252 out of 1000 spam messages

- We might define the spam probability of “secret” as 252/1000 = 0.252
- In other words, the probability of seeing the word “secret” in a piece of spam is 0.252

This idea of a probability of some event being based on some known fact is called **conditional probability**

Spamicity

The probability of seeing a particular word **w** in an email we know is spam will be denoted:

P(w|spam)

Read this as “the probability of seeing word w, given a spam email”

From the Internet we can download training data that gives us these probabilities for a large number of words.

- This will allow us to "train" our algorithm to be able to detect spam
- The data is made available by people who design spam filtering algorithm.

Ultimately, we want to compute the “spamicity” of w, which is P(spam|w)

- This is the probability that an email is spam, given that w appears in the email

Spamicity

The textbook's SpamLab contains training data we can load using this code:

```
from PythonLabs.SpamLab import load_probabilities
pbad = load_probabilities('email/bad.txt')
pgood = load_probabilities('email/good.txt')
```

pbad is a dictionary that tells us the probability of a word appearing in a spam message

Likewise, **pgood** is a dictionary that tells us the probability of a word appearing in a non-spam message

Spamicity

For example, **pbad['money']** is 0.127 and **pgood['money']** is 0.0164

We see that the probability of “money” appearing in a spam message is 0.127, and its probability of appearing in a non-spam message is 0.0164

What if we encounter a word that is not in either dictionary?

- Then, we really don't know anything about the word and can't use it to help us identify spam messages

Spamicity

With **pbad** and **pgood** we can now define the “spamicity” of a word (between 0 and 1)

- The spamicity will be closer to 1 than to 0 when an word appears in more spam messages than good messages
- The spamicity is 0.5 if the word appears in as many spam messages as good messages.
- The spamicity will be closer to 0 when a word is found in more good messages than in spam messages

Define spamicity of a word w using this formula:

$$\begin{aligned}\textbf{spamicity(w)} &= \textbf{P(spam|w)} \\ &= \textbf{P(w|spam) / (P(w|spam) + P(w|good))}\end{aligned}$$

This formula is based on a concept called *Bayesian inference*, which is explained a little in the textbook if you want to check it out

The spamicity() Function

- The two conditional probabilities in the formula will come directly from the **pbad** and **pgood** dictionaries
- We can now write a function to compute spamicity:

```
def spamicity(w, pbad, pgood):
    if w in pbad and w in pgood:
        return pbad[w] / (pbad[w] + pgood[w])
    else:
        return None
```

- For example, spamicity("money") is 0.89, meaning that we predict 89% of incoming messages containing the word “money” are spam
- Note if the word **w** is not in both dictionaries, the value **None** is returned

Identifying Junk Mail

Now we will use our **spamicity** function to help us classify entire emails as good or spam

- Somehow we need to combine the spamicity values of the words in a message

The approach we will take is to consider “interesting” words – those words with high or low spamicity

Let's define the “interestingness quotient” (IQ) of a word w as:

$IQ(w)=|0.5-s|$, where s is the spamicity of word w

- The IQ of a word will range from 0.0 to 0.5, with 0.5 meaning a very interesting word
- So a word with a high spamicity will have an IQ near 0.5, but so will a low-spamicity word

Identifying Junk Mail

So the $IQ(w)=|0.5-s|$, where s is the spamicity of word w

Let's consider some examples:

- If $s = 0.90$, then $|0.5 - 0.90| = 0.4$
- If $s = 0.05$, then $|0.5 - 0.05| = 0.45$

A “boring” word would have s near 0.5.

- If $s = 0.47$, then it's IQ is $|0.5-0.47| = 0.03$, which is quite low

Identifying Junk Mail

We will use a data structure called a **priority queue**,

- A **priority queue** is a special type of list that is always sorted
- This will lets us add and remove items from a collection, always putting the highest priority item at the front

The SpamLab contains a version of a priority queue we can use to keep track of the most interesting words in an email

- It can take in a word along with it's spamicity value
- And will sort words by their "interestingness quotient" (IQ)

Identifying Junk Mail

As we add or remove words, the most interesting word will always be at the front

Here's a short example of how we might use the queue

```
from PythonLabs.SpamLab import WordQueue

queue = WordQueue(10)       # creates a queue to hold 10 words
s = spamicity('there', pbad, pgood)
queue.insert('there', s )      # the insert method translates the spamicity to IQ
s = spamicity('book', pbad, pgood)
queue.insert('book', s)
```

Identifying Junk Mail

Now we can define the top-level function **pspam**, which will give us a probability that a particular message is spam

The input will come from a file

The function will depend on another function called **combined_probability** that uses some formulas from probability theory to combine all the word spamicity values into a single number

See junk_mail.py

The pspam() Function

```
# import statements omitted to save space
def pspam(fn):
    queue = WordQueue(15)
    pbad = load_probabilities('email/bad.txt')
    pgood = load_probabilities('email/good.txt')
    with open(fn) as message:
        for line in message:
            for w in tokenize(line):
                p = spamicity(w, pbad, pgood)
                if p is not None:
                    queue.insert(w, p)
    return combined_probability(queue)
```

Set up the priority queue of interesting words

The pspam() Function

```
# import statements omitted to save space
def pspam(fn):
    queue = WordQueue(15)
    pbad = load_probabilities('email/bad.txt')
    pgood = load_probabilities('email/good.txt')
    with open(fn) as message:
        for line in message:
            for w in tokenize(line):
                p = spamicity(w, pbad, pgood)
                if p is not None:
                    queue.insert(w, p)
    return combined_probability(queue)
```

Initialize dictionaries

The pspam() Function

```
# import statements omitted to save space
def pspam(fn):
    queue = WordQueue(15)
    pbad = load_probabilities('email/bad.txt')
    pgood = load_probabilities('email/good.txt')
    with open(fn) as message:
        for line in message:
            for w in tokenize(line):
                p = spamicity(w, pbad, pgood)
                if p is not None:
                    queue.insert(w, p)
    return combined_probability(queue)
```

Open the file for reading

The pspam() Function

```
# import statements omitted to save space
def pspam(fn):
    queue = WordQueue(15)
    pbad = load_probabilities('email/bad.txt')
    pgood = load_probabilities('email/good.txt')
    with open(fn) as message:
        for line in message:
            for w in tokenize(line):
                p = spamicity(w, pbad, pgood)
                if p is not None:
                    queue.insert(w, p)
    return combined_probability(queue)
```

$\leftarrow$ Grab the next line of text from the file

The pspam() Function

```
# import statements omitted to save space
def pspam(fn):
    queue = WordQueue(15)
    pbad = load_probabilities('email/bad.txt')
    pgood = load_probabilities('email/good.txt')
    with open(fn) as message:
        for line in message:
            for w in tokenize(line):
                p = spamicity(w, pbad, pgood)
                if p is not None:
                    queue.insert(w, p)
    return combined_probability(queue)
```

Grab the next word from the line

The pspam() Function

```
# import statements omitted to save space
def pspam(fn):
    queue = WordQueue(15)
    pbad = load_probabilities('email/bad.txt')
    pgood = load_probabilities('email/good.txt')
    with open(fn) as message:
        for line in message:
            for w in tokenize(line):
                p = spamicity(w, pbad, pgood)
                if p is not None:
                    queue.insert(w, p)
    return combined_probability(queue)
```

Compute the word's spamicity

The pspam() Function

```
# import statements omitted to save space
def pspam(fn):
    queue = WordQueue(15)
    pbad = load_probabilities('email/bad.txt')
    pgood = load_probabilities('email/good.txt')
    with open(fn) as message:
        for line in message:
            for w in tokenize(line):
                p = spamicity(w, pbad, pgood)
                if p is not None:
                    queue.insert(w, p)
    return combined_probability(queue)
```

If the word has a spamicity value…

The pspam() Function

```
# import statements omitted to save space
def pspam(fn):
    queue = WordQueue(15)
    pbad = load_probabilities('email/bad.txt')
    pgood = load_probabilities('email/good.txt')
    with open(fn) as message:
        for line in message:
            for w in tokenize(line):
                p = spamicity(w, pbad, pgood)
                if p is not None:
                    queue.insert(w, p)
    return combined_probability(queue)
```

…then insert it into the priority queue, sorted by interestingness

The pspam() Function

```
# import statements omitted to save space
def pspam(fn):
    queue = WordQueue(15)
    pbad = load_probabilities('email/bad.txt')
    pgood = load_probabilities('email/good.txt')
    with open(fn) as message:
        for line in message:
            for w in tokenize(line):
                p = spamicity(w, pbad, pgood)
                if p is not None:
                    queue.insert(w, p)
    return combined_probability(queue)
```

Return the probability that this email is spam

pspam() Example #1

```
pspam('../PythonLabs/data/email/msg1.txt')
```

- Result: 0.929304 (high probability of spam)

File contents: (correctly identified as spam)

```
Hurting for funds right now?

It doesn't have to be that way. Here is 1,500 to ease your pain:
http://bulk.hideorganic.com/171026390236329103248372180

Transfer immediately to the account of your choice:
http://bulk.hideorganic.com/171026390236438808248372180

Take your time to pay off this amazing loan. Small payment due in late September
or early October (and not all in one payment!).
```

pspam() Example #2

pspam('../PythonLabs/data/email/msg2.txt')

- Result: 4.400695e-05 (practically a zero probability)

File contents: (correctly identified as non-spam)

Hi John:

Interesting that the key might be preventing ANY crystals from being able to nucleate - which kicks off a chain reaction and the whole thing goes to hell. Thus the very clean pot and not allowing anything to splash up onto the sides. Cooking really is chemistry!
Thanks for the links.

Susie

[... rest of message follows ...]

pspam() Example #3

pspam('../PythonLabs/data/email/msg3.txt')

- Result: 0.058101 (low probability of spam)

File contents: (incorrectly identified as non-spam)

> **Guess what conery@cs.uoregon.edu!**
> **AUTO CLEARANCE ENDS TONIGHT! : Price Drop On All Vehicles**
> **Want To Drive A Brand New Car Today For A Fraction Of What You Thought You Would Pay?**
> **Now You Can!**
> **Dealers Have Drastically Reduced MSRPs.**
> **AVAILABLE ONLY UNTIL 10:00 PM TONIGHT!**
> **http://server.beavercreekdistrict.com/3813010413df842258012632451675**
> **Click this link to unsubscribe: http://server.beavercreekdistrict.com/3813010413df52801063245167**

pspam() Example #4

pspam('../PythonLabs/data/email/msg4.txt')

- Result: 3.758445e-15 (practically a zero probability)

File contents: (correctly marked as non-spam)

Hi John,

I meant to ask you if you tried the revised cat command. Were you able to do what you needed?

Regarding your lab meetings... sure, I could come and give a brief description and answer any questions your group members might have. My assistant, Erik, has just put up more information from Chris' slides onto the wiki that might be helpful. It would be helpful to me if I knew in advance more specifically what kind of questions to address before coming - perhaps you can collect some at today's group meeting?

Cheers,
Rob

Example: Date Decoder

Consider the task of converting a date from one format to another

- A date of the form 8-MAR-85 includes the name of the month, which must be translated to a number
- We can use a dictionary to map month names to numbers

Let's consider a function **`date_decoder`**

- First it uses string operations to split the date into its three parts
- Then it translates the month to digits and corrects the year to include all four digits: 70-99 will be mapped to 1970-1999, and 00-69 will be mapped to 2000-2069
- Finally, the function returns the tuple **(year, month, day)**

See date_decoder.py

Example: date_decoder.py

```
def date_decoder(date):
    months = {'jan': 1, 'feb': 2, 'mar': 3, 'apr': 4,
              'may': 5, 'jun': 6, 'jul': 7, 'aug': 8,
              'sep': 9, 'oct': 10, 'nov': 11, 'dec': 12}
    parts = date.lower().split('-')
    day = int(parts[0])
    month = months[parts[1]]
    year = 1900 + int(parts[2])
    if int(parts[2]) <= 69:
        year += 100
    return year, month, day

    print(date_decoder('8-MAR-85'))
    print(date_decoder('17-Apr-25'))
```

Example: date_decoder.py

```
def date_decoder(date):
    months = {'jan': 1, 'feb': 2, 'mar': 3, 'apr': 4,
              'may': 5, 'jun': 6, 'jul': 7, 'aug': 8,
              'sep': 9, 'oct': 10, 'nov': 11, 'dec': 12}
    parts = date.lower().split('-')
    day = int(parts[0])
    month = months[parts[1]]
    year = 1900 + int(parts[2])
    if int(parts[2]) <= 69:
        year += 100
    return year, month, day

    print(date_decoder('8-MAR-85'))
    print(date_decoder('17-Apr-25'))
```

Make the input string lowercase and then split it into separate parts, using **'-'** as the separator

Example: date_decoder.py

```
def date_decoder(date):
    months = {'jan': 1, 'feb': 2, 'mar': 3, 'apr': 4,
              'may': 5, 'jun': 6, 'jul': 7, 'aug': 8,
              'sep': 9, 'oct': 10, 'nov': 11, 'dec': 12}
    parts = date.lower().split('-')
    day = int(parts[0])
    month = months[parts[1]]
    year = 1900 + int(parts[2])
    if int(parts[2]) <= 69:
        year += 100
    return year, month, day

    print(date_decoder('8-MAR-85'))
    print(date_decoder('17-Apr-25'))
```

Extract the string containing the day (e.g., `'8'`) and convert it to an integer

Example: date_decoder.py

```
def date_decoder(date):
   months = {'jan': 1, 'feb': 2, 'mar': 3, 'apr': 4,
                   'may': 5, 'jun': 6, 'jul': 7, 'aug': 8,
                   'sep': 9, 'oct': 10, 'nov': 11, 'dec': 12}
   parts = date.lower().split('-')
   day = int(parts[0])
   month = months[parts[1]]
   year = 1900 + int(parts[2])
   if int(parts[2]) <= 69:
      year += 100
   return year, month, day

   print(date_decoder('8-MAR-85'))
   print(date_decoder('17-Apr-25'))
```

Extract the string containing the month (e.g., `'MAR'`) and use the dictionary to map it to an integer

Example: date_decoder.py

```
def date_decoder(date):
    months = {'jan': 1, 'feb': 2, 'mar': 3, 'apr': 4,
              'may': 5, 'jun': 6, 'jul': 7, 'aug': 8,
              'sep': 9, 'oct': 10, 'nov': 11, 'dec': 12}
    parts = date.lower().split('-')
    day = int(parts[0])
    month = months[parts[1]]
    year = 1900 + int(parts[2])
    if int(parts[2]) <= 69:
        year += 100
    return year, month, day

    print(date_decoder('8-MAR-85'))
    print(date_decoder('17-Apr-25'))
```

Extract the string containing the year (e.g. '85'), convert it to an integer, then add 1900

Example: date_decoder.py

```
def date_decoder(date):
    months = {'jan': 1, 'feb': 2, 'mar': 3, 'apr': 4,
              'may': 5, 'jun': 6, 'jul': 7, 'aug': 8,
              'sep': 9, 'oct': 10, 'nov': 11, 'dec': 12}
    parts = date.lower().split('-')
    day = int(parts[0])
    month = months[parts[1]]
    year = 1900 + int(parts[2])
    if int(parts[2]) <= 69:
        year += 100
    return year, month, day

    print(date_decoder('8-MAR-85'))
    print(date_decoder('17-Apr-25'))
```

Add 100 to the year if the two-digit year is less than 70

Example: date_decoder.py

```
def date_decoder(date):
    months = {'jan': 1, 'feb': 2, 'mar': 3, 'apr': 4,
              'may': 5, 'jun': 6, 'jul': 7, 'aug': 8,
              'sep': 9, 'oct': 10, 'nov': 11, 'dec': 12}
    parts = date.lower().split('-')
    day = int(parts[0])
    month = months[parts[1]]
    year = 1900 + int(parts[2])
    if int(parts[2]) <= 69:
        year += 100
    return year, month, day

    print(date_decoder('8-MAR-85'))
    print(date_decoder('17-Apr-25'))
```

Return the tuple containing all three parts of the date

Example: date_decoder.py

```
def date_decoder(date):
    months = {'jan': 1, 'feb': 2, 'mar': 3, 'apr': 4,
              'may': 5, 'jun': 6, 'jul': 7, 'aug': 8,
              'sep': 9, 'oct': 10, 'nov': 11, 'dec': 12}
    parts = date.lower().split('-')
    day = int(parts[0])
    month = months[parts[1]]
    year = 1900 + int(parts[2])
    if int(parts[2]) <= 69:
        year += 100
    return year, month, day

    print(date_decoder('8-MAR-85'))   ←   1985, 3, 8
    print(date_decoder('17-Apr-25'))
```

Example: date_decoder.py

```
def date_decoder(date):
    months = {'jan': 1, 'feb': 2, 'mar': 3, 'apr': 4,
              'may': 5, 'jun': 6, 'jul': 7, 'aug': 8,
              'sep': 9, 'oct': 10, 'nov': 11, 'dec': 12}
    parts = date.lower().split('-')
    day = int(parts[0])
    month = months[parts[1]]
    year = 1900 + int(parts[2])
    if int(parts[2]) <= 69:
        year += 100
    return year, month, day

    print(date_decoder('8-MAR-85'))
    print(date_decoder('17-Apr-25'))
```

← 2025, 4, 17

Example: Student Database

A dictionary can contain any data we like – this includes lists

Imagine we wanted to maintain lists of students organized by major

We could make a dictionary where the key is *major* (string) and the *value* for each key is the list of student names (a list of strings)

Consider a function **add_student** that takes three arguments:

1. A dictionary structured as described above
2. The name of a student
3. The major for that student

The function adds the student to the database

Example: Student Database

```
def add_student(majors, student_name, student_major):
    majors.setdefault(student_major, [])
    majors[student_major].append(student_name)
```

The first line initializes the list of students in a major to be the empty list

- This code is executed the first time a new major is encountered

The second line locates the list for a particular major (**majors[student_major]**) and then appends that student's name to the list with **append(student_name)**

Example: Student Database

To see how this function works, first create an empty dictionary:

```
major_dict = {}
```

Then we can call the function to add students one by one:

```
add_student(major_dict, 'Adam', 'CSE')
add_student(major_dict, 'Dave', 'CSE')
add_student(major_dict, 'Chris', 'ECO')
add_student(major_dict, 'Terry', 'AMS')
add_student(major_dict, 'Erin', 'CSE')
add_student(major_dict, 'Frank', 'ECO')
```

major_dict = { **'CSE'**: ['Adam', 'Dave', 'Erin], **'ECO'**: ['Chris', 'Frank'], **'AMS'**: ['Terry'] }

Example: Student Database

We can answer several questions now:

- Who is majoring in Computer Science?

```
cse_majors = major_dict['CSE']
```

- How many students are majoring in Economics?

```
num_econ = len(major_dict['ECO'])
```

A dictionary does not support "reverse lookup"

Multiple keys could actually be mapped to the same value

- For example, if the students have multiple majors

Another example, consider a dictionary where the keys are book titles and the values are authors

- Since a single author might write several books, there is no way with a dictionary to **reverse** the title-to-author mapping and uniquely map authors to book titles

Example: Student Database

- In the prior student example, each student has exactly one major, so we could create a new dictionary that maps students to majors
- To do this we will need to iterate over the keys of the **major_dict** dictionary
- Fortunately, there is a dictionary method that will help with this process: **keys()**

```
student_dict = {}                     # map: student -> major
for major in major_dict.keys():       # for each major:
    for s in major_dict[major]:       # for each student in that major:
        student_dict[s] = major       # record that student's major

student_dict = {'Adam': 'CSE', 'Dave': 'CSE', ...}
```

Example: Acronym Generator (v1)

Let's explore a function that will create an acronym from the first letter of each “long” word in a list

We will define a “long” word to be any word with more than two letters

After studying this first version, we will look at a second version that affords a little extra flexibility in creating acronyms

Example: acronym1.py

```
def acronym(phrase):
    result = ''
    words = phrase.split()
    for w in words:
        if len(w) >= 3:          # keep only long words
            result += w.upper()[0]
    return result
```

Example: Acronym Generator (v2)

Python allows function arguments to have **default values**

- If the function is called without the argument, the argument gets its default value
- Otherwise, the argument's value is given in the normal way

We have seen a few examples of functions that have optional arguments

A good example is the **round()** function, which takes two arguments: the value to round and an optional argument that indicates how many digits after the decimal point we want

- If the second argument is not provided, the number of digits defaults to 0. For example:

```
round(4.56324) = 5
round(4.56324, 2) = 4.56
```

Example: Acronym Generator (v2)

The second version of **acronym** takes an optional argument, **include_shorts**, that tells the function to include the first letter of all words (including short words), but short words will **not** be capitalized if they are included

The first version of **acronym** simply discarded all short words

Example: acronym2.py

```
def acronym(phrase, include_shorts=False):
    result = ''
    words = phrase.split()
    for w in words:
        if len(w) >= 3:
            result += w.upper()[0]
        elif include_shorts:
            result += w.lower()[0]
    return result
```

By default, the optional argument is **False**, causing short words to be excluded

When the optional argument is **True** and **w** is a short word, the first letter of the word in lowercase is concatenated to **result**

Example: acronym() (v2)

Examples:

acronym('United States of America') still returns **'USA'**

acronym('United States of America', True) returns **'USoA'**

Optional Arguments

As another example, suppose we want to make a revised version of the **bmi()** function from earlier in the course:

```
def bmi(weight, height):
    return (weight * 703) / (height ** 2)
```

This version of **bmi()** assumes weight is given in pounds and height in total inches

Suppose instead we want to give the programmer the option to use metric or standard (English) units

- We can add a third, optional argument, **units**, that defaults to metric if the programmer doesn't give a third argument

Let's see the function on the next slide

Example: bmi_v4.py

```
def bmi(height, weight, units = 'metric'):
    if units == 'metric':
        return weight / height**2
    elif units == 'standard':
        return (weight * 703) / (height ** 2)
    else:
        return None
```

Examples:	Return Value:
`bmi(100, 150, 'standard')`	**10.545**
`bmi(100, 150)`	**0.015**
`bmi(100, 150, 'metric')`	**0.015**
`bmi(100, 150, 'unknown')`	**None**

Questions?