'tis the time for final projects...

Today continues our objects+classes theme...
+ it's a *sales pitch* for the three possible *projects*:

- Picobot
- TextID
- vPool

I've got my eyes on some of these projects!

Eye'll bet!
Today continues our objects+classes theme... 
+ it's a *sales pitch* for the three possible projects:

I've got my eyes on some of these projects!

Eye'll bet!

Picobot
TextID
vPool
Final projects

Final CS hw

- open-ended
- comprehensive
- three choices...

Working in teams of 1-3 is OK

Teams need to work together - *in the same place* - and need to share the work equally...

Pairs/trios are welcome (larger should split)

**Teaming is extra-encouraged** on the final project!
Final projects

Project Options, Links, and Details

We have three possible final project themes in CS5. All of the provide the chance for creative expansion of the basic themes...!

Here are the three project descriptions:

- vPool
- Game AI: Connect Four
- TextID
- PicobotProject

Start, Milestone, and Final-project submission

Your **starting point** is due as part of **hw11**

- That is, Monday, April 17th
- The most important objective is to ensure you’ve gotten a start.

The **milestone** is due as part of **hw12**

- The milestone is due Monday, April 24th
- Unfortunately, we won’t be able to provide feedback on these until after
- Its most important goal is to make sure you’ve made progress
- And, if you’ve finished, wonderful!
- You’ll need to submit a zip file named
  - vpython_milestone.zip or
  - textid_milestone.zip or
  - picobot_milestone.zip
The projects...

VPool

Ron Hagrid

TextID

Picobot!
What CSers "do"

or think as much...

final project algorithms...
We have WS and JKR texts – and their word counts! Which one is a better match for the unknown text in the middle? Why? & How?

Final-project algorithms

Brainstorming algorithms + examples...

Word-counts for J.K. Rowling
{ "love": 25, "spell": 275, "potter": 700 }

Word-counts for WS
{ "love": 3, "thou": 1, "potter": 2, "spam": 4 }

Word-counts for another text
{ "love": 25, "spell": 275, "potter": 700 }

How could you write a Python program to create a Picobot program that can solve the empty-room's challenge... without simply giving the answer away?

Sphere-sphere collisions

Meta-programming

Draw arrows to show how b and g would move if they collide off-center.
We have WS and JKR texts – and their word counts! Which one is a better match for the unknown text – in the middle? *Why? & How?*

**Text-comparing**

**WS**

```json
{ "love": 50, "spell": 8, "thou": 42 }
```

_word-counts for W. Shakespeare_

**JKR**

```json
{ "love": 3, "thou": 1, "potter": 2, "spam": 4 }
```

_word-counts for another text_

**JJKR**

```json
{ "love": 25, "spell": 275, "potter": 700 }
```

_word-counts for J.K. Rowling_

**Final-project algorithms**

Draw arrows to show how b and g would move if they collide off center.

**Meta-programming**

How could you write a *Python program* to create a *Picobot program* that can solve this empty-room's challenge... without simply giving the answer away...?
Though Robin Ellacott’s twenty-five years of life had seen their moments of drama and incident, she had never before woken up in the certain knowledge that she would remember the coming day for as long as she lived.
Though Robin Ellacott’s twenty-five years of life had seen their moments of drama and incident, she had never before woken up in the certain knowledge that she would remember the coming day for as long as she lived.
I was given e-text copies of Cuckoo to compare against Rowling’s own The Casual Vacancy, Ruth Rendell’s The St. Zita Society, P.D. James’ The Private Patient and Val McDermid’s The Wire in the Blood. [...] 

I actually ran four separate types of analyses focusing on four different linguistic variables. While anything can in theory be an informative variable, my work focuses on variables that are easy to compute and that generate a lot of data from a given passage of language. One variable that I used, for example, is the distribution of word lengths. Each novel has a lot of words, each word has a length, and so one can get a robust vector of % of the words in this document have exactly letters. Using a distance formula (for the mathematically minded, I used the normalized cosine distance formula instead of the more traditional Euclidean distance you remember from high school), I was able to get a measurement of similarity, with 0.0 being identity and progressively higher numbers being greater dissimilarity.
the TextID project

**Big ideas:**

1. Build *lexical* models of bodies of text...
2. Use a similarity score to measure *Shakepearity* vs. *NYTimes-iness* vs. *WSJournalicity* vs. *even better*: your own choice of two or more comparisons...

- **Big Bang Theory** vs. **Modern Family**
- **NYTimes-iness** vs. **WSJournalicity**
- **Rowlingness** vs. **Shakepearity**

**even better:** *your own choice of two or more comparisons...*
Model matching

Suppose we have two models:

WS: {
  "love": 50,
  "spell": 8,
  "thou": 42
}

JKR: {
  "love": 25,
  "spell": 275,
  "potter": 700
}

Which model does this "mystery" text better match?

Unknown text: {
  "love": 3,
  "thou": 1,
  "potter": 2,
  "spam": 4
}
Model matching

Suppose we have two models:

WS: { "love": 50, "spell": 8, "thou": 42 }

Unknown text: { "love": 3, "thou": 1, "potter": 2, "spam": 4 }

JKR: { "love": 25, "spell": 275, "potter": 700 }

what's the first problem!?
Model matching

Suppose we have two models:

WS: { "love": 50, "spell": 8, "thou": 42 }

JKR: { "love": 25, "spell": 275, "potter": 700 }

models with different TOTALS

Unknown text: { "love": 3, "thou": 1, "potter": 2, "spam": 4 }
Model *matching*

WS: { "love": 50,  
"spell": 8,  
"thou": 42 }  

JKR: { "love": 25,  
"spell": 275,  
"potter": 700 }  

normalize for size

WS: { "love": 0.50,  
"spell": 0.08,  
"thou": 0.42 }  

JKR: { "love": 0.025,  
"spell": 0.275,  
"potter": 0.700 }  

Unknown text: { "love": 3,  
"thou": 1,  
"potter": 2,  
"spam": 4 }  

now, how to compare with these counts?
Model matching

WS: { "love": 50, "spell": 8, "thou": 42 }

JKR: { "love": 25, "spell": 275, "potter": 700 }

normalize for size

WS: { "love": 0.50, "spell": 0.08, "thou": 0.42 }

JKR: { "love": 0.025, "spell": 0.275, "potter": 0.700 }

Unknown text: { "love": 3, "thou": 1, "potter": 2, "spam": 4 }

We multiply the probabilities of each word!

... and see which is higher probability...
Model matching

WS: { "love": 0.50, "spell": 0.08, "thou": 0.42 }

JKR: { "love": 0.025, "spell": 0.275, "potter": 0.700 }

the **WS**-probability of each word in **Unknown text**

```
.50 • .50 • .50 • _?_ • ? • ? • ? • ? • ? • ? • ?
```

```
love  love  love  thou  potter  potter  spam  spam  spam  spam
```

**Unknown text**: { "love": 3, "thou": 1, "potter": 2, "spam": 4 }
**Model matching**

Suppose we have two normalized models:

WS: 

- "love": 0.50
- "spell": 0.08
- "thou": 0.42

JKR:

- "love": 0.025
- "spell": 0.275
- "potter": 0.700

\[
\begin{align*}
0.50 \cdot 0.50 \cdot 0.50 \cdot 0.42 \cdot 1 \cdot 1 \cdot 1 \cdot 1 \cdot 1 \cdot 1 \\
\end{align*}
\]

\[
0.0525
\]

Unknown text:

- "love": 3
- "potter": 2
- "thou": 1
- "spam": 4

Why is this especially incorrect?

I've got near-zero ideas on this one!
Model matching

Suppose we have two normalized models:

WS: { "love": 0.50, "spell": 0.08, "thou": 0.42 }

JKR: { "love": 0.025, "spell": 0.275, "potter": 0.700 }

Correct, but not helpful!

Unknown text: { "love": 3, "thou": 1, "potter": 2, "spam": 4 }

I've got near-zero ideas on this one!
Model matching

WS: { "love": 0.50, "spell": 0.08, "thou": 0.42 }

JKR: { "love": 0.025, "spell": 0.275, "potter": 0.700 }

for missing words, use half the smallest value – across both models!

\[
\begin{array}{cccccccc}
\text{love} & \text{love} & \text{love} & \text{thou} & \text{potter} & \text{potter} & \text{spam} & \text{spam} \\
3 & 1 & 2 & 4 & 0.012 & 0.012 & 0.012 & 0.012 & 0.012 & 0.012 \\
\end{array}
\]

\[= 1.57 \times 10^{-13}\]

\[0.0000000000000157\]

Unknown text: { "love": 3, "potter": 2, "thou": 1, "spam": 4 }
Model matching

WS: {'love': 0.50, 'spell': 0.08, 'thou': 0.42}

JKR: {'love': 0.025, 'spell': 0.275, 'potter': 0.700}

\[ \frac{1}{2} \times \text{minimum value across both models!} \]

\[ \frac{1}{2} \times 0.000000000000157 = 1.57 \times 10^{-13} \]

Unknown text: {'love': 3, 'potter': 2, 'thou': 1, 'spam': 4}

half of \( \varepsilon \)!
Model matching

WS: { "love": 0.50,
    "spell": 0.08,
    "thou": 0.42 }

JKR: { "love": 0.025,
       "spell": 0.275,
       "potter": 0.700 }

\[ 3 \log(0.50) + \log(0.42) + 2 \log(0.012) + 4 \log(0.012) = -29.48 \]

Unknown text: { "love": 3,
                 "potter": 2,
                 "thou": 1,
                 "spam": 4 }
Model matching

WS: \{ "love": 0.50, "spell": 0.08, "thou": 0.42 \}

JKR: \{ "love": 0.025, "spell": 0.275, "potter": 0.700 \}

\[ 3 \log(0.025) + \log(0.012) + 2 \log(0.7) + 4 \log(0.012) = -33.89 \]

Unknown text: \{ "love": 3, "potter": 2, "thou": 1, "spam": 4 \}
Model matching

WS: { "love": 0.50, "spell": 0.08, "thou": 0.42 }

JKR: { "love": 0.025, "spell": 0.275, "potter": 0.700 }

Unknown text:
{ "love": 3, "potter": 2, "thou": 1, "spam": 4 }

-29.48

-33.89

the better match...
You'll use at least 5 models...

```
TextModel

... will contain at least five Python dictionaries:

word counts

{\'and\': 3, \'poptartful\': 1, \'liked\': 1, \'spamful\': 1, \'like\': 2, \':\': 1, \'spam\': 2, \'i\': 1, \'42\': 1, \'all\': 1, 
\'thanksgiving\': 1, \'will\': 1, \'bring\': 1, \'poptarts\': 3, \'spams\': 1, \'by\': 1, \'are\': 2}

word-length counts

{0: 1, 1: 1, 2: 2, 3: 6, 4: 5, 5: 3, 7: 1, 8: 3, 10: 1, 12: 1}
```
I like poptarts and 42 and spam. Spamful poptarts are like poptartful spams -- and are liked by all!

Will _Thanksgiving_ bring spam poptarts?

**TextModel**

... will contain at least five Python dictionaries:

- **word counts**
  
  ```python
  {'and': 3, 'poptartful': 1, 'liked': 1, 'spamful': 1, 'like': 2, ':': 1, 'spam': 2, 'i': 1, '42': 1, 'all': 1, 'thanksgiving': 1, 'will': 1, 'bring': 1, 'poptarts': 3, 'spams': 1, 'by': 1, 'are': 2}
  ```

- **word-length counts**
  
  ```python
  {0: 1, 1: 1, 2: 2, 3: 6, 4: 5, 5: 3, 7: 1, 8: 3, 10: 1, 12: 1}
  ```

- **???-counts**
  
  ```python
  {'and': 3, ':': 1, 'all': 1, 'like': 3, 'thanksgiving': 1, 'spam': 4, 'i': 1, '42': 1, 'by': 1, 'will': 1, 'bring': 1, 'ar': 2, 'poptart': 4}
  ```

- **???-counts**
  
  ```python
  {12: 1, 5: 1, 7: 1}
  ```

- **???-counts**
  
  ```python
  {'!': 1, '-': 2, '?': 1, '_': 2, ':': 1}
  ```

**What are these three other dictionaries counting?!**
I like poptarts and 42 and spam. Spamful poptarts are like poptartful spams -- and are liked by all!
Will _Thanksgiving_ bring spam poptarts?

TextModel
... will contain at least five Python dictionaries:

```
word counts

{'and': 3, 'poptartful': 1, 'liked': 1, 'spamful': 1, 'like': 2, ':': 1, 'spam': 2, 'i': 1, '42': 1, 'all': 1, 'thanksgiving': 1, 'will': 1, 'bring': 1, 'poptarts': 3, 'spams': 1, 'by': 1, 'are': 2}

word-length counts

{0: 1, 1: 1, 2: 2, 3: 6, 4: 5, 5: 3, 7: 1, 8: 3, 10: 1, 12: 1}

word-stem-counts

{'and': 3, ':': 1, 'all': 1, 'like': 3, 'thanksgiv': 1, 'spam': 4, 'i': 1, '42': 1, 'by': 1, 'will': 1, 'bring': 1, 'ar': 2, 'poptart': 4}

sentence-length-counts

{12: 1, 5: 1, 7: 1}

punctuation-counts

{':': 1, '-': 2, '?': 1, '_': 2, '.': 1}

dictionaries count!
TextID's building blocks...

1. Split up the text into words (first pass)
2. Model punctuation marks (optional)
3. Model sentence lengths (using '. ! ? ')
4. Take a breather...
5. Clean up the words (second pass)
6. Model words and word lengths
7. Stem words and model those stems
8. You're ready to score against your model!
Stemming

An algorithm that outputs the **root** of the input word.

<table>
<thead>
<tr>
<th>Stem Function</th>
<th>Result</th>
<th>Example</th>
<th>Rule</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>stem('parties')</code></td>
<td>'parti'</td>
<td>ends in <code>ies</code></td>
<td></td>
</tr>
<tr>
<td><code>stem('love')</code></td>
<td>'lov'</td>
<td>ends in <code>vow + cons + e</code></td>
<td></td>
</tr>
<tr>
<td><code>stem('swerving')</code></td>
<td>'swerv'</td>
<td>ends in <code>vow + cons + ing</code></td>
<td></td>
</tr>
<tr>
<td><code>stem('quickly')</code></td>
<td>'quick'</td>
<td>ends in <code>vow + cons + ly</code></td>
<td></td>
</tr>
<tr>
<td><code>stem('slowest')</code></td>
<td>'slow'</td>
<td>ends in <code>vow + cons + est</code></td>
<td></td>
</tr>
<tr>
<td><code>stem('stems')</code></td>
<td>'stem'</td>
<td>ends in <code>cons + s</code></td>
<td></td>
</tr>
<tr>
<td><code>stem('stemming')</code></td>
<td>'stem'</td>
<td>ends in <code>vow + 2*cons + ing</code></td>
<td></td>
</tr>
</tbody>
</table>

these don't have to be words, just **stems**...

but, they **can** also be words

either way, they all have exceptions!
Stemming!

def stem(w):

- pies  stem('parties')  → 'parti'
- kite  stem('love')  → 'lov'
- nothing  stem('swerving')  → 'swerv'
- ally  stem('quickly')  → 'quick'
- stem('slowest')  → 'slow'
- stem('stems')  → 'stem'
- lemming  stem('stemming')  → 'stem'
- under  stem(____undo____)  → ___do___
Stemming!

def stem( w ):  

exception
Find an English exception to each of these stemming patterns:

pies  
stem('parties') \rightarrow 'parti'  
ends in ies

kite  
stem('love') \rightarrow 'lov'  
ends in vow + cons + e

nothing  
stem('swerving') \rightarrow 'swerv'  
ends in vow + cons + ing

ally  
stem('quickly') \rightarrow 'quick'  
ends in vow + cons + ly

interest  
stem('slowest') \rightarrow 'slow'  
ends in vow + cons + est

princess  
stem('stems') \rightarrow 'stem'  
ends in cons + s

lemming  
stem('stemming') \rightarrow 'stem'  
ends in vow + 2*cons + ing

under  
stem(____undo____) \rightarrow  ____do____
create your own rule for stemming

incorrect stem

\pi  
kit  
noth  
al  
inter  
princes  
lem  
der
TextID's library resources

7.1. string — Common string operations

5.6.1. String Methods

string.punctuation
String of ASCII characters which are considered punctuation

\texttt{str.replace(old, new[, count])}
Return a copy of the string with all occurrences of substring \texttt{old} replaced by \texttt{new}.

\texttt{str.lower()}
Return a copy of the string with all the cased characters [4] converted to lowercase.

\texttt{str.split([sep[, maxsplit]])}
Return a list of the words in the string, using \texttt{sep} as the delimiter string. If \texttt{maxsplit} is given, at most \texttt{maxsplit} splits are done (thus, the list will have at most \texttt{maxsplit+1} elements). If \texttt{maxsplit} is not specified or -1, then there is no limit on the number of splits (all possible splits are made).

If \texttt{sep} is given, consecutive delimiters are not grouped together and are deemed to delimit empty strings (for example, \texttt{"1,2".split(",")} returns [\texttt{"1"}, \texttt{"", \"2\"}]). The \texttt{sep} argument may consist of multiple characters (for example, \texttt{"1<>2<>3".split("<>")} returns [\texttt{"1"}, \texttt{"2"}, \texttt{"3"}]). Splitting an empty string with a specified separator returns [\texttt{""}].
7.1. string — Common string operations

5.6.1. String Methods

string.ascii_lowercase

I bet they told you the age of the library was over... but it's just begun!
Naïve Bayes classification

Bayesian spam filtering

Bayesian spam filtering (ˈbeɪzɪən/ bay-zee-ən; after Rev. Thomas Bayes) is a statistical technique of e-mail filtering. In its basic form, it makes use of a naïve Bayes classifier on bag of words features to identify spam e-mail, an approach commonly used in text classification.

Constructing a classifier from the probability model

The discussion so far has derived the independent feature model, that is, the naïve Bayes classifier. The naïve Bayes classifier combines this model with a decision rule. One common decision rule is to choose the hypothesis that is most probable; this is known as the maximum a posteriori decision rule. The corresponding classifier, a Bayes classifier, is the function classifying with this rule:

\[
\text{classify}(f_1, \ldots, f_n) = \arg \max_c p(C = c) \prod_{i=1}^{n} p(F_i = f_i|C = c).
\]

Don't take these formulas too seriously...
Naïve Bayes classification

Bayesian spam filtering (ˈbɛzɪən/ bay-zee-ən; after Rev. Thomas Bayes) is a statistical technique of e-mail filtering that makes use of a naïve Bayes classifier to identify spam. A naïve Bayes classifier assumes that the presence of a particular feature of a class is unrelated to the presence of any other feature. The corresponding classifier, a Bayes classifier, is the function

\[
\text{classify}(f_1, \ldots, f_n) = \arg \max_c p(C = c) \prod_{i=1}^{n} p(F_i = f_i|C = c).
\]

This quantity is called "spamicity" (or "spaminess") of the word "replica", and can be computed. The number of words in a representation determines the complexity of the classifier, and can be computed. The naive Bayes hypothesis is that the most probable hypothesis that is most likely to be true given the evidence.

Don't take these formulas too seriously...
to finish where we started...?
The Picobot project

Big ideas

(1) Implement Picobot in Python
(2) *Train Python to write successful Picobot programs!*

talk about going *full circle...*
Picobot *returns*!

What data structures (classes) might be helpful in implementing Picobot?
Picobot program

What in Python could most usefully hold all of these *rules*?

What *type* should *rules* be?

*We have some known information ... and want to look things up, based on that!*
class Program:

What in Python could most usefully hold all of these rules?

What type should rules be?

rules[(1, "NExx")]= ("W", 0)
Picobot program

What type in Python could most usefully hold the *environment*?

```python
class World:

    # What class that you've already written will be most similar to Picobot's World?

    # What will *room* be?
```

```
+oooooooooPooooooooooooooooo+
+o             o o+
+o             o o+
+o             o o+
+o             o o+
+o             o o+
+o             o o+
+o             o o+
+o                     o+
+o                     o+
+o                     o+
+o                     o+
+o                     o+
+o                     o+
+o                     o+
+o                     o+
+o                     o+
+o                     o+
+o                     o+
+o                     o+
+o                     o+
+o                     o+
+o                     o+
+ooooooooooooooooooooooo+
+ooooooooooooooooooooooo+
```

Wall: +
Visited: O
Picobot: P
Picobot program

What type in Python could most usefully hold the environment?

```python
class World:

    What class that you've already written will be most similar to Picobot's World?

What will room be?
```

A 2d data structure (like mset!)

A list-of-lists-of-one-character-strings....
The Picobot project

Current State: 1
Current Rule: 1 N*W* -> X 2

First, build an **ASCII simulation**

Picobot started here...

then, **evolve** it...

with your own **genetic algorithm**

and is now here...

http://rednuht.org/genetic_cars_2/ or http://boxcar2d.com/
Box2d: https://www.youtube.com/watch?v=uxourlPIf8

Your ASCII is likely to be more monochromatic!
Genetic Algorithms...

*genetic algorithms* are for optimizing *hard-to-describe functions* with *easily-splittable solutions*.

the current "fitness" leader

the current "generation"

fitnesses!

Rafael Matsunaga: rednuht.org/
Program evolution

An example of genetic algorithms, which are used for optimizing hard-to-describe functions with easily-splittable solutions.

Start with a population of, say, ~200 random Picobot programs...

Then, measure the fitness of all of those programs...
Program *evolution*

An example of *genetic algorithms*, which are used for optimizing *hard-to-describe functions* with *easily-splittable solutions*.

program p1

fitness = 0.03

program p2

fitness = 0.05

*fitness ~ coverage*
mate + mutate the fittest rulesets
to create a new generation of ~200 programs...

program p1
fitness = 0.03

program p2
fitness = 0.05

program c1
fitness = 0.19

What the goal?
Repeat this "survival of the fittest" process for many generations...

... and by the end, your Python code should/will have evolved a much more capable Picobot program!
**Extra:** Picobot graphics...

Choice of graphical packages
- 2d: graphics.py from C4
- 3d: VPython et al.

2d is **flat-out** awesome!

All the best stuff is 3D: Coffee, spam, poptarts, etc.

Mine's going to be in 5d!
open ended...
Project space...

C4 AI

Picobot!

TextID

VPool

algorithmic

open-ended
(and 3d!)

practical / checkable
3d graphics-based game using VPython

Let's play!

3d graphics-based game using VPython

I'll take your cue.

vPool

... it's not really very constrained at all!

A few constraints...

need \( \geq 4 \) physically interacting objects

allow the user to direct 1+ objects, either by keyboard or mouse or both

needs a game goal + be winnable!

must detect some "linear" and some "spherical" collisions and implement their results on the motion of the objects
The vPool project

To now, VPython has **eventually** worked for everyone. *See us for help!*

- *Linear collisions* should be somewhere ("walls")
- *Spherical collisions* should be somewhere ("points")
- You need "pockets" – *or some other game objective*
- You need **user control** of at least one object (mouse/kbd)

VPython was designed to make 3d physics simulations simpler to program – as a result, the library is physics-free!

⇒ Phunky Physics is welcome!

*A few examples to get you thinking...*
Seen before?

https://www.youtube.com/watch?v=SmFEK2gq4QQ
Scene before!

https://www.youtube.com/watch?v=SmFEK2gq4QQ
Enjoy the projects! ... the graders certainly do!

An unusual variation on VPool 🐘
Enjoy the projects! ...

An unusual variation on VPool ...

the graders certainly do!

Questions? Thoughts? Let's chat!

Good luck next!