Introduction to modeling, and Perl

24.964—Fall 2004
Modeling phonological learning

Class 1 (9 Sept 2004)

Introduction

(Syllabus and mechanics)

Why learn to model?

Example: describing phonotactics

	Tagalog	English	Polish
[ta]	✓	✓	✓
[tra]	*	✓	✓
[rta]	*	*	✓

Why learn to model?

The “popular model”

- Children hear what their language sounds like, and they use their knowledge of existing words to decide about what’s possible
- Tagalog speakers: don’t know any words with [tra], so reject it as zero probability
- English speakers: know both [ta] and [tra] words, but no [rta]; reject as highly improbable (or impossible)
- Polish-learning children: know words of all types, so find support for accepting all three

Why learn to model?

Tjong Kim Sang & Nerbonne (2000) *Learning the logic of simple phonotactics*

- Took a corpus of existing Dutch words
- Model looks at each word, noting what segments can occur next to one another
  - [praːt]: infers that [pr], [raː], [aːt] are allowable sequences
- Testing whether a new word is possible: does it contain any two-character sequences that haven't been seen before?
- Model trained on most of the words in the corpus
  - A few words set aside for testing (test positives)
  - Testing also includes randomly generated words with illegal sequences (test negatives)

Why learn to model?

Tjong Kim Sang & Nerbonne (2000)

	Simple model	
Task	% accepted positives	% rejected negatives
Orthographic	99.3±0.3	55.7±0.9
Phonetic	99.0±0.5	76.8±0.5

Why learn to model?

Tjong Kim Sang & Nerbonne (2000)

- Then trained a model, which tried to learn rules about possible combinations (not just possible two-character sequences)

Why learn to model?

Results of baseline model

	Simple model	
Task	% accepted positives	% rejected negatives
Orthographic	99.3±0.3	55.7±0.9
Phonetic	99.0±0.5	76.8±0.5

Why learn to model?

Results of rule-learning model

	Simple model	
Task	% accepted positives	% rejected negatives
Orthographic	99.3±0.3	55.7±0.9
Phonetic	99.0±0.5	74.8±0.5

Why learn to model?

Tjong Kim Sang & Nerbonne (2000)

- Finally, augmented their model to incorporate some notion of syllable structure

$$C_1\ C_2\ C_3\ V\ C_4\ C_5\ C_6$$

- Can't have C_1 without C_2, C_3 without C_2, C_1 can't be a stop, etc.

Why learn to model?

Results of augmented model:

	Simple model	
Task	% accepted positives	% rejected negatives
Orthographic	98.6±0.3	84.9±0.3
Phonetic	99.0±0.5	91.9±0.3

Why learn to model?

Konstantopoulos (2002) *Learning Phonotactics Using ILP*

- Similar task, slightly different model
- Model also tries to learn rules about what can come before/after what

Primitives	% accepted pos	% rejected neg	# of rules
Segments	99.3%	79.8%	1154
Feature classes	94.2%	92.6%	181
Sonority relations	93.1%	83.2%	11

Why learn to model?

Gildea and Jurafsky (1996) Learning Bias and Phonological-Rule Induction

- Attempted to train models to learn simple phonological rules of English, such as flapping
  - t → ɾ / V́ (r) _ V (flap medially after an unstressed V and an optional r)

- All that the rule cares about is stress, possible r's, and presence of a following vowel.

- Model must learn to ignore everything else.

Why learn to model?

Gildea and Jurafsky (1996)

Training items	States	Error rate
6250	19	2.32%
12500	257	16.40%
25000	141	4.46%
50000	192	3.14%

- Model fails to improve, even after VERY many examples

Why learn to model?

Gildea and Jurafsky (1996)

- Added bias for segments to remain unaltered by rules ($\approx$ Faithfulness)

Why learn to model?

Gildea and Jurafsky (1996)

Training items	States	Error rate
6250	3	0.34%
12500	3	0.14%
25000	3	0.06%
50000	3	0.01%

- Performing optimally even at earliest testing stage

Why learn to model?

Albright and Hayes (2003)

- Task: learn how to form English past tenses
- Approach: examine the changes involved (suffixation, vowel changes, etc.), and evaluate how reliable/accurate they are

Why learn to model?

Albright and Hayes (2003)

- A surprising result: the rule with the best trade-off of accuracy and generality

$$\varnothing \rightarrow \text{t} \;/\; \begin{bmatrix} -\text{son} \\ +\text{cont} \\ -\text{voi} \end{bmatrix} \_$$

- A failing of the model? Or an empirical discovery?

Why learn to model?

- "Good analytical hygiene"
- Novel evidence for empirical usefulness of theoretical proposals
- Novel evidence for analytical usefulness of theoretical proposals
- Source of novel empirical discoveries

Introduction to Perl

What does the following program do?

```
$n=q y$$YVAR;;y;$q=$n=~y%$N-ZA-M;%_A-Z_%;;print map{eval
join$/,(map{";#"}(2..$_)),qq@\$p=$n@;chr$p+$q}qw &64
93 100 100 103 24 111 103 106 100 92 25 2&
```

Introduction to Perl

What does the following program do?

```
$n=q y$$YVAR;;y;$q=$n=~y%$N-ZA-M;%_A-Z_%;;print map{eval
join$/,(map{";#"}(2..$_)),qq@\$p=$n@;chr$p+$q}qw &64
93 100 100 103 24 111 103 106 100 92 25 2&
```

- This may be the kind of thing you imagine when you think of computer programming
- Don't worry! We won't be doing anything remotely like this in this class

Introduction to Perl

What does the following program do?

```
print "Hello world!\n";
```

Perl trivia

- Stands for *Practical Extraction and Report Language*

Perl trivia

- Stands for *Practical Extraction and Report Language*
- Creator: Larry Wall
  - Attended grad school in linguistics (UCLA, UC Berkeley)
  - (Was an aspiring missionary at the time)

Introduction to Perl

Basic mechanics:

- Perl programs are simply text files, containing lists of instructions
  - You can create them with Notepad, TextEdit, Microsoft Word, etc. (save as text only)
  - (It will save you time and hassle to download and install one that's more powerful, and intended for programming—more on this in a minute)
- In order to run them, you call the *Perl interpreter*
  - This is a (free) program, which you may need to install—more on this in a minute, too

Get a good text editor

(Notepad/TextEdit/etc. will do the trick, but in the long run it pays to get something more sophisticated)

- Unix: Emacs, vi, ...
- Mac: I recommend AlphaX
  - http://www.maths.mq.edu.au/~steffen/Alpha/AlphaX/
- Windows: SciTE is good
  - http://scintilla.sourceforge.net/SciTEDownload.html

Getting Perl

- Unix, Mac OS X: you have it already, by default
- Windows: ActivePerl distribution
  - http://www.activestate.com/Products/ActivePerl/
- Older Mac systems: MacPerl
  - http://www.ptf.com/macperl/

Creating and running a program

hello1.pl

```
print "Hello world!\n";
```

Creating and running a program

hello1.pl

```
print "Hello world!\n";
```

Creating and running a program

hello1.pl

```
print "Hello world!\n";
```

Creating and running a program

hello1.pl

```
print "Hello world!\n";
```

Creating and running a program

hello1.pl

```
print "Hello world!\n";
```

Creating and running a program

hello1.pl

```
print "Hello world!\n";
```

Using variables to store text

hello2.pl

```
$greeting = "Hello world!";
print "$greeting\n";
```

- The simplest type of variable in Perl is one that holds a single value (number, bit of text, etc)
- *Scalar* variable: indicated with $

Using variables to store text

Assigning a value to a variable:

```
$variablename = value;
```

- Value can be a number, a string, a variable, etc.
  - `$days_in_a_week = 7;`
  - `$my_name = "Adam";`
  - `$name_of_user = $my_name;`

Using variables to store text

hello2b.pl

```
$world = "Hello";
$hello = "world!";
print "$world $hello\n";
```

Using variables to store text

Another type of variable: arrays

item 1	item 2	item 3	...	item *n*

```
$greeting = "Hello world!";
print "$greeting\n";
```

- An array is indicated with @ (`@arrayname`)
- Individual elements in the array are referred to by their position (or *index*: `$arrayname[0]`, `$arrayname[1]`, etc.

Using variables to store text

hello3.pl

```
$greeting[0] = "Hello";
$greeting[1] = "world!";
# The following two lines do exactly the same thing
print "$greeting[0] $greeting[1]\n";
print "@greeting\n";
```

Using variables to store text

Assigning values to an array:

- One technique:

  ```
  $arrayname[0] = $item1;
  $arrayname[1] = $item2;
  ```

  etc...

- Another technique:

  ```
  @arrayname = ($item1, $item2, etc...);
  ```

Using variables to store text

hello3b.pl

```
@greeting = ("Hello", "world");
# The following two lines do exactly the same thing
print "$greeting[0] $greeting[1]\n";
print "@greeting\n";
```

Manipulating variables

simplemath.pl

```
$x = 1;
print "The value of \$x is $x\n";
$x = $x + 2;
print "The value of \$x is $x\n";
$x = $x * 2;
print "The value of \$x is $x\n";
$x = $x / 3;
print "The value of \$x is $x\n";
$x = $x - 1;
print "The value of \$x is $x\n";
$x++;
print "The value of \$x is $x\n";
$x--;
print "The value of \$x is $x\n";
```

Manipulating variables

One other useful operation: concatenation

```
$greeting = "Hello" . " " . "world!";
```

Using loops

loop1.pl

```
# A for loop from 1 to 10
for ($i = 1; $i < 11; $i++) {
     print "$i\n";
}
```

Using loops

```
for (initial state, condition, operation) { ...}
```

- Here, initial state is for $i to have value of 1
- Condition is to keep going as long as $i is less than 11
  - `x < y` means x is less than y
  - `x <= y` means x is less than or equal to y
  - Similarly, `x > y`, `x >= y`: x greater than (or equal to) y
  - `x == y` means x equals y
- Each time we run the loop, we add one to $i (`$i++`)
- The stuff to do is between curly braces: `{...}`

Using loops

loop1.pl

```
# A for loop from 1 to 10
for ($i = 1; $i < 11; $i++) {
     print "$i\n";
}
```

How could we modify this program to do the same thing?

Using loops to access arrays

hello4.pl

```
@greeting = ("Hello", "world!");
for ($i = 0; $i <= 1; $i++) {
     print "$greeting[$i] ";
}
print "\n";
```

Using loops to access arrays

hello5.pl

```
@greeting = ("Hello", "world!");
for ($i = 0; $i <= $#greeting; $i++) {
     print "$greeting[$i] ";
}
print "\n";
```

- `$#arrayname` refers to the index of the last element in the array

Putting it together

cv.pl

```
@consonants = ('p','t','k','b','d','g','f','s','z','m','n',
               'l','r');
@vowels = ('a','e','i','o','u');
# Let's also keep track of how many words we have generated
$number_of_words = 0;
# Loop through consonants
for ($c = 0; $c <= $#consonants; $c++) {
     # Loop through vowels
     for ($v = 0; $v <= $#vowels; $v++) {
          # Print out this CV combination
          print "$consonants[$c]$vowels[$v]\n";
          # Add one to the number of words
          $number_of_words++;
     }
}
print "\nGenerated a total of $number_of_words words\n";
```

Putting it together

How would you generate words with CVCV structure?

Putting it together

How would you generate words with CVCV structure?

```
C1: Loop through all possible consonants
    V1: Loop through all possible vowels
        C2: Loop through all possible consonants
            V2: Loop through all possible consonants
                print C1V1C2V2
            End V2 loop
        End C2 loop
    End V1 loop
End C1 loop
```

Putting it together

cvcv.pl

```
@cons = ('p','t','k','b','d','g','f','s','z','m','n','l','r');
@vow = ('a','e','i','o','u');
$number_of_words = 0;
for ($c1 = 0; $c1 <= $#cons; $c1++) {
    for ($v1 = 0; $v1 <= $#vow; $v1++) {
        for ($c2 = 0; $c2<= $#cons; $c2++) {
            for ($v2 = 0; $v2<= $#vo; $v2++) {
                print "$cons[$c1]$vow[$v1]$cons[$c2]$vow[$v2]\n";
                # Add one to the number of words
                $number_of_words++;
            }
        }
    }
}
print "\nGenerated $number_of_words legal words\n";
```

Checking conditions

Task: filter out CVCV words where C1=C2

```
  if (condition) { ...}
```

`$x == $y`	`x` equals `y` (numeric)
`$x != $y`	`x` doesn't equal `y` (numeric)
`$x eq $y`	`x` equals `y` (strings)
`$x ne $y`	`x` doesn't equal `y` (strings)

(Also `$x > $y`, `$x < $y`, `$x >= $y`, `$x <= $y` for numbers)

Checking conditions

Other control structures:

- `if (`*condition*`) { ...}`
- `if (`*condition*`) { ...}`
  `else { ...}`
- `if (`*condition*`) { ...}`
  `elsif (`*condition*`) { ...}`
  `else { ...}`
- `unless (`*condition*`) { ...}`

(We'll see more later)

Checking conditions

cvcv2.pl

```
@cons = ('p','t','k','b','d','g','f','s','z','m','n','l','r');
@vow = ('a','e','i','o','u');
$number_of_words = 0;
for ($c1 = 0; $c1 <= $#cons; $c1++) {
     for ($v1 = 0; $v1 <= $#vow; $v1++) {
          for ($c2 = 0; $c2<= $#cons; $c2++) {
               for ($v2 = 0; $v2<= $#vow; $v2++) {
                    if ($c1 eq $c2) {
                         print "*$cons[$c1]$vow[$v1]$cons[$c2]$vow[$v2]\n";
                    } else  {
                         print "$cons[$c1]$vow[$v1]$cons[$c2]$vow[$v2]\n";
                         # Add one to the number of words
                         $number_of_words++;
                    }
               }
          }
     }
}
print "\nGenerated a total of $number_of_words words\n";
```

Checking conditions

cvcv2b.pl

```
@cons = ('p','t','k','b','d','g','f','s','z','m','n','l','r');
@vow = ('a','e','i','o','u');
$number_of_words = 0;
for ($c1 = 0; $c1 <= $#cons; $c1++) {
     for ($v1 = 0; $v1 <= $#vow; $v1++) {
          for ($c2 = 0; $c2<= $#cons; $c2++) {
               for ($v2 = 0; $v2<= $#vow; $v2++) {
                    if ($c1 ne $c2) {
                         print "$cons[$c1]$vow[$v1]$cons[$c2]$vow[$v2]\n";
                         # Add one to the number of words
                         $number_of_words++;
                    }
               }
          }
     }
}
print "\nGenerated a total of $number_of_words words\n";
```

Summary so far

We have learned the Perl syntax for:

- Storing and accessing values in variables (scalars, arrays)
- Using loops to actions repeatedly
- Checking values, and performing actions based on the result

Pattern matching

Strategy used in `cvcv2.pl` for detecting OCP violation:

- When constructing CVCV string, compare current C1 and C2
- If identical, don't output the string

Another plausible strategy:

- Construct the current CVCV string
- Examine results, looking for $\mathrm{C}_i \ldots \mathrm{C}_i$ sequence (that is, identical C's separated by at least a vowel)
- If found, don't output the string

Pattern matching

Looking for a string within another string:

```
if ($mystring =~ m/searchstring/) { ... }
```

Or, simply:

```
if ($mystring =~ /searchstring/) { ... }
```

Pattern matching

A few things to learn as you need them:

- `[ab]` means "either a or b" (a, b); this can be expanded, so `[abc]` = either a, b, or c, etc...
- `[^a]` means "anything other than a"; `[^ab]` means "anything other than an a or a b", etc. (set negation)
- `a*` means "any number of a's (from 0 to infinity)" (nothing, a, aa, aaa, aaaa, aaaaa, ...)
- `a+` means "one or more a's" (a, aa, aaa, aaaa, aaaaa, ...)

- `ab+` means “an a, followed by one or more b’s” (ab, abb, abbb, abbbb, ...)

- `(ab)+` means “one or more consecutive occurrences of ab” (ab, abab, ababab, abababab, ...)

- `a?` means “an optional a”

- `^a` means “an a at the beginning of the string”

- `a$` means “an a at the end of the string”

- `.` (a period) means “any character”

Pattern matching

patternmatch.pl

```
if ("blah" =~ /a/) {
    print '/a/' . "\n";
}
if ("blah" =~ /^a/) {
    print '/^a/' . "\n";
}
if ("blah" =~ /ba/) {
    print '/ba/' . "\n";
}
if ("blah" =~ /b.a/) {
    print '/b.a/' . "\n";
}
if ("blah" =~ /[a-h]*/) {
    print '/[a-h]*/' . "\n";
}
if ("blah" =~ /^[a-h]*$/) {
    print '/^[a-h]*$/' . "\n";
}
if ("blah" =~ /[a-m]*/) {
    print '/[a-m]*/' . "\n";
}
if ("blah" =~ /^[a-m]*$/) {
    print '/^[a-m]*$/' . "\n";
}
```

Pattern matching

Reminder: cvcv2.pl

```
@cons = ('p','t','k','b','d','g','f','s','z','m','n','l','r');
@vow = ('a','e','i','o','u');
$number_of_words = 0;
for ($c1 = 0; $c1 <= $#cons; $c1++) {
     for ($v1 = 0; $v1 <= $#vow; $v1++) {
          for ($c2 = 0; $c2<= $#cons; $c2++) {
               for ($v2 = 0; $v2<= $#vow; $v2++) {
                    if ($c1 eq $c2) {
                         print "*$cons[$c1]$vow[$v1]$cons[$c2]$vow[$v2]\n";
                    } else  {
                         print "$cons[$c1]$vow[$v1]$cons[$c2]$vow[$v2]\n";
                         # Add one to the number of words
                         $number_of_words++;
                    }
               }
          }
     }
}
print "\nGenerated a total of $number_of_words words\n";
```

Pattern matching

cvcv3.pl

```
@cons = ('p','t','k','b','d','g','f','s','z','m','n','l','r');
@vow = ('a','e','i','o','u');
$number_of_words = 0;
for ($c1 = 0; $c1 <= $#cons; $c1++) {
   for ($v1 = 0; $v1 <= $#vow; $v1++) {
      for ($c2 = 0; $c2<= $#cons; $c2++) {
         for ($v2 = 0; $v2<= $#vow; $v2++) {
            $word = "$cons[$c1]$vow[$v1]$cons[$c2]$vow[$v2]";
            unless ($word =~ /$cons[$c1].$cons[$c1]/) {
               print "$word\n";
            }
         }
      }
   }
}
```

Pattern matching

cvcv4.pl

```
@cons = ('p','t','k','b','d','g','f','s','z','m','n','l','r');
@vow = ('a','e','i','o','u');
$number_of_words = 0;
for ($c1 = 0; $c1 <= $#cons; $c1++) {
   for ($v1 = 0; $v1 <= $#vow; $v1++) {
      for ($c2 = 0; $c2<= $#cons; $c2++) {
         for ($v2 = 0; $v2<= $#vow; $v2++) {
            $word = "$cons[$c1]$vow[$v1]$cons[$c2]$vow[$v2]";
            if ($word =~ /$cons[$c1].$cons[$c1]/) {
               print "$word\tC1=C2\n";
            } elsif ($word =~ /$vowels[$v1].$vowels[$v1]/) {
               print "$word\tV1=V2\n";
            } elsif ($word =~ /[pbmf].[pbmf]/) {
               print "$word\tTwo labials\n";
            } elsif ($word =~ /[iu]$/) {
               print "$word\tFinal high vowel\n";
```

```
                } else { print "$word\n"; }
            }
        }
    }
}
```

Dealing with files

readfile1.pl

```
#Read a file, print its line to the screen.
$input_file = "sample.txt";
open (INFILE, $input_file) or die "The file $input_file could not be found\n";

# Loop, continuing as long as lines can be read from the file
while ($line = <INFILE>)
{
  $line_count++;
  print "$line_count  $line";
}

close INFILE;
```

Dealing with files

readfile2.pl

```
#Read a file, print its line to the screen.
$input_file = "sample.txt";
$output_file = "sample-output.txt";

open (INFILE, $input_file) or die "The file $input_file couldn’t be found\n";
open (OUTFILE, ">$output_file") or die "The file $output_file couldn’t be written\n";

# Loop, continuing as long as a line can be read successfully from the file
while ($line = <INFILE>)
{
  $line_count++;
  printf OUTFILE "$line_count  $line";
}

close INFILE;
close OUTFILE;
```

What would you think this program should do?

readfile3.pl

```
$input_file = "sample.txt";
$output_file = "sample-output.txt";

open (INFILE, $input_file) or die "The file $input_file couldn't be found\n";
open (OUTFILE, ">$output_file") or die "The file $output_file couldn't be written\n";

# Loop, continuing as long as a line can be read successfully from the file
while ($line = <INFILE>)
{
    $count = 0;
    $lines++;
    while ($line =~ m/[aeiou]/) {
        $count++;
    }
    print "Line $lines: $count vowels\n";
}

close INFILE;
close OUTFILE;
```

What would you think this program should do?

readfile3b.pl

```
$input_file = "sample.txt";
$output_file = "sample-output.txt";

open (INFILE, $input_file) or die "The file $input_file couldn’t be found\n";
open (OUTFILE, ">$output_file") or die "The file $output_file couldn’t be written\n";

# Loop, continuing as long as a line can be read successfully from the file
while ($line = <INFILE>)
{
    $count = 0;
    $lines++;
    while ($line =~ m/[aeiou]/g) {
        $count++;
    }
    print "Line $lines: $count vowels\n";
}

close INFILE;
close OUTFILE;
```

Some more useful operations

`chomp($x)`	removes newline (`\n`) from end of line
`lc($x)`	converts $x to lower case
`@fields = split(/\t/, $x)`	splits string $x into an array, using tab as a delimiter
`($var1, $var2) = split(/\t/, $x)`	assigns split fields to different variables
`$x =~ s/search/replace/`	searches $x for `search` and replaces with `replace` (1st instance only)
`$x =~ s/search/replace/g`	searches $x for `search` and replaces with `replace` (all instances)

Exercise

What would be some other ways to count the number of vowels in each line?

Another exercise

Read in a file of arithmetic statements, and check to see whether they are correct.

x OPERATION $y = z$

(checkmath.pl)

Last exercise for the day

Converting romanized Japanese text from the “official” Kunrei-shiki (Manbushō) romanization scheme to the more commonly used Hepburn scheme.

Details at: http://en.wikipedia.org/wiki/Romaji

Last exercise for the day

```
$input_file = "Japanese-ToConvert.txt";
open (INFILE, $input_file) or die "Warning! Can’t open input file: $!\n";

while ($line = <INFILE>) {
    # Crucial rule ordering: this needs to go first
    $line =~ s/hu/fu/g;

    # The major difference is use of <y> after t,s,z
    $line =~ s/ty/ch/g;
    $line =~ s/sy/sh/g;
    $line =~ s/zy/j/g;
    # Also, palatalization before i
    $line =~ s/ti/chi/g;
    $line =~ s/si/shi/g;
    $line =~ s/zi/ji/g;
    # And assibilation of t before u
    $line =~ s/tu/tsu/g;

    print "$line";
}
```

Assignment

Grapheme to phoneme conversion, for Italian

Resources for learning Perl

- On-line documentation:
  - http://www.perl.com/pub/q/documentation
- Other on-line resources
  - http://learn.perl.org
- Wall, Christiansen & Orwant: *Programming Perl (3rd ed.)*
  - Comprehensive, readable; somewhat expensive ($50)