Lecture 2: Regular Expressions, Language Models, and Perl

Lecturer: Mark Hasegawa-Johnson (jhasegaw@uiuc.edu)
TA: Sarah Borys (sborys@uiuc.edu)

May 19, 2005

1 Bash, Sed, Awk

1.1 Installation

If you are on a unix system, bash, sed, gawk, and perl are probably already installed. If not, ask your system administrator.

If you are on Windows, download the Cygwin setup program from http://www.cygwin.com. Choose the "advanced installation" option, so that you can choose to install useful programs such as perl that would not be installed by default. cygwin installation can be run as many times as you like; anything already installed on your PC will not be re-installed. In the screen that asks you which pieces of the OS you want to install, be sure to select (DOC)→(man), (Interpreters)→(gawk,perl), and (TEXT)→(less). I also recommend (Math)→(bc), a simple text-based calculator, and (Network)→(inetutils,openssh). You can also install a complete X-windows server and set of clients from (XFree86)→(fvwm,lesstif,Xfree86-base,XFree86-startup,etc.), allowing you to (1) install X-based programs on your PC, and (2) run X-based programs on any unix computer on the network, with I/O coming from windows on your PC. Setting up X requires a little extra work; see http://xfree86.cygwin.com.

If cygwin is installed in your computer in the directory c:/cygwin, it will create a stump of a unix hierarchy starting in that directory. For example, the directory c:/cygwin/usr/local/bin is available under cygwin as /usr/local/bin. Arbitrary directories elsewhere on your c: drive are available as /cygdrive/c/....

In order to use cygwin effectively, you need to set the environment variables HOME (to specify your home directory), DISPLAY (if you are using X-windows), and most importantly, PATH (to specify directories that should be searched for useful programs. This list should include at least /bin;/usr/bin;/usr/local/bin;/usr/X11R6/bin).

In order to use bash, sed, awk, and perl, you will also need a good ASCII text editor. You can download Xemacs from http://www.xemacs.org.

1.2 Reading

Manual pages are available, among other places, at http://www.gnu.org/manual. If your computer is set up correctly, you can also read the bash man page by typing 'man bash' at the cygwin/bash prompt.

- Read the bash manual page, sections: (Basic Shell Features)→(Shell Syntax, Shell Commands, Shell Parameters, Shell Expansions). (Shell Builtins)→(Bourne Shell Builtins, Bash Conditional Expressions, Shell Arithmetic, Shell Scripts). Alternatively, you can try reading the tutorial chapter in the O'Reilly bash book.
- Read the sed manual page, or the sed tutorial chapter in the O'Reilly 'sed and awk' book.
- 'gawk' is another name for 'awk' on GNU-based systems — the 'gawk' program has a few more features than the original 'awk' program. You may eventually want to learn gawk or awk, but it's not required. The section of the gawk man page called 'Getting Started with awk' is pretty good. So are the tutorial chapters in the O'Reilly 'sed and awk' book.

1.3 A bash/sed example

Why not use C all the time? The answer is that some tasks are easier to perform with other programming languages:

- Manipulate file hierarchies: use bash and sed.
- (Simple manipulation of tabular text files: gawk)
- Manipulate text files: use perl.
- Manipulate non-text files: use C.

perl can do any of these things, but isn't very efficient for numerical calculations. C can also do any of the things listed, but perl has many builtin tools for string manipulation, so it's worthwhile to learn perl. gawk is easier than perl for simple manipulation of tabular text; it's up to you whether or not you want to try learning it.

bash is a POSIX-compliant command interpreter, meaning that, like the DOSshell, you can type in a program name, and the program will run. Unlike the DOSshell, bash is also a pretty good programming language (not as good as BASIC or perl, but better than DOSshell or tcsh).

For example, suppose you want to search through the entire /data/timit/train hierarchy[1], apply the C program "extract" to all WAV files in order to create MFC files, and create a file with extension TRP containing only the third column of each PHN file, and then move all of the resulting files to a directory hierarchy under /data/newfiles/train (but the new directory hierarchy doesn't exist yet). You could do all that by entering the following, either at the bash command prompt or in a shell script:

```
if [ ! -e ${HOME}/newfiles/train ]; then
  mkdir ${HOME}/newfiles;
  mkdir ${HOME}/newfiles/train;
fi
for dr in dr{1,2,3,4,5,6,7,8}; do
  if [ ! -e ${HOME}/newfiles/train/${dr} ]; then
    echo mkdir ${HOME}/newfiles/train/${dr};
    mkdir ${HOME}/newfiles/train/${dr};
  fi
  for spkr in `ls ${HOME}/timit/train/${dr}`; do
   if [ ! -e ${HOME}/newfiles/train/${dr}/${spkr} ]; then
    echo mkdir ${HOME}/newfiles/train/${dr}/${spkr};
    mkdir ${HOME}/newfiles/train/${dr}/${spkr};
   fi
   cd ${HOME}/timit/train/${dr}/${spkr};
   for file in `ls`; do
    case ${file} in
     *.wav | *.WAV )
       MFCfile=${HOME}/newfiles/train/${dr}/${spkr}/`echo ${file} | sed 's/wav/mfc/;s/WAV/MFC/'`;
       echo Copying file ${PWD}/${file} into file ${MFCfile};
       extract ${file} ${MFCfile};;
     *.phn | *.PHN )
       TRPfile=${HOME}/newfiles/train/${dr}/${spkr}/`echo ${file} | sed 's/phn/trp/;s/PHN/TRP/'`;
       echo Extracting third column of file ${PWD}/${file} into file ${TRPfile};
       gawk '{print $3}' ${file} > ${TRPfile};
     esac
   done
 done
done
```

[1]There are several copies of TIMIT floating around the lab. You can also buy your own copy for $100 from http://www.ldc.upenn.edu, or download individual files from that web site for free.

Once you have created the entire new hierarchy, you can list the whole hierarchy using

```
ls -R ${HOME}/newfiles | less
```

You may have noticed by now that bash suffers from cryptic syntax. bash inherits syntax from 'sh', a command interpreter written at AT&T in the days when every ASCII character had to be chiseled on stone tablets in triplicate; thus bash uses characters economically. Three rules will help you to use bash effectively:

1. Keep in mind that ', ‘, and " mean very different things. { and ${ mean very different things. [standing alone is a synonym for the command 'test'.

2. When trying to figure out how bash parses a line, you need to follow the seven steps of command expansion in the same order that bash follows them: brace expansion, tilde expansion, variable expansion, command substitution, arithmetic expansion, word splitting, and filename expansion, in that order. No, really, I'm serious. Trying to read bash as pseudo-English leads only to frustration.

3. When writing your own bash scripts, trial and error is usually the fastest method. Use the 'echo' command frequently, with appropriate variable expansions at each level, so you can see what bash thinks it is doing.

About awk or gawk: the only thing you absolutely need to know about gawk is that if you type the following command into the bash prompt, the file foo.txt will contain the M'th, N'th, and P'th columns from the file bar.txt (where M,N,and P should be any single digits):

```
awk '{printf("%s\t%s\t%s\n",$M,$N,$P)}' bar.txt > foo.txt
```

2 Perl

2.1 Reading

Read chapter 1 of *Programming Perl* by Larry Wall and Randall Schwartz [5]. This book is sometimes available on-line at perl.com; in the perl community, it is called "The Camel Book" in order to distinguish it from all of the other perl books available. Larry Wall (the author of perl, as well as the author of the book) is an occasional linguist with a good sense of humor. This chapter is possibly the best written introduction to perl data structures and control flow, and contains better documentation on blocks, loops, and control flow statements (if, unless, while, until) than the man pages.

The manual pages are available in HTML format on-line at http://www.perl.com/doc/. Download the gzipped tar file, and unpack it on your own PC, so that you can keep it open while you program. You can read the manual pages using "man" under cygwin, but it is much easier to navigate this complicated document set using HTML. Before programming, you should read Chapter 1 of the Camel Book, plus the `perldata` man page and the first halves of the `perlref` and `perlsub` manual pages. While programming, you should have an HTML browser open so that you can easily look for useful information in the three manual pages just listed, and also in the `perlsyn, perlop, perlfunc, perlref`, and `perlre` pages.

2.2 Perl Data Types and Variable References

This section will rapidly introduce perl data types, perl syntax, and the perl method for creating variable references. If you get confused, try the perlsyn and perlref man pages.

Perl uses three data types: scalars, arrays, and hash tables. Variables are never declared in perl; perl creates them as soon as they are used. Therefore the syntax must specify the type of a variable. Scalars always start with $, e.g., $foo is a scalar variable. Arrays always start with @, e.g., @foo is an array. Hash tables always start with %, e.g., %foo is a hash table.

A scalar variable always starts with $. Scalars may be numbers, strings, or references to other objects. For example, the code below prints the sentences "Interesting programs fail the 1st time" and then "Interesting programs fail the 2st time" to the special file STDOUT (the screen), with a newline at the end of each sentence:

```
$a=1;
$b="Interesting programs";
print STDOUT "$b fail the  ${a}st time\n";
$c=\$a;
$a=2;
$d=$$c;
$d=-1;
if ($a >= 2) {
  print STDOUT "$b fail the $${c}st time\n";
}
else {
  print STDOUT "test failed\n!";
}
```

Here are a few things to notice.

- Variables need not be declared in perl. In the code above, all variables are created as soon as they are used.
- `${a}` and `$a` mean the same thing. The brackets are used to specify the domain of the $ operator; without the brackets in line 3, perl would look for a variable called `$ast`.
- The line "`$c=\$a`" creates a *reference* for $a, and stores the reference in $c. From this point on, "$c" is a synonym for "a". Therefore $$c is a synonym for $a. A *reference* in perl is similar to a *pointer* in C, if you find that analogy useful. If you hate C, you may find it more useful to think of references as "synonyms" (an interpretation that is somewhat closer to the truth).

- Strings can be quoted using either double quotes (") or single quotes (’). Variables in a double-quoted string are evaluated; variables in a single-quoted string are not. Thus the line

  ```
  print STDOUT "\$b fail the $${c}st time\n"
  ```

  prints “Interesting programs fail the 2st time”. If the last line read

  ```
  print STDOUT ’\$b fail the $${c}st time\n’;
  ```

  then the program would print out exactly the characters “$b fail the `$${c}st time\n`”.

- The if-else syntax works just like it should. Perl only does two unique things with if-else. First, the body must be enclosed in brackets (`{` and `}`), so it’s always clear where is the beginning and ending of the if-else. Second, since every scalar can be evaluated as either a string or a number, numeric comparisons and string comparisons are completely different functions. Numeric expressions look like ($a == $b), ($a >= $b), ($a < $b), and so on; string expressions look like ($a eq $b), ($a ne $b), and so on.

An array variable always starts with @. An array is just a list of scalars, stored and indexed in numerical order, starting with element number 0. The elements of an array are scalars, thus the elements of array @a are $a[0] through $a[$#a]. The notation “$#a” refers to the index of the last element stored in array @a. The notation “1..5” creates a list of all digits between 1 and 5; similarly “’a’..’z”’ creates a list of all letters between ’a’ and ’z’. A useful syntax using arrays is the “foreach” syntax, used like this:

```
@array1=(0..2);
foreach $i (0..2) {
  $array1[$i] = $i * $a[$i];
}
foreach $entry (@array1) {
  print STDOUT "$entry\n";
}
$aref=\@array1;
${$aref}[2]=3;
print STDOUT "@array1\n";
print STDOUT "@{$aref}\n";
```

The code above prints the numbers 0,1, and 4 on lines by themselves. Then it prints “0 1 3” on a line by itself, twice in a row. The line `$aref=\@array1` creates a reference to @array1, and places the reference into $aref; from that point on, $aref is a synonym for array1, @$aref is a synonym for @array1, and $$aref[0] is a synonym for $array1[0]. In line 9, the brackets (`{` and `}`) are used to specify the order of evaluation: $aref should be evaluated first to find the reference to array1, and only then should its third element be extracted. In fact, the brackets in lines 9 and 11 are not necessary, because, without brackets, perl would evaluate these arguments in exactly the order specified here. However, it never hurts to put in extra brackets, if you’re not sure of the order in which perl will evaluate your statement.

The third data type, the hash table, is the most important data structure in perl. A hash table always starts with %. A hash table is like an array, but the index (called the “key”) can be any scalar — a digit, a string, or even a reference. If the string ”something” is one of the keys in hash table %foo, then the corresponding value is accessed using $foo`{"something"}`. Hash tables are initiated using even-length lists, containing key-value pairs. “keys %foo” returns a list of the keys; “vals %foo” returns a list of the values. Thus, the following code prints “1 apple apple” on the first line and “65 bob 65 on the second line. The order “1 apple”, and the order “65 bob”, are specified by the “sort” function. Entries in a hash table may be stored in any order, so without sorting the output, you never know in what order they will be.

```
%foo = (1, "bob");
$foo{"apple"}=65;
@keylist = sort keys %foo;
```

```
print STDOUT "@keylist apple\n";
@vallist = sort vals %foo;
print STDOUT "@vallist $foo{'apple'}\n"
```

Doubly-indexed arrays and hash tables are extremely useful, and their syntax is cryptic. It is possible to create doubly-indexed tables on the fly, just like any other perl variable. For example, the following code prints out the lines "0 0 0, 0 0 0", "0 1 2, 0 1 2", and "0 2 4, 0 2 4":

```
foreach $i (0..2) {
  foreach $j (0..2) {
    ${$array2[$i]}[$j] = $i * $j;
    ${$hash2{$i}}{$j} = $i * $j;
  }
}
foreach $i (keys %{$hash2}) {
  @foo = vals %{$hash2{$i}};
  print STDOUT "@foo, @{$array2[$i]}\n";
}
```

The brackets in lines 3 and 4 are necessary!! If you write `$$array2[$i][$j]`, perl will try to dereference "$array2" as a scalar first, rather than doing what you wanted (dereferencing the array element `$array2[$i]`). However, the notation `${$array2[$i]}[$j]` is really ugly and cumbersome, even by perl standards. Therefore perl provides the following mnemonically useful synonyms:

```
$array2[$i][$j]  is a synonym for  ${$array2[$i]}[$j]
$hash2{$i}{$j} is a synonym for  ${$hash2{$i}}{$j}
```

Therefore the code above could be written as

```
foreach $i (0..2) {
  foreach $j (0..2) {
    ${$array2[$i]}[$j] = $i * $j;
    ${$hash2{$i}}{$j} = $i * $j;
  }
}
foreach $i (keys %{$hash2}) {
  @foo = vals %{$hash2{$i}};
  print STDOUT "@foo, @{$array2[$i]}\n";
}
```

2.3 Perl I/O

When the perl program "foo.pl" is called using the command line "foo.pl A B C," the arguments are stored into a special array called @ARGV. Thus the following program prints out its arguments, one on each output line:

```
#!/usr/bin/perl
# This is a comment.
# This is also a comment.
# This comment is here to tell you that
#   the line below will print, to standard output,
#   one command line argument per line
#
foreach $i (0..$#ARGV) {
  print STDOUT "$ARGV[$i]\n";
}
```

The line `#!/usr/bin/perl` is a special unix syntax. On a unix system, if the file "foo.pl" starts with this line, then one may type "foo.pl A B C" in order to run the program. Specifically, unix will open the file "foo.pl," and look on its first line to see what interpreter should be used to run it. Upon discovering that /usr/bin/perl is the correct interpreter, unix will fire up /unix/bin/perl, and feed it the rest of the commands in the file.

All other lines starting with # are treated as comments, and ignored by perl. Always comment your code.

Files may be opened using the "open" function. Opened files are accessed via filehandle variables. Filehandle variables are just alphanumeric strings in perl. The syntax "¡FILE1¿" reads in one line of text from FILE1. For example, suppose that the code below is in a program called "foo.pl." Then the unix command line "foo.pl file1.txt file2.txt" will print the contents of file1.txt to STDOUT, and will then wait for the user to enter text into the text prompt. Any text that the user enters, up to the first control-D character, will be written out to file2.txt:

```
#!/usr/bin/perl
# Program foo.pl
# Author: Mark Hasegawa-Johnson, 5/18/2005
#
# Function: Read from $ARGV[0] to STDOUT, write STDIN to $ARGV[1]
#
# Usage: foo.pl file1.txt file2.txt
#
# Open file1.txt. If the open fails, stop the program, and print out an
# error message.  The variable $! is a perl special variable,
# containing the system error message returned when perl failed to open file1.txt
open(FILE1,$ARGV[0]) || die "Unable to read from $ARGV[0]: $!";

# Open file2.txt for writing.
# The syntax ">$ARGV[1]" is a special syntax meaning the file
# should be opened for writing, not for reading
open(FILE2,">$ARGV[1]") || die "Unable to write to $ARGV[1]: $!";

# Read all entries from file1.txt, and write them to stdout
# The ‘‘while’’ command causes its following loop to continue
# until the end of the file is reached
while( $input = <FILE1> ) {
   print STDOUT "$input";
}
close(FILE1);

# Read from STDIN until the user hits control-D; write these to FILE2
print STDOUT "Type as much as you like; end by hitting CTRL-D\n";
while( $input = <STDIN> ) {
   print FILE2 "$input\n";
}
close(FILE2);
```

2.4 Regular Expressions

Regular expressions are the main reason for using perl. Other languages have regular expressions that are almost as convenient, but not quite. For more information on ideas presented in this section, see the `perlre` man page.

A *regular expression* is a *regular grammar* [3], plus some other stuff. Let's talk about *regular grammars* first. A *grammar* is a set of rules, or any other specification, that distinguishes between "legal" and "illegal" strings. A *parser* is an algorithm or computer program that determines whether or not a given string

satisfies a given grammar. A *regular grammar* (also called a *finite state grammar*) is any grammar that can be parsed by a finite state machine. Equivalently, a regular grammar is any grammar that can be specified by appropriate combinations of the following three rules.

1. A **concatenation** rule specifies that subgrammar `C` is composed of the concatenation of any string from subgrammar `A`, followed by any string from subgrammar `B`. In regular expression syntax, concatenation is expressed by sequence: `C=AB`.

2. A **disjunction** rule specifies that subgrammar `C` is composed of either any string from subgrammar `A`, OR any string from subgrammar `B`. In regular expression syntax, concatenation is expressed by `|`: `C=(A|B)`.

3. A **Kleene closure** rule specifies that subgrammar `C` is composed of an arbitrary non-negative number of consecutive strings, each of which is selected from subgrammar `A`. In regular expression syntax, Kleene closure is represented using `*`: `C=(A)*`. The parentheses are optional if `A` contains only one character.

For example, consider the following regular expression:

```
G1:      (sc|g)o*(al|re)
```

Grammar G1 matches the following strings: "score", "goal", "goooooal", "scooooooore". It also matches a few strings that were probably not intended by the designer: "scoal", "gore", "scre", "gal", and "scal", for example.

Perl provides the following extra matching operators. All of these can be broken down into various combinations of concatenation, disjunction, and closure, but it is often useful to have special notation for these frequent events:

- `.` — Any Character. Match any single character. Thus the grammar `Mar.` matches the strings "Mark", "Marc", "Mary", and "Mart". The grammar `illi.*` matches "illinois", "illini", "illiac", and "illi45 18[2b}3".

- `+` — Positive-definite Kleene closure. Concatenate one or more sub-grammars (remember that `*` requires zero or more). For example, `(sc|g)o+(al|re)` is almost the same as grammar G1, except that it doesn't match the strings "gal", "scre", "scal", or "gre".

- `?` — Optional Character. Match zero or one instances of the preceding character. `s?he` matches the strings "he" and "she".

- `[]` — Sub-vocabulary. Match exactly one character from the list of characters specified within the brackets. For example, the grammar `[cdb](a|o|ea)[tgr]` matches "cat", "dog", "bear", "cear", "dear", "car", et cetera.

- `[a-z]` — Range. Match exactly one character from the characters whose ASCII value is between 'a' and 'z'. The grammar `[a-z]+` matches any string of lowercase letters.

- `[^A]` — Negated subvocabulary. Match any character *except* those specified in list A.

- `\` — Quote. Interpret the following character literally, instead of using its special meaning. For example, the grammar `\[[a-z]+\]` matches the strings "[a]", "[aoidb]", and "[bigbadwolf]".

- Special Matching Characters:
  - `\d` matches any digit
  - `\s` matches any whitespace (space, tab, newline) — thus `\s+` matches any sequence of whitespace characters.
  - `\S` matches any non-whitespace
  - `\w` matches any characters that can be part of a word — alphanumeric or `_`, but no other punctuation. Thus `\w+` matches any single word.

– `\n` matches a newline

Regular expressions in perl are not strictly regular. The following characters are *context-dependent:* they force the rest of the grammar to match only if surrounding context is correct.

- `^` — match the beginning of the line. For example, `^\s+#` matches any # symbol and the whitespace that precedes it, but ONLY if the # is the first non-whitespace character on the line.
- $ — match the end of the line. `;\s+$` matches a semicolon and its following whitespace, but only if the semicolon is the last non-whitespace on the line.
- `\b` — match a word boundary. The regular expression `s?he` matches the strings "she" and "he" regardless of their context — thus, for example, the string "he" in the word "withheld" is matched. In order to require the match to be a complete word, you can specify the grammar `\bs?he\b`.

Regular expressions are used, in perl, using one of the following four syntactic constructions:

1. Matching. Syntax:

   ```
   VARIABLE =~ /REGEX/
   ```

   For example,

   ```
   #!/usr/bin/perl
   print STDOUT "Do you want to proceed?";
   # Read in the user's response
   $response = <STDIN>;
   # If response was yes, Yes, yeah, yippeee, etc., then...
   if ($response =~ /^[yY].*/) {
     print STDOUT "Good!  So do I\n";
   }
   ```

   In this example, notice the "line begin" marker `^` at the beginning of the regular expression. A regular expression match is considered successful if it matches ANY PART of the variable — thus the expression `[yY].*` would match the final "y" in "no way". The character `^` specifies that the match should fail unless the detected `[yY]` character is the first character in the line entered by the user.

2. Substitution. Syntax:

   ```
   VARIABLE =~ s/REGEX/REPLACEMENT/[MODIFIERS]
   ```

   The optional modifiers may include "g" (perform the substitution as many times as possible, instead of performing it only once), or "i" (case insensitive matching). The following example reads in words from the user, and prints back the same words, with all digits replaced by the character *:

   ```
   #!/usr/bin/perl
   # Read in user responses until user enters control-D
   while($response = <STDIN>) {
     # Replace any digits with the character *
     $response =~ s/\d/\*/g;
     # Print the response
     print STDOUT $response;
   }
   print STDOUT "Last response was $response";
   ```

3. Translation. Syntax:

```
VARIABLE =~ tr/FROM_VOCAB/TO_VOCAB/
```

Translation is special, in that the TO_VOCAB must contain exactly as many characters as the FROM_VOCAB. Any character in the FROM_VOCAB is replaced by the corresponding character in the TO_VOCAB. This syntax is most often used to uppercase or lowercase a string, thus

```
#!/usr/bin/perl
# Read in user responses until user enters control-D
while($response = <STDIN>) {
  # Uppercase any lowercase letters
  $response =~ tr/[a-z]/[A-Z]/;
  # Print the uppercased response
  print STDOUT $response;
}
print STDOUT "Last response was $response";
```

4. String Extraction. String extraction is actually a matching feature, not a syntax. It works like this: any time perl encounters the "(" character in a regular expression, it extracts the string matched by the regular expression between "(" and its matching ")", and inserts that substring into one of the special variables $1 through $9. Thus, for example, to extract the components of a fully specified pathname, one could write

```
#!/usr/bin/perl
# Read in user responses until user enters control-D
while($pathname = <STDIN>) {
  # Check to see if this is a fully specified path
  if($response =~ /(.*)\/((.*)\.(.*))/) {
    # If so, extract the path, filename, root, and extension
    $path = $1;
    $filename = $2;
    $rootfilename = $3;
    $extension = $4;
    # Print them to standard output
    print STDOUT "Path is $path, Filename $filename is composed of\n";
    print STDOUT "  root name $rootfilename and extension $extension\n";
 }
 else {
  # If not, print an error message
  print "Unable to parse pathname\n";
 }
}
```

3 N-Gram Language Modeling

A probabilistic grammar of language L may be considered useful if it satisfies one of the following two objectives:

1. Specifies the probability of observing any particular string of words, $W = [w_1, \ldots, w_M]$ in language L.
2. Specifies the various ways in which the meanings of words $[w_1, \ldots, w_M]$ may be combined in order to compute a sentence meaning, and specifies the probability that any one of the acceptable sentence meanings is what the talker was actually trying to say.

Shannon [4] proposed a simple language model satisfying criterion number 1, but not criterion number 2. His language model has since been called the "N-gram model." The model proposes that the probability of any sentence is:

$$p(W) = \prod_{m=1}^{M} p(w_m | w_{m-N+1}, \ldots, w_{m-1}) \tag{1}$$

where the words $w_{-N}, \ldots, w_{-1}$ are defined to be the special symbol "SENTENCE_START." If the length of the N-gram, N, is larger than the length of the sentence, M, a correct N-gram specifies the probability of the sentence exactly. In practice, most N-grams are either bigrams ($N = 2$) or trigrams ($N = 3$), although a few sites have experimented with variable-length N-grams.

The maximum likelihood estimate of the bigram, unigram, and "zero-gram" probabilities are

$$p_{ML}(w|v) = \frac{C(v,w)}{\sum_w C(v,w)} \tag{2}$$

$$p_{ML}(w) = \frac{C(w)}{\sum_w C(w)} \tag{3}$$

$$p_{ML}(\cdot) = \frac{1}{N(\cdot)} \tag{4}$$

where the "token count" $C(v,w)$ is the number of times that the word sequence (v,w) was observed in a training corpus, where v and w are any two words in the dictionary. The "type count" $N(\cdot)$ is the number of distinct words (so $p_{ML}(\cdot)$ is the hypothesis that all words are equally likely). Training corpora vary in size: real conversational speech corpora may be as large as 1-10 million word tokens (Switchboard is 1 million, Fisher corpus is 10 million); in order to get larger corpora, it is necessary to use written texts, e.g. the Gigaword corpus contains 1 billion word tokens.

Because of the infinite productivity of human language, there are always many perfectly reasonable word sequences that are not observed in any finite-sized training corpus. If we assign zero probability to unseen bigrams, then the recognizer is guaranteed to fail to recognize them if they ever occur in test data. It is usually better to "smooth" the N-gram using deleted interpolation or backoff. Deleted interpolation [2] is just a linear interpolation of probabilities:

$$p_I(w|v) = \lambda p_{ML}(w|v) + (1-\lambda) p_{ML}(w) \tag{5}$$

where $0 \leq \lambda \leq 1$; multi-level interpolation can be used to combine maximum likelihood probabilities from several levels, e.g., 4-gram, 3-gram, 2-gram, 1-gram, and 0-gram (uniform). "Backoff" is like interpolation, but the coefficient λ depends on v, and the backoff probability $p(w)$ is not necessarily a maximum likelihood estimate:

$$p_B(w|v) = \lambda(v) p_{ML}(w|v) + (1-\lambda(v)) p_1(w) \tag{6}$$

Many different schemes can be used to calculate $\lambda(v)$ and $p_1(w)$. The most common is the "add-one" probability estimate, also called Good-Turing backoff. Good-Turing backoff is computed by adding $D \approx 1$ to the count of every event in the database:

$$p_{GT}(w|v) = \frac{C(v,w) + D}{\sum_w (C(v,w) + D)} \tag{7}$$

(exercise for the reader: show that Eq. 7 can be written in the form of Eq. 6). Finally, Kneser and Ney propose the following backoff:

$$p_{KN}(w|v) = \frac{\max(0, C(v,w) - D) + D\alpha(v,w)}{\sum_w C(v,w)} \tag{8}$$

where the backoff factor is

$$\alpha(v,w) = \frac{N(v,\cdot)N(\cdot,w)}{N(\cdot,\cdot)} \tag{9}$$

$N(v,\cdot)$ is the number of distinct bigrams observed in the training corpus that start with v, and $N(\cdot,w)$ is the number of distinct bigrams observed to end with w. Eq. 9 therefore has the interesting property that

$$\sum_w (\alpha(v,w) - 1) = \sum_v (\alpha(v,w) - 1) = 0 \tag{10}$$

The zero-sum property in Eq. 10 has two useful consequences. First, $p_{KN}(w|v)$ is a probability. Second, the marginal counts are correct. In equation form, these properties are written as

$$\sum_w p_{KN}(w|v) = 1 \tag{11}$$

$$\sum_v p_{KN}(w|v)C(v) = C(w) \tag{12}$$

Chen and Goodman [1] showed that Eq. 8 gives better N-gram probability estimates, in general, than Eqs. 5 or 7. They also demonstrated a method for extending Eq. 8 to allow multiple backoff, e.g., from a 4-gram to a 3-gram to a 2-gram to a unigram to a uniform distribution. They demonstrated that the optimal backoff factor D should be somewhere between $D = 0.6$ (ideal for a bigram that occurs only once in the training data) and $D = 1.4$ (ideal for a bigram that occurs three or more times in the training data).

The goal of a language model is to estimate the frequency of any particular word sequence in the *test data*. The ML model is exactly equal to the bigram frequency in the *training data*; the only reason that we use backoff is because the test and training data may be different. A useful measure of language model quality is the cross-entropy between the trained backoff model $p_B(w|v)$ and the word frequencies $C_T(v,w)$ in the test database:

$$H(T;B) = -\frac{1}{N_T} \sum_v \sum_w C_T(v,w) \log_2 p_B(w|v) \tag{13}$$

It can be shown that, by this definition, $H(T;B) \geq H(T;T)$, i.e., the cross-entropy is always larger than the entropy that would be obtained by using the test corpus frequencies $p_T(w|v) = C(v,w)/C(v)$. Therefore, the better the language model is, the smaller $H(T;B)$ becomes.

Eq. 13 includes summation, over word *types*, of the test-corpus counts, which is equivalent to summing over word *tokens*, thus

$$H(T;B) = -\frac{1}{N_T} \sum_{n=1}^{N_T} \log_2 p_B(w_n|w_{n-1}) \tag{14}$$

The "perplexity" of the language model can be roughly understood as the average number of words from which the recognizer must choose in order to recognize each new word in the test database. It is defined as

$$P(T;B) = 2^{H(T;B)} \tag{15}$$

4 Homework

Download the Switchboard transcriptions from http://www.isip.msstate.edu/projects/switchboard/.

Use perl or python to accumulate sufficient statistics from the first 2/3 of the Switchboard corpus (dialogues 2005 through fo 3999) for the estimation of a Kneser-Ney-smoothed bigram language model (Eq. 8). Then use the remaining dialogues (dialogues 4000 through 4940) to estimate the perplexity of your language model.

Your code should have three distinct sections: (1) First, find the bigram counts $C(v, w)$ for every possible word-pair in the training data, and then (2) manipulate the bigram counts in order to calculate $p_{KN}(w|v)$, (3) read in the test data, one transcription at a time, and estimate the perplexity of the model.

The Switchboard transcriptions include annotation of silences, background noises, word fragments, and other non-speech events. Use perl's *regular expression* capability to map all such events to the most reasonable language-modeling category. For example, a word fragment might be mapped to the intended word; or it might be mapped to the intended word followed by the special word token "DISFLUENCY," i.e. `bec[ause]-` could become either "because" or "because DISFLUENCY." For the purpose of language modeling, background noises such as `[laughter]` and `[silence]` and `[mouthnoise]` can all be eliminated from the transcription.

Notice that there are more than 30,000 distinct words in Switchboard. If you try to represent $C(v, w)$ or $p(w|v)$ as a fully enumerated table, you will wind up with a table of size 900M. Don't do that. Instead, use a perl *hash table* that includes entries for only the bigram pairs that are actually observed in the database (about 2M entries).

In order to read in all of the transcription files, you may do one of two things: (1) use perl's `dir` function to traverse through the various subdirectories, or (2) use the unix `ls` command to list all of the desired transcription files into a master script file (e.g., type `ls {2,3}*/*/*-trans.text > training.scp`), and then write the perl program so that it reads, one at a time, the files specified by training.scp.

References

[1] Stanley F. Chen and Joshua Goodman. An empirical study of smoothing techniques for language modeling. *Computer Speech and Language*, 13:359–394, 1999.

[2] Kai-Fu Lee, Hsiao-Wuen Hon, and Raj Reddy. An overview of the sphinx speech recognition system. *Trans. ASSP*, 38(1):35–45, January 1990.

[3] S. E. Levinson. Structural methods in automatic speech recognition. *Proc. IEEE*, 73:1625–1650, 1985.

[4] Claude Shannon. *The Mathematical Theory of Information*. University of Illinois Press, 1955.

[5] Larry Wall and Randall Schwartz. *Programming perl*. O'Reilly and Associates, Inc., Sebastopol, CA, 1991.