

# Assignment #6: Markov-Chain Language Learner

CSCI E-220 Artificial Intelligence

Due: Thursday, November 5, 2009

## 1. General Idea

**You will write a program that determines (as best as possible) the language in which a test sentence has been written, using a simple one-place lookahead Markov-chain algorithm. You can write it in LISP, but if you prefer you may choose another programming language.**

Your program will begin by reading a series of input files in known languages, and constructing probability matrices for each. It will keep track of these arrays in a LANGUAGES list (which will be a list of probability-value arrays). Then it will read test sentences from a file and report the language that each sentence is most likely in, along with a probability estimate of correctness.

Your program must have two top-level function calls (or the equivalent in your chosen language):

```
(learn "language")
```

which will attempt to open a language file and learn it as specified in section 2, and

```
(test "testfile")
```

which will open a test file and read it one sentence at a time, reporting on each sentence.

Although perhaps other languages are better suited to this program, LISP will do perfectly well. You'll need to know about functions that open input streams, that read characters from the streams, that determine ASCII codes for the characters, and that make arrays, but none of those is particularly tricky once you know them, and this document will provide details.

You will also need to understand how the probability values for single-place lookahead Markov-chains are compared with each other. The topic will be discussed in class, but this document will also provide some guidance in that area.

Sections 2 and 3 below will refer to a number of LISP functions you may not be familiar with; section 4 will have further details about how to use them. Also read Graham section 4.1 (on Arrays) and see page 354 for the various character-manipulation functions. You will not need to deal with string types in this program since all the input can be dealt with character-by-character.

## **2. Initializing the Program - Learning Languages**

Six test corpuses are provided for you, one each in English, French, Spanish, Italian, German, and Portuguese. These files are in the `~lib220/asst6` directory, with the name of each file the same as its language. Copy them all to your own directory. Then:

### **2.1. Read each file one character at a time, and construct a frequency matrix (array) for that language. Use the make-array function (see Graham p. 58).**

Details: Treat upper and lower case letters as identical, and treat any non-letter as a space. Multiple spaces should be treated as a single space. Each of your arrays will need to be 27 X 27. Use the "0" row and column for end-of-word markers (spaces or unknown characters) and 1 through 26 for A through Z. Do not worry about accented characters -- they will all have been translated to their 7-bit ASCII equivalents in the corpuses you receive (e.g. é and è will have been turned into e, and ñ to n, etc.). If I've made a mistake and the samples contain accented characters you will treat them like any other non-letter. Use the `char-code` function to find a letter's decimal ASCII value. In case you're not familiar with ASCII codes, the uppercase letters A-Z are 65-90 respectively, and lowercase letters a-z are 97-122 respectively. The Space character value is 32, but that's not important because you will treat any non-letter as an end-of-word marker.

### **2.2 From each frequency matrix, construct a probability matrix. Once you've done this, you no longer need the original frequency matrices.** The only thing you lose by doing this is a measure of the quantity of data in your original corpus (which is, perhaps, a measure of the reliability of your guesses later on). You gain a good deal of efficiency in the testing phase. **You will store these probability matrices in a list.**

Details: A **frequency** matrix tells you how many occurrences you found of letter A followed by letter B. What you really want to know is the **probability** that letter A will be followed by letter B. This can be computed across each row as simply the frequency in any particular cell divided by the sum of all the entries in that row. Be sure to compute the probabilities by row, not by column! That is, you want to know the probability, given that a particular letter X has just occurred, that each of the other letters will occur next (and not the other way around).

**DO NOT USE ANY PROBABILITIES OF ZERO!** Any value of zero in the frequency matrix should be treated as a small positive number less than 1 (**please use 0.1 for this assignment**) when computing the probability matrix, so that very rare letter-frequency pairs, or the occasional typographical error, will not immediately disqualify a language. Multiplying by zero is a strong final judgement, and a single typo or an unusual proper noun in a test input string could disqualify every language simultaneously if values of zero were allowed. On the other hand, you do not want to treat unusual pairs as if they were as common as a pair that did occur in your input even once.

### **3. Testing a Sentence**

Once you have learned the six input languages, you are ready to test a new sentence to see what language it is in! Your top-level function for this section will loop as follows:

#### **3.1 Read a sentence from a testfile and compute a probability from each array as you read.**

Details: A test file is provided in `~lib220/asst6/test-data`. After opening the file, read it character by character until you reach a period (you can do a test for equality versus `#\.` or versus the ASCII character code 46). You'll need to keep track of a list of probability values, one for each language in your LANGUAGES list. (Note: the probabilities should be initialized to 1.00, as they can only go down as you read. You'll want the `make-list` function for this task; see below.)

#### **3.2 Compare the probabilities and return the best one.**

Details: The most important piece of output you will be returning is the name of the language array corresponding to the highest probability in your list. You will also need to return an estimate of the probability that the sentence is in fact in that language as opposed to one of the others. All of the probabilities in your current list will be vanishingly small, so those numbers are not interesting: what is interesting is how they compare to each other. For this assignment, you will normalize the value you return by assuming that the sentence must be in one of the languages you have learned so far. Thus, the probability of a particular language is the ratio of the value for that language to the sum of all the values in your list, and the highest such ratio corresponds to the most likely language.

For example, if your probability list is (0.001 0.002 0.003 0.0005 0.0005 0.002), then the third value (0.003) is the highest value, and the probability you will give as output is 0.003 divided by 0.009 (the sum of all the numbers), which is 0.333. There is a one-third chance that the sentence is in the language represented by the third array in your LANGUAGES list (whatever that is).

#### **3.3 Finally, you will loop and process another sentence, until you reach the end of the file.**

### **4. LISP functions you may need**

**Array functions** (note: you will only need two-dimensional matrices for this program):

`(setq *print-array* nil)` - this call, typed exactly as shown, is used to prevent arrays from printing their entire contents to the screen every time you create one. You may not notice the difference while your program is running but you most certainly will if you try to create arrays directly on the command line. This is the sort of function call you can put at the beginning of your program, or can type directly on the command line before testing individual functions.

`(make-array '(rows cols))` - creates an array with `rows` rows and `cols` cols.

Usage example: `(setq ENGLISH-FREQ (make-array '(27 27)))`

`(setf (aref array row col) value)` - sets the row/col element of array to value

Usage example: `(setf (aref ENGLISH-FREQ 0 0) 22)`

`(aref arrayname row col)` - returns the value of the row/col element of arrayname

Usage example: `(aref ENGLISH-FREQ 0 0)` would return 22 after the previous example.

### List functions

`(make-list number :initial-element value)` - creates a new list with "number" elements in it, each one equal to "value".

Usage example: `(make-list (length LANGUAGES) :initial-element 1.0)`

### Opening an input file stream, and reading characters from it:

`(open "filename" :direction :input)` - opens the file named "filename" and returns an input stream. The keywords `:direction` and `:input` are typed exactly as shown.

Usage example: `(setq inputstream (open "filename" :direction :input))`

Characters in LISP are printed with leading `#\` symbols, e.g. `#\B` is the uppercase-B character. Some special characters are printed with names (`#\Space` and `#\Newline`, for instance). You can type those forms directly into a function if you wish to test for equality. Frequently it is easier to use the ASCII character code equivalents, since integers are easier to manipulate.

`(read-char stream nil eof-value)` - reads a single character from a named input stream. The "nil" indicates that end-of-file should be tested for, rather than be allowed to generate an error condition, and the eof-value is returned at end of file.

Usage example: `(read-char inputstream nil nil)` - to read from the input stream defined above, and return nil at end-of-file.

`(char-code char)` - returns the ASCII character code for a character. To turn lowercase letters into uppercase or vice versa, use ASCII codes and note that the difference between an uppercase letter code and the same letter in lowercase is always 32.

## 5. What to Hand In

Hand in a commented listing of your program, and a runtime script showing your program learning each of the six languages (no output necessary during learning) and then testing each sentence in the test file. For each test sentence, you should output the "most likely" language and the probability associated with it. (Optionally you may suggest other possible languages for the sentence, but if you do so you must list them in descending order of probability).

## 6. Optional extension:

(10%) Write a two-place lookahead algorithm with 3-dimensional arrays (or with a two-dimensional array with  $27^2=729$  rows and 27 columns). Conceptually this program is not more complex than the main program, although there is more to keep track of. You'll still be calculating output probability matrices based on single 27-place rows; but there will be 729 of them.

However, to make this extension interesting, you will need to do a couple of things:

- **Create much larger learning corpuses** for each of the six languages. The small corpuses I gave you don't contain nearly enough text to differentiate nicely between 729 possible preconditions for each subsequent letter. You can use Internet searches to find newspaper or other articles written in each language. Is 50 kilobytes of information per language sufficient? Or 100kb? Start there, and find out if you gain any accuracy or confidence as compared to a one-letter lookahead algorithm (see below). If not, try even more data. Be sure to convert non-English characters to the basic 26-letter alphabet. I have a little PERL routine that you may borrow (ask me!) to help you with the conversion task if you aren't sure how to do it yourself.

- **Test some sentences against both a one-letter and two-letter lookahead algorithm** using the same learning corpus, to see whether there is actually any gain in confidence -- or accuracy! -- using a two-letter lookahead algorithm. Can you find sentences that are NOT properly identified using a one-letter lookahead algorithm, but ARE properly identified with a two-letter lookahead algorithm? If so, provide examples and some thoughtful explanation. If not, why not? You may also want to modify your program to accept input from the command line rather than from a file, for ease of testing.