Homework: What Language is this Web Page?

COMP 50

Fall 2013

Contents

Deadline

This homework is due at 11:59PM on **Tuesday, November 26**. Submit your solutions in a single file using the COMP 50 Handin button on DrRacket; the homework is the `trigrams` homework.

All the exercises should be done using the Intermediate Student Language with lambda.

Introduction

This homework is a little different from your others. Instead of focusing on a particular skill or aspect of the design recipe, it asks you to apply *all* your skills to a problem: identifying what natural language a given text is written in. You will be learning a couple of new techniques, including a lot of math and a little input and output.

There is only one problem: *design a function that is given a URL and figures out what language it is written in.* I actually want three versions:

- The first version returns a list that lists each possible language along with the logarithm of a probability. Higher probabilities are better, and the list should be sorted by probability with the highest probability at the front.
- The second version returns a list that lists each possible language along with the cosine of an angle. Higher cosines are better, and the list should be sorted by cosine with the highest cosine at the front.
- The third version either picks the winning language or says "I don't know" by returning `false`. To write this version you will have to decide which method works better—the probability method or the cosine method—so you'll need to do some experiments, which I would like you to describe in the comments.

Domain knowledge: Document classification

The general problem of deciding "what kind of thing is this document?" is called *classification*. The underlying principles are the same whether you are trying to tell French from English or whether you are trying to decide if a newspaper article is about business or sports. The story looks something like this:

- Create some kind of *model* that describes a large population of documents. For example, the model might say that words like "revenue" are more likely associated with business whereas words like "ball" are more likely associated with sports.

  A model has numerical *parameters*, which *quantify* the judgments made by the model. The parameters don't just say "more likely;" they say *how much* more likely.

- The parameters of the model are usually determined by examining *training data*. If we have a large sample of newspaper articles, we can look at the sports pages and at the business pages and see how often the word "ball" occurs.

  If you have ever filled in a Recaptcha, you are providing Google with free expert labor, helping them train their model of how images of old books related to words and letters.

- Training data are divided into *corpora*. Each *corpus* consists of a set of documents that go together. You will have an English corpus, a French corpus, a Polish corpus, and so on.[1]

- Once you build a model of each corpus, the next step is usually to *validate* the models. This involves some kind of quantitative analysis to see if the model is any good. Sadly, we won't have time for this step.

- Finally, the models are ready to be used to *classify*. An unknown text is presented each model, and you ask the model if the text is a good match. The best match wins. Depending on the techniques that are used, your models might or might not be able to tell you how confident you should be in the answer.

[1] *Corpus* is the Latin word for "body;" *corpora* is the plural form.

Trigram models

We'll focus on models that look at *trigrams*. A trigram is a three-character string, or equivalently, a sequence of three one-character strings.[2] The sentence `"See Spot run."` has these trigrams:

```
(list "See" "ee " "e S" " Sp" "Spo" "pot" "ot " "t r" " ru" "run" "un.")
```

In larger documents, trigrams are duplicated. We can build a model of a large text by keeping track of each trigram and how often it occurs. For example, if the text is `"The cat in the hat"`, we can represent a trigram model using an association list:

```
'((" ca" 1) (" ha" 1) (" in" 1) (" th" 1) ("The" 1) ("at " 1)
  ("at." 1) ("cat" 1) ("e c" 1) ("e h" 1) ("hat" 1) ("he " 2)
  ("in " 1) ("n t" 1) ("t i" 1) ("the" 1))
```

Larger texts result in larger models. Here are the 10 most common trigrams in Mr Lincoln's Gettysburg Address:

```
((" th" 34) ("the" 20) ("at " 18) ("hat" 15) ("ed " 13)
 ("her" 13) ("tha" 13) (" de" 12) ("he " 11) ("re " 11))
```

It turns out that some languages are more likely to use certain trigrams than others. Here, for example, are trigrams that are found in English texts but are much more likely to be French:

```
("un " "une" "tou" "ais" "uel" "eur")
```

By contrast, here are some trigrams that are found in French texts but are much more likely to be English:

```
("the" "ing" "you" " th" "one")
```

Recommended structure of a model

I recommend that you define a model by parts, and that when you build a model, you keep track of the following information:

- The name of the corpus, e.g., `"English"`
- A mapping that takes each possible trigram into the number of times that trigram was observed in the corpus (a binary-search tree will do well enough)
- The total number of observations used to create the model
- The number of *distinct* trigrams observed, i.e., the number of nodes in the binary tree

Don't ever store a trigram with a zero count—instead, assume that if a trigram is not found in the binary-search tree, it was never observed.

[2] DrRacket uses something called Unicode characters. For reasons that are not obvious to the amateur, we will instead use "bytes" as characters.

Structure of a document classifier

With this background, you can probably break down the problem a little further:

1. Get your hands on a several corpora from known languages.
2. Build a trigram model of each corpus.
3. Get a sequence of bytes from a URL—that's your *test document*.
4. Examine the trigrams in the test document and decide which model best fits the test document.

There are a couple more steps that are easy to overlook:

5. Not all URLs point to text documents. You will need a model for "none of the above."
6. Building a model from training data is expensive. To avoid repeating that expense each time you want to classify a URL, I will ask you to save your models to disk. In general, being able to save data on disk is a valuable skill.

   To save and restore, you will *serialize* and *unserialize* your models:

   - To serialize a value is to convert it to a sequential form for writing to disk.
   - To unserialize the data on disk is to convert it back from its sequential form to its original form.

   Serialization is most commonly done by converting a data structure to a sequence of bytes or characters. In COMP 50, we will take advantage of the facilities that Racket provides and instead convert data to and from S-expressions, which Racket can read and write on our behalf.

Access to data on disk

This section of the homework gives a short tutorial that explains some of the terminology and organization of files on disk. If you have used "drives" and "folders," you have already encountered the organization.

What's on disk is organized into *filesystems*. (On Windows, a filesystem is called a "drive". I use the Unix terminology, which is appropriate to both Linux and OSX, because they are Unix dialects.) A filesystem is a kind of tree; the internal nodes are called *directories*[3] and the leaf nodes are called *files*. A file stores a sequence of *bytes*. A byte is kind of like a *character* or a string of length 1, except there are exactly 256 possible bytes. (A byte is not exactly the same thing as a character; BSL and ISL support very large character sets by using one, two, or even three or more bytes to represent a single character.)

Within a filesystem, both files and directories are referred to using *pathnames*. A pathname gives a sequence of directory names starting at the root of the filesystem. Here are some examples of pathnames:

- `/comp/50/www/homework/trigrams.page`
- `/h/nr/.profile`

That second path means "in the root directory, in the directory `home` contained within the root, in the directory `nr` contained within `home`, the file `.profile`." Windows uses different notation for pathnames; Windows notation uses horrible colons and backslashes:

- `C:\WINDOWS\TMP`

This is about all you need to know to start using the trigrams teachpack.

[3] On Windows and in some graphical desktops, directories are called "folders."

The trigrams teachpack

To give you access to training corpora and to web pages, I've bundled a number of tools into a special teachpack, which you can install from http://www.cs.tufts.edu/comp/50/packages/trigrams.zip. Once you've installed it, you can put the directive

```
(require comp50/files)
```

into your programs.

Functions in the teachpack

With the teachpack installed, you'll get the following goodies:

```
;; DATA DEFINITION
;; A path-string is one of:
;;  - A path
;;  - A string

;; directory-names : path-string -> (listof string)
;; list the names in the given directory, in alphabetical order

;; build-path : path-string path-string -> path
;; create a path by combining parts using the native notation

;; comp50-path : path-string -> string
;; return the full path of a file relative to a comp50 collection

;; basename : path-string -> string
;; returns the last name in a given path

;; DATA DEFINITION
;; A `1string` is a string of length exactly 1

;; read-1bytes : path-string -> (listof 1string)
;; return the contents of a given file, interpreted as bytes

;; geturl-1bytes : string -> (listof 1string)
;; produce the list of bytes served by the given URL, which must be good

;; DATA DEFINITION
;; An sx is one of:
;;   - A string
;;   - A number
;;   - A symbol
;;   - A (listof sx)

;; write-file-sexp : string sx -> string
;; write the given S-expression to the given named file, returning the file's name

;; read-file-sexp : string -> sx
;; read the given S-expression from the file of the given pathname
```

To help you use the teachpack, here are some functional examples:

```
> (directory-names (comp50-path "language-test"))
(list "abcxyz")
> (directory-names (comp50-path "tiny-language-training"))
(list "English" "French" "German" "Polish")
> (directory-names (comp50-path "language-training"))
(list
 "Czech"
 "English"
 "French"
 "German"
 "Hungarian"
 "Italian"
 "Polish"
 "Spanish")
> (read-1bytes (comp50-path (build-path "language-test" "abcxyz")))
(list "a" "b" "c" "x" "y" "z")
> (geturl-1bytes "http://www.cs.tufts.edu/comp/50/nothing.txt")
(list "N" "o" "t" "h" "i" "n" "g" "’" "s" " " "h" "e" "r" "e" "." "\n")
> (basename "/home/nr/.profile")
".profile"
> (basename "/comp/50/www/homework")
"homework"
> (write-file-sexp "/tmp/test.rktd" '(An S-exp ("string" 7 9)))
"/tmp/test.rktd"
> (read-file-sexp "/tmp/test.rktd")
(list 'An 'S-exp (list "string" 7 9))
> (write-file-sexp "/tmp/image.rktd" (circle 5 "solid" "black"))
write-file-sexp: expects a S-expression as second argument, given #<image>
>
```

Data in the teachpack

In addition to the functions you will use to work with paths, directories, files, and URLs, the teachpack also provides data. The data are organized into the following directories:

- `(comp50-path "language-test")` contains one file called `abcxyz`. The contents of that file are the string `"abcxyz"`. It is there for testing.
- `(comp50-path "language-training")` contains a full set of training corpora written using the UTF-8 byte encoding. It is not useful for testing, but it *is* useful for actual classification. Each sub-directory (as found with `directory-names`) is named after a language. If you use it with `build-path`, as in

  ```
  (build-path (comp50-path "language-training") "English")
  ```

  you will get a directory that contains the English corpus, which is a set of files written in English. To train your model of English, you should read *all* of these files.

  Depending on how clean your code is and on how the file server is feeling at any given moment, training all the models in the teachpack might take anywhere from half a minute to several minutes.
- `(comp50-path "latin1-language-training")` contains the same set of training corpora, but they are encoded using the Latin1 (ISO-8859-1) character set instead of the UTF-8 character set. There is a tradeoff:

  - The Latin1 character set is more likely to use one byte per character and is therefore more likely to produce meaningful trigrams.
  - But the Latin1 character set includes only 256 characters and is totally unable to handle Russian (or any other language that does not use the Roman alphabet).

  You can just pick a collection of training corpora, or you can see which ones give you the best results.

- `(comp50-path "tiny-language-training")` contains a smaller set of much smaller corpora. Here are all the files and their contents:

  ```
  tiny-language-training/English/cat:The cat ate a rat.
  tiny-language-training/French/cat:Le chat a mangé un rat.
  tiny-language-training/German/cat:Die Katze aß eine Ratte.
  tiny-language-training/Polish/cat:Kot zjadł szczura.
  ```

  This directory is intended for testing; for example, you can write `check-expect` expressions that make sure your trigram counts are correct.

Underlying math: Probability

Probability basics and vocabulary

When working with probability, don't trust your intuition. There is ample evidence that human brains are especially bad at reasoning with probability and statistics. Our brains are wired with built-in *biases* and *heuristics* that routinely overestimate the probabilities of some kinds of events and underestimate the probability of others. For example, if a story sounds good, our brains are going to think it more likely. For more on this phenomenon you can see the Wikipedia article on the conjunction fallacy; for lots more, read Daniel Kahneman's book *Thinking, Fast and Slow*.

Since you can't trust your brain, who can you trust? Trust yourself to work out the math. Here's the vocabulary:

Formula	Thing or pronunciation
A	A proposition, like "Linda is a bank teller"
B	Another proposition, like "Linda is active in the feminist movement"
H	Another proposition, often a *hypothesis*, like "this document is in French"
O	Another proposition, often an *observation*, like "this document contains the letters `oue`"
E	Another proposition, often *evidence* (e.g., from an observation)
$A \wedge B$	Pronounced "A and B"
$A \vee B$	Pronounced "A or B"
$P(A)$	Pronounced "Probability of A"
$P(AB)$	Pronounced "Probability of A and B"
$P(A\|B)$	Pronounced "Probability of A *given* B"
$P(H\|O)$	Pronounced "Probability of H *given* O"
$P(O\|H)$	Pronounced "Probability of O *given* H"

Table 1: Notations and vocabulary of probability

The critical bit here is the vertical bar pronounced "given." That bar is meaningful only inside the context of a $P(...)$, The technical name for a probability with given is a *conditional probability*. The condition is what's given:

- Given that the article says "ball," what's the probability that it's about business? $P(Business|Ball)$.
- Given that the article says "revenue," what's the probability that it's about business? $P(Business|Revenue)$.

You won't quite get to the stage of computing these probabilities—the arithmetic is just a little too complicated to be worth it. But you'll compute similar probabilities.

Mathematical reasoning about probability

The keys to the whole kingdom come from just one equation:

$$P(A \wedge B) = P(A)P(B|A)$$

which says "The probability of A and B is the same as the probability of A times the probability of B given A. Examples:

- If you have two six-sided dice, the probability of throwing eleven is equal to the probability of throwing at least five on the first die, times the probability of throwing on the second die, a number that when given the number on the first die, adds to eleven.
- The probability of winning the daily double is the probability of betting the winning horse in the first race, times the probability of betting the winning horse in the second race *given* that you bet the winning horse in the first race.[4]

Because order doesn't matter, $A \wedge B$ is exactly the same as $B \wedge A$. We therefore often write

$$P(A \wedge B) = P(A)P(B|A) = P(B)P(A|B)$$

In other words, when you have two things occurring together, you get to choose which one is given.

Independence

What if B is independent of A? Then the probability of B given A is the same as the probability of B without any knowledge of A:

$P(B|A) = P(B)$ (**when independent**)

If A and B are independent, then $P(A \wedge B) = P(A)P(B)$.

[4] It's no accident that these examples are from gambling. The theory of probability was *invented* by gamblers.

From trigram counts to probabilities

OK, so you've got a model that tells you how many times each trigram occurs in a corpus. We're going to make an *outrageous* assumption and pretend that all trigrams in a model are independent. This assumption is absolutely not true—but the results are still good enough to classify a lot of documents.

What we'll compute is the probability of seeing a given document E (evidence) under the hypothesis (H) that the document was generated using the given model. The second outrageous assumption is that the probability of producing a document is the same as the probability of producing all its trigrams. Under these two assumptions, we have

$$P(E|H) = P(E_1 \ldots E_n|H)$$

$$P(E|H) = P(E_1|H) \times P(E_2|H) \times \cdots \times P(E_n|H)$$

Where E_1 is the first trigram, E_2 is the second trigram, and so on.

The arithmetic of unlikely events

The World-Wide Web contains more than a billion web pages. Therefore probability of producing any one of them is on average no more than one in a billion. A billion is 10^9 so immediately we're talking -90 decibans. But it gets worse: the trigram models are *far* more likely to produce gibberish than to produce real web pages.[5] So the probabilities you will compute are much, much smaller. Tiny probabilities present some computational problems:

- Exact arithmetic on fractions with thousands of digits is very, very slow.
- *Inexact* arithmetic is not capable of representing very small probabilities—it will round them to zero.

The solution is your old friend (or enemy) the logarithm. You will calculate the *log* of all probabilities, in decibans. The equations reduce to

$$\log P(E|H) = \log P(E_1|H) + \log P(E_2|H) + \cdots + \log P(E_n|H)$$

You can compute logarithms in decibans using the function `log-decibans`:

```
(define log-10/10 (/ (log 10) 10))

;; log-decibans : number -> number
;; return the 10 times the base-10 logarithm of the given number
(define (log-decibans x)
  (/ (log x) log-10/10))

(check-within (log-decibans 1)      0 EPSILON)
(check-within (log-decibans 10)    10 EPSILON)
(check-within (log-decibans 100)   20 EPSILON)
(check-within (log-decibans 1/10) -10 EPSILON)
```

[5] Think of the trigram models as monkeys trying to type Shakespeare. Only we have some English monkeys, some French monkeys, and so on.

The probability of a single trigram

So given a model H and a trigram E_i, we can estimate the probability $P(E_i|H)$ very simply: the number of times E_i was observed divided by the total number of observations. There's just one problem with this idea:

- If you encounter a trigram that doesn't appear in the corpus, this technique assigns it zero probability. Everything multiplies out to zero, and your total estimate of $P(E|H)$ is zero.

  This outcome is clearly no good: for example, a document written in English might contain a math formula whose trigrams don't appear anywhere in the training corpus. It's still in English, and you want to classify it so.

The solution to this problem is to take some probability away from the *observed* trigrams and assign it to the *unobserved* trigrams. This technique brings the probabilities of observed and unobserved trigrams closer together, making the probability distribution less jagged, and so it is called *smoothing*. I'm recommending you use a simple method called Laplace smoothing.

With Laplace smoothing, you pretend to have an additional α observations of *each possible trigram*. The α is a *smoothing parameter*, and I recommend you set $\alpha = 1$ (although you are welcome to try smaller values). The Wikipedia page uses an equation

$$\hat{\theta}_i = \frac{x_i + \alpha}{N + \alpha d}$$

where

θ_i	is your best estimate of $P(E_i\|H)$
x_i	is the number of times trigram E_i is observed in the corpus H
α	is the smoothing parameter, often 1
N	is the total number of trigrams observed in the corpus H
d	is the total number of *possible* trigrams, which is 256^3

So your estimate should be

$$\hat{P}(E_i|H) = \frac{\text{count}(E_i, H) + \alpha}{\text{observations}(H) + \alpha 256^3}$$

where $\text{count}(E_i, H)$ is the number of times trigram E_i was observed in model H and $\text{observations}(H)$ is the total number trigrams observed in model H.

Classification using bit vectors

There is an alternative method of classification that uses bit vectors instead of probabilities. It can be relatively fast, and depending on the training corpora and the actual documents being classified, it sometimes does better than a probabilistic classifier and sometimes worse. The good news is that the code is fairly simple. That bad news is that it is based on linear algebra in high-dimensional spaces—math you may not be familiar with, and that I won't be able to explain very well.

Imagine a hypercube in 256^3 dimensions. That's one dimension for every trigram. We can summarize an entire corpus by a single vector in this high-dimensional space:

- Each dimension corresponds to a trigram.

- If the given trigram is *present* in the corpus, the value of the vector along that dimension is 1.
- If the given trigram is *absent* from the corpus, the value of the vector along that dimension is 0.

The *length* of this vector is the square root of the sums of the squares of the dimensions of the vector, which is just the square root of the number of *distinct* trigrams observed. (I told you the code would be fairly simple.)

Using exactly the same technique, you can also compute a vector for a document.

Your classifier can then compare the corpus and the document by looking at whether the vectors are pointing in roughly the same direction. That is, we want to know if the *angle* between the vectors is small. Lo and behold, if we want to know the angle between two vectors, there is a lovely (if inexplicable) mathematical fact:

> The dot product of two vectors it is the product of the lengths of the two vectors and the cosine of the angle between them.

The dot product of two vectors V and V' is the sum over all dimensions of the components of V and V' from the given dimension. For the restricted vectors we're talking about, that dot product is simply the number of *distinct* trigrams that the document and the model have in common:

- Trigram present in document and model? $1 \times 1 = 1$ and that trigram contributes 1 to the dot product.
- Trigram missing from one, the other, or both? Zero times anything is zero, and that trigram contributes nothing to the dot product.

With this information you can easily compute the cosine of the angle between two vectors. Because smaller angles lead to larger cosines, you can use the cosine itself as the score. You can also easily test that when you compare a document with itself, you correctly get a cosine of 1.

Classification

A classifier is given a list of models and a document and tells which model best explains the document.

- Warning: **Not all URLs contain documents.** Your classifier must include a model that means "none of the above." A reasonable choice is a model in which all trigrams are equally likely. Because of the way Laplace smoothing works, the model with an empty tree and no observations is such a model.

  For the bit-vector classifier, it is not practical to create a tree that has all 256^3 trigrams in it. You will have to find some other way of using cosines and dot products to compare with that model. You can recognize the model easily: it is the only model that has not observed any trigrams. For purposes of cosine classification, you want to pretend that it has observed *every* trigram.

I intend for you to write two classifiers: one based on probability, and one based on bit vectors and dot products. A classifier should take a URL as input and return an `(alistof number)` that associates each model to a *score*. Include a model of every training corpus, *plus* the `"Other"` model.

For testing purposes, you probably will also want to be able to classify strings.

Data definition

An `(alistof X)`, aka association list of `X`, is either

- `empty`
- `(cons (list name score) pairs)` where `name` is a name, `score` is a number, and `pairs` is an `(alistof X)`.

Expectations and advice for the homework

I expect you to submit a solution that contains these functions:

- A function we can run that builds models from the training corpora, *plus* the `"Other"` model, and saves them to disk.
- A probabilistic classifier of URLs that uses models saved on disk
- A bit-vector classifier of URLs that uses models saved on disk
- Functions that save a list of models to disk and restore a list of models from disk.

Advice

I have lots of advice.

Binary search trees Incorporate your binary-search-tree solutions wholesale. You'll need insertion, search, and conversion to and from lists. Don't forget to include your tests!

Models vs trees Your main data structures will be `model` and search trees of type `(bst number)`. The tree stores trigram counts and will be part of a model. Any time you create a function on your wish list, you'll need to think about whether trees or models go in and come out. If your code seems awkward, don't be afraid to change decisions!

When possible, avoid recursive functions My solution uses the standard list functions aggressively, and I recommend you do the same.

Your BST tools will include a number of recursive functions, but these functions are already written. Don't revisit them: they work; they are tested; and they are ready to use. Treat your BST tools as you would treat a teachpack.

The one place you absolutely *must* use recursion is when you are visiting all the trigrams in a list of bytes. You will need to do this in more than one context, so if you can do it, *I encourage you to create an abstraction for "combine all trigrams."* It should not look that different from the `fold` function in the book or the `my-foldr` function we developed in class.

Depending on how you choose to tackle the construction of models, you might also want a small number of recursive functions that consume binary trees. Or you could equally well choose to convert the trees to lists and use the standard list-abstraction functions.

Testing In many respects the most difficult part of this assignment is writing the functional examples and tests. You can write a very few tests using models and trees directly. But if everything that goes into your tests is a model or tree that is fully written out using `make-model` and `make-node`, you probably will not finish. What I recommend instead is that most of your tests be built using data that is constructed by other functions (which are themselves tested).

For example, let's suppose I want to test the a function that counts how many times a trigram appears in a model, and I *already* have a *tested* function that builds a model from a list of 1-character strings. Then I might write a test something like this:

```
(check-expect
  (trigram-count "at " (model-of-chars (explode "A fat cat and a rat ate that")))
  3)
```

Here's what I want you to notice about this test:

- To understand the test, I need know the signature and purpose statements of functions `trigram-count` and `model-of-chars`.
- Once I understand, I can immediately count how many times `"at "` occurs in the string, and I can confirm that it occurs three times (once each at the ends of the words "fat", "cat", and "rat"). In other words, it's very easy for me to see that the test is right.
- If this test fails, I won't be sure whether to blame `trigram-count` or `model-of-chars`. I should already have tests for `model-of-chars`, and these may help me figure out which function is at fault. But in the worst cases, I may have to write out separate tests with full data, so I know which function is wrong.

Save your full-data tests for two situations:

- Use full data for the foundations of your program.
- Use full data when other tests fail and you don't know why.

URLs to try I have put up eleven samples at this URL:

http://www.cs.tufts.edu/~nr/cgi-bin/50sample.cgi?one

You can change `one`, to `two`, `three`, and so on up through `eleven`. I may add more. Most of these samples use the UTF-8 character set, but you can train on whichever character set you want.

Sample output Here are my results classifying a famous text.

- First, the likelihood of generating the text from each model:

```
'(("English"   #i-124879.38840876492)
  ("German"    #i-136502.98974519692)
  ("French"    #i-138755.4804011435)
  ("Spanish"   #i-140403.25530245638)
  ("Italian"   #i-140643.53502622052)
  ("Hungarian" #i-141477.80928918015)
  ("Czech"     #i-142751.49331737417)
  ("Polish"    #i-143355.53183452427)
  ("Unknown"   #i-156920.91613972152))
```

  Although all the models say that the monkeys are wildly unlikely to type this document, the English monkeys are far less unlikely than anybody else. The document is definitely in English.

- Here are the results of using the bit-vector classifier on the same text:

```
'(("English"   #i0.36898273367830303)
  ("German"    #i0.25326695630096535)
  ("French"    #i0.24965274414918953)
  ("Italian"   #i0.24356957546717814)
  ("Spanish"   #i0.24140594907631213)
  ("Polish"    #i0.2054360930761648)
  ("Czech"     #i0.19650868079989064)
  ("Hungarian" #i0.19644545018071768)
  ("Unknown"   #i0.007751224544134543))
```

The rankings are a bit different, but English is still clearly on top.

Here's a different sample, from a text that mixes two languages:

```
'(("English" #i-258080.31739165334)
  ("French"  #i-259023.08641358386)
  ("Spanish" #i-270013.69208337733)
  ("German"  #i-271836.10062956245)
  ("Italian" #i-273627.51560851745)
  ("Hungarian" #i-281419.25852654263)
  ("Czech"   #i-281716.0129245808)
  ("Polish"  #i-284114.33976111293)
  ("Unknown" #i-307050.5955772443))
```

Although a thousand decibans looks like a lot, the quantitative probabilities are not to be trusted; this one should be considered a tie between English and French.

I also classified a mystery document in which that most likely outcome was -1703 decibans and the least likely was -1712 decibans. In this case the classifier has no idea what is going on.