INFO-Y004: Natural Language Processing
Assignment 1: Language Modeling

Brian Delhaisse

March 22, 2015

1 Introduction

The goal of this assignment is to build a language model and to detect in which language a test document is written. For this purpose, we will use a 3-gram letter model. Based on this model and on what we train, we should also be able to generate random output text.

For this assignment, the programming language that I used is Python (2.7.5).

2 Data Collection

For this project, I choose to train on documents that are written in English, French, and Italian. Each document is approximately composed of 20,000 words and is taken from Wikipedia[1]. The two test files are respectively in English[2] and in French [3], and both are not used for training. To get those documents, I used the python library "wikipedia"[4] and save their content into text files.

3 Preprocessing and Design

The preprocessing step consist to clean/simplify the data, and keep the essential elements. In my code, the preprocessing step first transforms the non-ascii characters into ascii characters; for example the characters with accent such as é, è, ê,... are transformed into e. After this step, only the lowercase characters of the alphabet and the space character are kept, all the other characters (special characters, numbers,...) are discarded because they are not useful for the language model, that is, they don't help to detect in which language the text is written.

Concerning the design, I am principally using two 3D-table of dimension $27 \times 27 \times 28$. Each dimension has as indices the 26 characters of the alphabet + the space character. The last dimension contains an extra index to store the probabilities $P(l_2|l_1) = \frac{count(l_1,l_2)}{count(l_1)}$.

- The first 3D table has as elements the $P(l_i|l_{i-1}, l_{i-2})$, and has been smoothed (see the next section) and is used to compute the perplexity of a model when applied on a test file.

[1] For the French document:http://fr.wikipedia.org/wiki/France
For the English document: http://en.wikipedia.org/wiki/United_Kingdom
For the Italian document: http://it.wikipedia.org/wiki/Italia

[2] For the English test document: http://en.wikipedia.org/wiki/United_States

[3] For the French test document: http://fr.wikipedia.org/wiki/Belgique

[4] The Wikipedia Python library: https://pypi.python.org/pypi/wikipedia/

- The 2nd one contains as elements the cumulative probabilities of the first 3D table *before* the smoothing, and is used to generate output text.

I used 3D-tables because we are computing a trigram model for letters, and this data-structure which is compact and has O(1) lookup/insert will be useful for the smoothing (see next section). For words, I would have probably used nested dictionnaries.

4 Methods

4.1 Add-1 smoothing

I used the Laplace smoothing for simplicity reasons; it's really easy to compute this smoothing especially for a table of characters. For words, I would have probably used the extended Kneser-Ney smoothing.

The smoothing will be useful to compute the perplexity. The reason why smoothing is needed and is useful is because in the test file, it could be that there are combinations of letters that were never saw in the training set, and have thus an associated probability of 0. When computing the perplexity or computing the probability that a model represents a certain language, it could be disastreous to have probabilities equal to 0, because they would correspond to an ∞ perplexity or a probability 0 that the model represents the language. In order to generalize well, we need to smooth.

The Laplace smoothing consists of taking a little bit of the probability distribution to letters that have a probability different to 0, and to redistribute it to all the other letters that have a probability 0. More specifically, the Laplace smoothing assume that we have seen each letter one more time than we did. The Laplace smoothing is given by:

$$P_{Laplace}(l_i|l_{i-1}, l_{i-2}) = \frac{count(l_{i-2}, l_{i-1}, l_i) + 1}{count(l_{i-2}, l_{i-1}) + V} \tag{1}$$

where V is the number of types, in our case, it's equal to 27 (the alphabet characters + the space character).

4.2 Perplexity

As mentionned in the course, "The best model is the one that gives the highest probability to a given sentence"[5]. The perplexity of the whole document is given by:

$$PP(W) = P(l_1, l_2, ..., l_N)^{-\frac{1}{N}} = \sqrt[N]{\frac{1}{P(l_1, l_2, ..., l_N)}} \tag{2}$$

where N is the number of letters in the document. By looking at this equation (2), we can see that lower the perplexity is, better the model is.

This equation 2 can be rewritten, using the chain-rule, as:

$$PP(W) = \sqrt[N]{\prod_{i=1}^{N} \frac{1}{P(l_i|l_1, ..., l_{i-1})}} \tag{3}$$

By the Markov assumption and for 3-grams, we can finally rewrite this equation as:

$$PP(W) = \sqrt[N]{\prod_{i=1}^{N} \frac{1}{P(l_i|l_{i-2}, l_{i-1})}} \tag{4}$$

[5]Lecture3 - Language Modeling, slide27

By computing the perplexity of each model on the test file, the model which has the lowest perplexity is with a high probability the language in which the test file is written.

5 Results

To generate random outputs based on our language model (and indirectly on what we train), I'm using the 2nd 3D table, where elements are cumulative probabilities. To generate random outputs, we draw a random number between 0 and 1 and by looking where this number fit in the cumulative probability distribution, we will get our first letter. Then, we can do the same for the 2nd letter and 3rd letter. This cumulative probability distribution is handy because we don't take each time the letter which has the maximum probability $P(l_i|l_{i-1}, l_{i-2})$, but we sampled them based on the distribution.

- Example of generated text for the English language:
  zec he uk in the wasially of gance brigh ingland ingdom by itionquent pea und ithe wity stand northourelow paincentry th cas parch anned in in wast hat orts anicalis affer is a a gregaelant forth capire pown fulat voleand fraticuland affeated of to histe in the in of the ined wasingdomy and a gogre
  Perplexity score on the English test file: 8.042119117468124
  Perplexity score on the French test file: 14.860362390752547

- Example of generated text for the French language:
  u ne latortiquelutreminoucen ete la l comaitalimplus et sede surbasse dencortes es aujouvrempolou xient de deres du maisestant lalinucinal ela re l clue sien ce fortervarcs g a de et la conde km gracten magne les clouelogiere de la font dennaine pornerege-teue trestion et de con et son extes sondest
  Perplexity score on the English test file: 16.614221351056255
  Perplexity score on the French test file: 7.832592772616831

- Example of generated text for the Italian language:
  e ittengone croproprengobarciperentraiodel il pagliale dei iminizza durel te e de g pubile ne uni si minongardiconia via ma e a oto d i un battato a etea ranioncenza pa comendo atoreliccarda ecominnizza e na se gueridiventreziodecisvia setalligno e opo di a sta e l ilicultural ovarialla venonfia gia
  Perplexity score on the English test file: 18.636655549808044
  Perplexity score on the French test file: 15.70737989985601

As mentionned in the previous section, the model which has the lowest perplexity is with a high probability the language in which the test file is written. We can see that for the 2 test files, that our language model correctly detects the language of the document. Each probabilistic model is saved into a file called "data.txt".

6 Summary

To conclude, the 3gram letter model is quite good to recognize the language in which a text is written. The words of the generated text have generally no meaning, but we can still detect the language on which is based because the (short) generated words are real words of this one, and the frequency of each letter is different for each language.

Appendix - code

To launch the code, just type the command *python assignment1-letter.py* into the console.

In my code, I have 5 functions:

1. the *preprocessing(filename)* function which takes as argument the name of the file to read. It preprocesses the text as described in the section 3. The text is then translated to natural numbers to represent the indices of the 3d-table (e.g. the letter a is mapped to 0, b to 1, etc.). It's this translated text that is returned by the function.

2. the *buildModelLetter(filename)* function which takes as argument the name of the file to preprocess. This function builds the trigram letter model; it computes the smoothed probabilities and the cumulative probabilities which are respectively useful for computing the perplexity and generating text. It returns a list which contains the two 3D-table sticked together, and two 1D-table which contains the probabilities $p(l_i)$ and the cumulative probabilities.

3. the *checkModel(testFilename, listOfModels)* function which computes the perplexity for each model in the *listOfModels* on the test file specified by the *testFilename* argument. This test file is of course preprocessed before computing the perplexity. This function returns a list of perplexities.

4. the *saveModel(filename, model)* function which saves the probabilistic *model* into the file specified by the *filename* argument.

5. the *generateOutput(model, length)* function which print text of a certain *length* to the standard output, and is based on the *model* given.