Language and Computation

LING 227 01 / 627 01 / PSYC 327 01

Midterm take-home exam

Due: March 27, 2014

The solutions to the midterm must be handed in on paper at the beginning of the class. It must be typed (printed), no handwritten solution will be accepted. Learning how to type mathematical formulae, code and pseudo-code is part of the skills one has to acquire.

Additionally, you will also send your code in an email by 4 pm on the day of the deadline to `tamas.biro@yale.edu`. The file name must contain both your first name and last name, and **must not** contain white space characters.

This midterm contributes 20% to your final grade.

Comparing text categorization systems

We discussed in length the problem of text categorization in class. It is your turn now to implement two such approaches. The goals of this midterm are as follows:

1. Provide hands-on experience with one of the simplest computational linguistic problems, which nevertheless includes elements of many aspects of the field (such as n-grams, machine learning, familiarity with some corpora, various evaluation metrics, and error analysis).
2. Develop your programming skills, and show you some useful tricks.
3. Develop the skills that are necessary to critically assess a natural language processing system, by reflecting on how it relates to various aspects of natural language.

Here are two articles that summarize different approaches to text classification using n-grams (both are also available on the course website and on classes.v2, among "Resources/Readings"):

> Cavnar, William B., and John M. Trenkle (1994). "N-gram-based text categorization." *Proceedings of the 1994 Symposium on Document Analysis and Information Retrieval*, pp. 161–174.

> Damashek, Marc (1995). "Gauging similarity with n-grams: Language-independent categorization of text." *Science* 267:843–848.

You will first read these two articles. They both provide a solution for the problem of text categorization based on n-grams. You will then implement and test both approaches. You will finally report your results in a (scholarly) squib.

You will submit an at most **4-page-long "essay"** (single-spaced is ok, but strictly not more than four pages!). Please also provide your **codes** as an appendix, both on paper and in an email. The appendix does not count toward the page limit.

Your text can heavily rely on these two articles, referring to them for details. The squib should rather focus on what you have to add to these articles, details that are specific to your implementations, and so on. You may want to slightly alter the two methods relative to their original forms in the cited articles, in order to make them more comparable. Feel free to include bullet points, etc. in order to condense your paper.

Nevertheless, the structure of your paper should, at least symbolically, follow the basic structure of an article in computational linguistics (such as the *Cavnar and Trenkle* paper, but not the *Damashek* paper):

- **Abstract** [can be very short]
- **Section 1:** General introduction to the topic, main motivations.
- **Section 2:** Specifying the concrete research question being tackled.
- **Section 3:** Methodology, details of the experiments, implementations.
- **Section 4:** Primary results of the experiments described in Part 3.
- **Section 5:** Discussion of the results vis-à-vis the research question posed in Part 2. Problems and limitations.
- **Section 6:** General conclusion: what have we contributed to the topic introduced in Part 1? Directions for future research.
- **References** according to scholarly standards. [I accept all reference systems, as long as they are used consistently.]

In particular, you will pose the following **research question**:

> Does the *cosine measure* or the *out-of-place measure* provide a more useful character-based *n*-gram approach to retrieving texts belonging to a specific genre?

Beside shortly explaining and implementing each of the two approaches, you will also compare them: how do they differ conceptually, how do they differ in performance, and why do you think one works better than the other?

1.1 Using the Brown corpus with NLTK

During the experiment, you will work on the famous *Brown Corpus* (see, e.g., `http://fss.plone.uni-giessen.de/fss/faculties/f05/engl/ling/help/materials/restricted/brown/file/brown.pdf`). It comes conveniently together with NLTK: refer to the section 'Brown Corpus' in the NLTK book (`http://www.nltk.org/book/ch02.html`). See also the `CONTENTS` and `README` files in your `nltk_data/corpora/brown/`.

Try out the following commands (beside those discussed by the NLTK book):

```
>>> import nltk
>>> nltk.download('brown')
>>> brown.categories()
>>> brown.fileids()
>>> brown.raw('cn19')
>>> brown.sents('cn19')
>>> brown.words('cn19')
>>> brown.fileids(categories='news')
```

Your *training corpus*, based on which you will create your reference vectors, will be the first ten files of each category (e.g., `ca01` to `ca10` in the *news* category). Unfortunately, you will have to ignore the categories *science_fiction* and *humor*, for they include too few files.

Subsequently, you create a *test corpus* from all the files numbered between `11` to `15` (for instance, `ca11` to `ca15` in the *news* category). That will result in a test corpus of 65 files (given the 13 categories with more than 15 files each).

The tasks that each of your n-gram models have to solve is the following:

1. For each file in the test corpus, provide the category C_i to which it most probably belongs.

2. Given category C_i (such as, *news* or *adventure*), retrieve all texts in the test corpus that your model categorizes as belonging to C_i.

The file names in the Brown corpus (and the `corpora/brown/cats.txt` file) provide the *Golden standard* to which your models' output should be compared. For each C_i, a *true positive* is a text that has been correctly categorized as belonging to category C_i; a *false positive* is a text that actually does not belong to C_i, but has been categorized as such; a *true negative* is a text that has been correctly categorized as not belonging to C_i, and finally a *false negative* is a text that has been erroneously categorized as not belonging to C_i.

You will be able to employ different evaluation measures (refer to the lecture slides of February 06, pp. 7–9):

- **Accuracy:** The proportion of the files that have been correctly classified (either as *news*, or as *hobbies*, or as…) in the entire set of 65 files.
- **Precision:** For each category C_i, the proportion of the true positives to those that have been categorized as belonging to category C_i.
- **Recall:** For each category C_i, the proportion of the true positives to those that in reality belong to C_i (true positives and false negatives).
- **F-measure:** For each C_i, the harmonic mean of precision and recall.

Finally, you will also perform **error analysis**: a closer look at the results may reveal where things tend to go wrong. For instance, are there categories that are frequently confused? What happens if you merge these categories? Is there a specific text that is repeatedly misclassified? Check that specific text, and you might discover a sometimes extremely simple reason why it is an *outlier*. Sometimes, however, uncovering the reasons of some results is extremely hard.

1.2 Python tricks

First, since you will generate n-grams on the character level, you may wish to transform each text into a single string. Here is a first approximation:

```
>>> ' '.join(brown.words('cn19'))
```

But, before doing so, it may be useful to perform some pre-processing: removing punctuation marks, decapitalizing all upper case characters, etc. Does it improve the overall performance of your models if you also remove *stopwords*? A list of stopwords is also provided by NLTK:

```
>>> nltk.download('stopwords')
>>> from nltk.corpus import stopwords
>>> stopwords.words('english')
```

It is easy to generate n-grams of a string in Python. The trick is to first create slices of the string and then to `zip` them:

```
>>> string="abcdefg"
>>> string1=string[1:]
>>> string1
'bcdefg'
>>> zip(string,string1)
[('a', 'b'), ('b', 'c'), ('c', 'd'), ('d', 'e'), ('e', 'f'), ('f', 'g')]
>>> string2=string[2:]
>>> string2
'cdefg'
>>> zip(string,string1,string2)
[('a', 'b', 'c'), ('b', 'c', 'd'), ('c', 'd', 'e'),
('d', 'e', 'f'), ('e', 'f', 'g')]
```

Finally, please remember the remark made in the notes of February 13: You should count frequencies by using a hash table (or a dictionary in Pythonese), and not a $k \times k \times \ldots \times k$ table (where k is the size of the alphabet):

```
>>> string = "aabbaabbaaaa"       # a sample string
>>> trigrams=zip(string,string[1:],string[2:])
>>> counts={}                     # create an empty dictionary
>>> for trigram in trigrams :
...     if trigram in counts :    # check if dictionary entry exists
...         counts[trigram] += 1  # increase the count by 1
...     else :
...         counts[trigram] = 1   # create dictionary entry with value 1
...
>>> counts                        # trigrams with their frequencies
{('a', 'b', 'b'): 2, ('b', 'a', 'a'): 2, ('a', 'a', 'a'): 2,
('a', 'a', 'b'): 2, ('b', 'b', 'a'): 2}
```

This hash table is much more compact than a sparse matrix of size k^n.