FNLP: Lab Session 2

Smoothing and Authorship Identification

1 Aim

The aims of this lab session are to explore 1) the Laplace, Lidstone and Good-Turing smoothing methods for language models and 2) the use of language models in authorship identification. Successful completion of this lab will help you solidify your understanding of smoothing (important not just for LMs but all over NLP), perplexity (important also for assignment 1), and one type of text classification (authorship identification). By the end of this lab session, you should be able to:

- Compute smoothed bigram probabilities by hand for simple smoothing methods.
- Train an nltk language model with smoothing for unseen n-grams
- Make use of language models to identify the author of a text

2 Running NLTK and Python Help

2.1 Running NLTK

NLTK is a Python module, and therefore must be run from within Python. To get started on DICE, type the following in a terminal window:

```
$: python
>>> import nltk
```

2.2 Python Help

Python contains an inbuilt help module that runs in an interactive mode. To run the interactive help, type:

```
>>> help()
```

To exit, press CTRL-d.

If you know the name of the module that you want to get help on, type:

```
>>> import <module_name>
>>> help(<module_name>)
```

To exit, type "q" (for "quit").

If you know the name of the module *and* the method that you want to get help on, type:

```
>>> import <module_name>
>>> help(<module_name>.<method_name>)
```

To exit, type "q" (for "quit").

3 Introduction

Before continuing with this lab sheet, please download a copy of the lab template (`lab2.py`) for this lab from the FNLP course website. This template contains code that you should use as a starting point when attempting the exercises for this lab.

In some of the exercises, you'll use NLTK's `NgramModel` to train language models. The initialisation method for `NgramModel` is:

```
def __init__(self, n, train, pad_left=False, pad_right=False,
             estimator=None, *estimator_args, **estimator_kwargs):
    """
    Creates an ngram language model to capture patterns in n consecutive
    words of training text.  An estimator smooths the probabilities derived
    from the text and may allow generation of ngrams not seen during
    training.

    :param n: the order of the language model (ngram size)
    :type n: C{int}
    :param train: the training text
    :type train: C{iterable} of C{string} or C{iterable} of C{iterable} of C\
{string}
    :param estimator: a function for generating a probability distribution--\
-defaults to MLEProbDist
    :type estimator: a function that takes a C{ConditionalFreqDist} and
          returns a C{ConditionalProbDist}
    :param pad_left: whether to pad the left of each sentence with an (n-1)-\
gram of <s>
    :type pad_left: bool
    :param pad_right: whether to pad the right of each sentence with </s>
    :type pad_right: bool
    :param estimator_args: Extra arguments for estimator.
        These arguments are usually used to specify extra
        properties for the probability distributions of individual
        conditions, such as the number of bins they contain.
        Note: For backward-compatibility, if no arguments are specified, the
        number of bins in the underlying ConditionalFreqDist are passed to
        the estimator as an argument.
    :type estimator_args: (any)
    :param estimator_kwargs: Extra keyword arguments for the estimator
    :type estimator_kwargs: (any)
    """
```

4 Smoothing

In the final exercise of Lab 1, you were asked to calculate the probability of a word given its context, using an bigram language model with no smoothing. For the first two word-context pairs, these bigrams had been seen in the data used to train the language model. For the third word-context pair, the bigram had not been seen in the training data, which led to an estimated probability of 0.0.

Zero probabilities for unseen n-grams cause problems. Suppose for example you take a bigram language model and use it to score an automatically generated sentence of 10 tokens (say the output of a machine translation system). If one of the bigrams in that sentence is unseen, the probability of the sentence will be zero.

Smoothing is a method of assigning probabilities to unseen n-grams. As language models are typically trained using large amounts of data, any n-gram not seen in the training data is probably unlikely to be seen in other (test) data. A good smoothing method is therefore one that assigns a fairly small probability to unseen n-grams.

We'll implement two different smoothing methods: **Laplace** (add-one) and **Lidstone** (add-alpha), and we will also consider the effects of **backoff**, which is implemented in NLTK's `NgramModel`.

(NLTK also includes implementations of Laplace and Lidstone smoothing in its probability module; if you wish to look at the implementations they are here:
http://www.nltk.org/api/nltk.html#module-nltk.probability
However the point of this lab is to implement these simple methods yourself to make sure you understand them.)

4.1 Maximum-Likelihood estimation

Before implementing any smoothing, you should make sure you understand how to implement maximum likelihood estimation. In last week's lab, we used NLTK to do this for us by training a bigram language model with an MLE estimator. We could then use the language model to find the MLE probability of any word given its context. Here, you'll do the same thing but without using NLTK, just to make sure you understand how. We will also compare the smoothed probabilities you compute later to these MLE probabilities.

Exercise 0
Code has been provided that extracts all the words from Jane Austen's "Sense and Sensibility", and then computes a list of bigram tuples by pairing up each word in the corpus with the following word. Using these unigrams and bigrams, fill in the remaining code to compute the MLE probability of a word given a single word of context. Then uncomment the test code to compute the probabilities:

a. $P_{MLE}(\text{“end”}|\text{“the”})$

b. $P_{MLE}(\text{“the”}|\text{“end”})$

Make sure your answers match the MLE probability estimates from Exercise 5 of Lab 1, where we used NLTK to compute these estimates.

4.2 Laplace (add-1)

Laplace smoothing adds a value of 1 to the sample count for each “bin” (possible observation, in this case each possible bigram), and then takes the maximum likelihood estimate of the resulting frequency distribution.

Exercise 1
Assume that the size of the vocabulary is just the number of different words observed in the training data (that is, we will not deal with unseen words).

Add code to the template to compute Laplace smoothed probabilities, again without using NLTK.
Hint: if you have trouble, study the equations and example in Lecture 4

Now uncomment the test code and look at the estimates for:

a. $P_{+1}(\text{“end”}|\text{“the”})$

b. $P_{+1}(\text{“the”}|\text{“end”})$

How do these probabilities differ from the MLE estimates?

4.3 Lidstone (add-alpha)

In practice, Laplace smoothing assigns too much mass to unseen n-grams. The Lidstone method works in a similar way, but instead of adding 1, it adds a value between 0 and 1 to the sample count for each bin (in class we called this value alpha, NLTK calls it gamma).

Exercise 2
Fill in the code to compute Lidstone smoothed probabilities, then uncomment the test code and look at the probability estimates that are computed for the same bigrams as before using various values of alpha.

What do you notice about using alpha = 0 and alpha = 1? (Compare to the probabilities computed by the previous methods.)

What about when alpha = 0.01? Are the estimated probabilities more similar to MLE or Laplace smoothing in this case?

4.4 Backoff

Now we will look at the effects of incorporating **backoff** in addition to some of these simple smoothing methods. In a bigram language model with backoff, the probability of an unseen bigram is computed by "backing off": that is, if a word has never been seen in a particular context, then we compute its probability by using one fewer context words. Backing off from a bigram model (one word of context) therefore means we'd get estimates based on unigram frequencies (no context).

The mathematical details of backoff are a bit complex to ensure all the probabilities sum to 1. You needn't understand all the details of backoff but you should understand these basic principles:

- Bigram probabilities for *seen* bigrams will be slightly lower than MLE in order to allocate some probability mass to unseen bigrams.
- The unigram probabilities inside the backoff (i.e. the ones we use if we didn't see the bigram) are similar in their relatives sizes to the unigram probabilities we would get if we just estimated a unigram model directly. That is, a word with high corpus frequency will have a higher unigram backoff probability than a word with a low corpus frequency.

Look back at the initialization method for `NgramModel` earlier in the lab. If you pass in `MLEProbDist` as the estimator (which we did in the last lab), then no backoff is used. However, with any other estimator (i.e., smoothing), the `NgramModel` **does** use backoff.

Exercise 3
Use the code we have provided to train a bigram language model using Jane Austen's "Sense and Sensibility" and **Laplace** smoothing. (By using Laplace as the estimator, you are also turning on backoff in `NgramModel`.) Use this language model to compute the probability of the same bigrams we've been looking at all along. Uncomment the test code to see the results.

- Compare the probabilities you get for these bigrams to what you got when you computed Laplace yourself. Why are the probabilities produced by `NgramModel` with Laplace smoothing different from the probabilities you computed yourself?
- Now look at the estimated probabilities $\hat{P}(\text{"end"}|\text{"the"})$ and $\hat{P}(\text{"the"}|\text{"end"})$ as computed by the `NgramModel` and by the previous smoothing methods. Which method(s) produce larger differences between those probabilities? Do all the methods agree about which bigram has higher probability? If not, what is the reason for the difference?

5 Authorship Identification

5.1 Cross-entropy

In language modelling, a model is trained on a set of data (i.e. the training data). The cross-entropy of this model may then be measured on a test set (i.e. another set of data that is different from the training data) to assess how accurate the model is in predicting the test data.

Another way to look at this is: if we used the trained model to *generate* new sentences by sampling words from its probability distribution, how similar would those new sentences be to the sentences in the test data? This interpretation allows us to use cross-entropy for authorship detection, as described below.

`nltkx.NgramModel` contains the following cross-entropy method:

```
     def entropy(self, text, pad_left=False, pad_right=False,
                 verbose=False, perItem=False):
         """
         Calculate the approximate cross-entropy of the n-gram model for a
         given evaluation text.
         This is the average log probability of each item in the text.

         :param text: items to use for evaluation
         :type text: iterable(str)
         :param pad_left: whether to pad the left of each text with an (n-1)-gram\
     of <s> markers
         :type pad_left: bool
         :param pad_right: whether to pad the right of each sentence with an </s>\
     marker
         :type pad_right: bool
         :param perItem: normalise for length if True
         :type perItem: bool
         """
```

Exercise 4
We can use cross-entropy in authorship detection. For example, suppose we have a language model trained on Jane Austen's "Sense and Sensibility" (training data) plus the texts for two other novels (test data), one by Jane Austen and one by another author, but we don't know which is which. We can work out the cross-entropy of the model on each of the texts and from the scores, determine which of the two test texts was more likely written by Jane Austen.

Use:

- A trigram language model with a Lidstone probability distribution, trained on Jane Austen's "Sense and Sensibility" (austen-sense.txt)

  N.B. The "f.B()+1" argument (already provided for you in the code) means that we lump together all the unseen n-grams as a single "unknown" token.

- text a: austen-emma.txt (Jane Austen's "Emma")

- text b: chesterton-ball.txt (G.K. Chesterton's "The Ball and Cross")
- `NgramModel`'s entropy function: `lm.entropy(...)`

Compute both the *total* cross-entropy and the *per-word* cross entropy of each text. (Separate function templates are provided.)

NOW UNCOMMENT THE TEST CODE AND CHECK YOUR RESULTS
Which text has higher total cross entropy? Which has higher per-word cross entropy? What can these comparisons tell us about the authorship of the two texts, or about their other properties?

6 Going further

6.1 Padding

Redo exercise 4 setting `pad_left` and `pad_right` to `True` both when initialising the n-gram model and when computing entropy. What difference does this make?