

# CIS 530 Fall 2009 HW 2

Instructor: Mitch Marcus

TA: Constantine Lignos

Released: October 23rd, 2009

Due: October 29th, 2009 at 4PM

## Overview

This assignment covers building an n-gram model and applying a simple smoothing technique to it. It will also serve an introduction to Python classes to those of you less familiar with the language. Because you will be duplicating some functionality that's included in NLTK, you obviously cannot use the modules of NLTK that provide n-gram models or probability estimators. **Your solution may only import the `FreqDist` and `ConditionalFreqDist` classes from NLTK and no other modules/classes.** To make sure that you stick to the allowed portions of NLTK, this is what your import statement for NLTK should look like:

```
from nltk import FreqDist, ConditionalFreqDist
```

The following line, which imports the whole NLTK module, should not appear in your file as it allows you to import other NLTK sub-modules:

```
import nltk
```

**You will be penalized if you import `nltk` as whole because it makes it much harder for us to verify that you only used the allowed portions of NLTK.**

The length of an example solution is given with each problem to give you a point of reference. You should not worry about matching this length, but if your solution is much longer you are probably overcomplicating things. Solution lengths include any helper functions used and include all comments and line breaks needed to make the code readable.

For all problems, **do not do any more preprocessing/filtering of data than is explicitly requested in the problem.** For example, when you are asked to examine the words of a corpus, you should look at all tokens, even non-alphabetic ones. Note that many of the problems will request that you lowercase tokens/characters before counting them; this is very common practice in NLP.

## Submitting your work

### Code Organization

Please submit all your code for the assignment in a single file called "hw2\_yourpennkey.py" where "yourpennkey" is your PennKey.

### Submitting files

To electronically submit homework, if you're not already working on Eniac you need to place the file containing your solution on your SEAS account storage. One way to do this is using an SFTP client such as FileZilla.

Then connect via ssh to seas.upenn.edu and use the turnin command to submit your file for grading:

```
% turnin -c cis530 -p hw2 <filename>
```

This should print out a confirmation message. You can run `turnin` multiple times before the deadline, but each time you run `turnin`, it overwrites your previous submission for that assignment. You can check that the homework submitted successfully:

```
% turnin -c cis530 -v
```

This will show a list of the file(s) you have submitted.

## 1 Probability Estimation

We're going to implement smoothing with classes that take a `FreqDist` in their constructor and use it to estimate the probabilities of events.

### 1.1 MLE (5 Points)

As a warm up, create a class `MLEProbDist` with the following instance methods<sup>1</sup>:

- `__init__(freq_dist)`: `freq_dist` is a `FreqDist` that will be used to provide estimates. Do any initialization you require in this function.
- `prob(event)`: Return the probability of `event` as given by the `FreqDist` passed in the constructor.

Don't think too hard on this one, this is a trivial class that will serve as a model for the smoothed version.

Example solution length: 8 lines

### 1.2 Smoothing (10 Points)

Create a class `AddGammaProbDist` with the following instance methods:

- `__init__(freq_dist, bins, gamma)`: `freq_dist` is a `FreqDist` that will be used to provide estimates. `bins` is the integer number of bins to be used in the smoothing. `gamma` is a positive float that is the amount added to the count for each event during smoothing. Do any initialization you require in this function.
- `prob(event)`: Return the smoothed probability of `event` based on the count given by the `FreqDist` passed in the constructor.

This smoothed estimator is similar to Laplace ("Add-one") smoothing, except that here we use `gamma` as the amount to add to the count of each event. As derived from the class slides, the probability estimate for an event  $x$  in this estimator is:

$$\frac{c(x) + \gamma}{N + \gamma B}$$

Where  $c(x)$  is the number of times  $x$  was observed in  $N$  samples,  $\gamma$  (`gamma`) is the smoothing factor, and  $B$  (`bins`) is the number of possible outcomes in our smoothed world. Basically, we are adding  $\gamma$  to the count of all possible events and saying that the total number of possible outcomes in this distribution is  $B$ .

Example solution length: 12 lines

---

<sup>1</sup>When specifying class methods, we'll leave off `self`, which all methods have as their first argument in their definition.

## 2 A Sliding Window (10 points)

Create a function `make_ngram_tuples(samples, n)` that returns a sequence of all the n-grams seen in the input, in order. The function returns a sequence of tuples where each tuple is of the form `(context, event)`. The context is `None` (the Python built-in value `None`, not the string “None”) in the case of  $n = 1$  (unigrams), and for larger values of  $n$  it is a tuple<sup>2</sup> of the preceding  $n - 1$  words for each sample.

Consider this usage example:

```
>>> samples = ['her', 'name', 'is', 'rio', 'and', 'she', 'dances', 'on', 'the', 'sand']
>>> make_ngram_tuples(samples, 1)
[(None, 'her'), (None, 'name'), (None, 'is'), (None, 'rio'), (None, 'and'), (None, 'she'),
 (None, 'dances'), (None, 'on'), (None, 'the'), (None, 'sand')]
>>> make_ngram_tuples(samples, 2)
[('her',), 'name'), (('name',), 'is'), (('is',), 'rio'), (('rio',), 'and'),
 ('and',), 'she'), (('she',), 'dances'), (('dances',), 'on'), (('on',), 'the'),
 (('the',), 'sand')]
>>> make_ngram_tuples(samples + samples, 2)
[('her',), 'name'), (('name',), 'is'), (('is',), 'rio'), (('rio',), 'and'),
 ('and',), 'she'), (('she',), 'dances'), (('dances',), 'on'), (('on',), 'the'),
 (('the',), 'sand'), (('sand',), 'her'), (('her',), 'name'), (('name',), 'is'),
 (('is',), 'rio'), (('rio',), 'and'), (('and',), 'she'), (('she',), 'dances'),
 (('dances',), 'on'), (('on',), 'the'), (('the',), 'sand')]
```

Note that in Python a tuple displayed as `(x,)` is a tuple consisting of one element, `x`. The comma is there simply to establish that this is a tuple, and not just parentheses around a value. The final usage example makes it clear that this function simply delivers all n-grams seen in order and does not remove any duplicates; this is a crucial behavior of this function.

Example solution length: 17 lines

## 3 N Steps to Happiness (30 Points)

For this problem you'll implement a basic N-Gram model with optional smoothing, using the functions you've created already as helpers.

Define a class `NGramModel` with the following instance methods:

- `def __init__(training_data, n, full_data=None, gamma=1.0)`: Train an n-gram model of order `n` using the supplied samples in `training_data`, and using `full_data` and `gamma` for smoothing. Note that the last two arguments are optional, but supplying `full_data` is what determines whether the model is smoothed. Supplying `gamma` without `full_data` is incorrect usage of the function, and you need not worry about it.
- `prob(context, event)`: Return the probability of `event` given that it is preceded by `context`. Whether this estimate is smoothed or not is determined by how the model was initialized.
- `items()`: Return all events of non-zero probability in the model in the form of a sequence of tuples of the form `(context, event, prob)` where `context` is a tuple of strings as defined for `make_ngram_tuples`. The example below makes this much clearer. For an unsmoothed model, this will return all events seen in training, while for a smoothed model this will return all events seen in the data set used for smoothing (which is a superset of the training data).

---

<sup>2</sup>Why are we using tuples? Tuples are hashable, and you'll find that the output of this function may be very handy for initializing a structure with `context` as a key.

- `generate(n, context)`: Generate a sentence of up to length `n` starting with the words given in `context`. If the model is a unigram model, the only allowed value of `context` is `None`<sup>3</sup>, and for higher order models `context` is a tuple of strings as defined for `make_ngram_tuples`. Use the Shannon/Miller/Selfridge method (see the lecture “N-Gram Word-Based Models of Syntax” for details) to randomly select the next word. It is possible to hit a dead-end while generating. Understanding why this happens is left as an exercise, but if this happens during generation, you can simply stop and return the sentence generated so far. `generate` may only be called on an unsmoothed model.

Consider this usage example:

```
>>> words = ['oh', 'rio', 'rio', 'dance']
>>> model = NGramModel(words, 2)
>>> model.prob(('rio',), 'rio')
0.5
>>> model.items()
[ (('oh',), 'rio', 1.0), (('rio',), 'rio', 0.5), (('rio',), 'dance', 0.5) ]
>>> model.generate(4, ('oh',))
['oh', 'rio', 'rio', 'dance']
>>> model.generate(4, ('oh',))
['oh', 'rio', 'rio', 'rio']
>>> model.generate(4, ('oh',))
['oh', 'rio', 'dance']
```

Note that in the last call to `generate`, we hit a “dead-end” while generating so we stopped generating before outputting `n` words.

Obviously, you will find the functions created for previous parts of the assignment very useful in implementing this class. Detailed requirements for the model follow:

1. To record events, use a `ConditionalFreqDist` (CFD) where the condition is the context for each n-gram.
2. To return the probability of a word in a given context, apply the MLE/AddGamma estimators to the `FreqDist` of the context. Remember, a `ConditionalFreqDist` is really a dictionary comprised of a `FreqDist` for each condition.
3. Smoothing depends on having a definition of what “all possible n-grams” are. For example, in a bigram model the trivial definition would be to take all  $n$  words seen in training and say that there are thus  $n^2$  possible bigrams. We’ll take a different approach. We’ll smooth our model by training on a subset of a corpus and then using the full corpus to help improve our estimates. Thus we compute the smoothed version of the model as follows:
  - (a) Train a CFD on `training_data`, as in the unsmoothed case.
  - (b) Count how many **unique** words appear in each context in `full_data`.
  - (c) Use the number of words we saw in each context as the number of bins for the AddGamma estimator for each context.
4. While the AddGamma estimator will help you smooth out the probabilities of words not seen in training within contexts you did see in training, applying it to the CFD generated from the training data will not give you probabilities for contexts you’ve never seen (after all, they are not in the CFD). Don’t forget to handle this case. Hint: What happens if you pass an empty `FreqDist` to your AddGamma estimator?

---

<sup>3</sup>As usual, you don’t need to defensively code around this. Just don’t worry about what happens if a non-None context is passed to a unigram model or a None context is passed to a higher-order model.

You'll note that this form of smoothing won't assign a non-zero probability to all sequences but will assign a non-zero probability to any sequence in `full_data`. The unsmoothed version only assigns non-zero probabilities to sequences seen in the training corpus. Also, our AddGamma estimator doesn't check that a word you pass it was actually seen in `full_data` in that context; all words unseen in a given context in training are given the same probability. That's okay, but this is why we don't use `generate` on the smoothed model—we don't formally track the possible words for each context, only the ones we saw in training.

Example solution length: 94 lines, about half of them in `generate`

## 4 Picking a Gamma (15 Points)

For any form of smoothing where you add counts you need to decide how large a count to add. As discussed in the Manning and Schutze text, Laplace (add-one) smoothing usually adds too much, but adding 0.5 seems to be a good value for many applications if the set of possible n-grams is taken to be all possible combinations of words in the corpus. We consider a much smaller universe of all possible n-grams, so we need a way to get an idea of what values of  $\gamma$  are reasonable.

Write a function `sweep_smoothing(full_words, train_words, test_words, n, min_smooth, max_smooth)`. It trains a single unsmoothed n-gram model over order `n` on `test_words` and multiple smoothed models of the same order from `train_words`, using `full_words` for smoothing and varying values of  $\gamma$ . It computes K-L divergence between the unsmoothed and smoothed models for different  $\gamma$  values.

The function computes the K-L divergence between models over the probability for each n-gram seen in `test_words`, taking its probability in the unsmoothed model trained on `test_words` as  $P(x)$  and its probability in the smoothed model trained on `train_words` as  $Q(x)$ . It uses base 2 logarithms. Unlike in the first homework assignment, you need not adopt a modified form of K-L Divergence<sup>4</sup>.

This function returns a sequence of tuples of the form `(gamma, divergence)` where `gamma` is the value of  $\gamma$  used in smoothing the model and `divergence` is the K-L divergence of that model from the unsmoothed model trained on `test_words`. The function calculates these values for values of  $\gamma$  of the form  $1.0/x$  for integer values of  $x$  between `min_smooth` and `max_smooth`, inclusive.

Consider this usage example, but note that **the values of K-L divergence will vary wildly depending on the model and corpora used and you shouldn't read into the fake values returned here**. If you wanted to compare bigram models using values for  $\gamma$  between 1 and 1/4, you would do the following:

```
>>> sweep_smoothing(full_words, train_words, test_words, 2, 1, 4)
[(1.0, 0.1234), (0.5, 0.1234), (0.33333333333333331, 0.1234), (0.25, 0.1234)]
```

If you want to add additional methods to the `NGramModel` class to support this function, feel free to do so.

To test this function, one would want to use a moderately-sized corpus. But to make the strain on the shared computers a little lighter and your development time shorter, consider this corpus, which is the chorus of the Duran Duran song "Rio":

---

<sup>4</sup>Why? The danger in computing K-L divergence comes from situations where  $P(x)$  is non-zero and  $Q(x)$  is zero. Can that ever happen here?

```

train_words = [word.lower() for word in \
""Her name is Rio and she dances on the sand
Just like that river twists across a dusty land
And when she shines she really shows you all she can
Oh Rio Rio dance across the Rio Grande"".split()]
test_words = [word.lower() for word in \
""Her name is Rio she don't need to understand
I might find her if I'm looking like I can
Oh Rio Rio hear them shout across the land
From mountains in the North down to the Rio Grande"".split()]
full_words = train_words + test_words

```

Let this also be an example of how easy it is to find toy data to test out your work. In future assignments, you will need to do this on your own. This test set will also be handy and testing out other problems in this assignment.

Example solution length: 23 lines, including a helper for K-L divergence

## 5 Debriefing

Include the answers to these questions in comments at the end of your file.

1. Approximately how many hours did you spend on this assignment?
2. Would you rate it as easy, moderate, or difficult?
3. How deeply do you feel you understand the material it covers (0%-100%)?
4. Any other comments?

These questions are intended to help us calibrate the homework assignments. Your answers will not affect your grade, and unless you request it we will not contact you about them.

## Revision History

10/23/2009	Initial version
10/23/2009	Fixed description of problem 1 to make it clear that both functions take a <code>FreqDist</code> , not a <code>ConditionalFreqDist</code> . Corrected the class name for problem 3 to <code>NGramModel</code> . Made the description of problem 4 clearer. Formatting and typographic fixes.