CS447: Natural Language Processing Fall 2019

Homework 1

Handed Out: 09/13/2019 *Due: 11:59pm, 09/30/2019*

Please follow the submission guidelines given below and for any question ask on Piazza. Please document your code where necessary. Note we will use **Python 3** for this homework.

Getting started

All files that are necessary to do the assignment are contained in a tarball which you can get from:

`http://courses.engr.illinois.edu/cs447/HW/HW1/HW1.tar.gz`

You need to unpack this tarball (`tar -zxvf HW1.tar.gz`) to get a directory that contains the code and data you will need for this homework.

Part 1: Language models and smoothing (6 points)

1.1 Goal

The first part of the assignment requires you to train some simple n-gram language models on a corpus of movie reviews and to test them on two smaller corpora: a collection of positive reviews, and one of negative reviews.

1.2 Data

The corpora are stored as text files, where each line is one sentence from a review:

1. `train.txt`: contains the training corpus of movie reviews (30k sentences)
2. `pos_test.txt`: contains the test corpus of positive reviews (1k sentences)
3. `neg_test.txt`: contains the test corpus of negative reviews (1k sentences)

Tokens (this includes words and punctuation marks, which you should treat like regular tokens) are separated by whitespaces.

1.3 Provided Code

To get you started, we have provided the module `hw1_lm.py`. This file contains code for reading in the corpora (as in HW0) and maintaining a basic probability distribution (`UnigramDist`).

We have also defined a parent `LanguageModel` class, along with subclass definitions for the five language models we'd like you to implement. Your main task (described in section 1.4) is to implement the high-level methods of these models, specifically: generating sentences, calculating the probability of sentences, and computing the perplexity of a corpus.

Internally, we recommend that you treat each sentence as a list of tokens; this is the same representation returned by our input method.

1.4 What you need to implement

1.4.1 Preprocessing

To make your models more robust, it is necessary to perform some basic preprocessing on the corpora.

Sentence markers For all corpora, each sentence must be surrounded by a start of sentence and end of sentence marker (`<s> ... </s>`). These markers will allow your models to generate sentences that have realistic beginnings and endings, if you train your model properly.

Handling unknown words In order to deal with unknown words in the test corpora, all words that appear only once in the training corpus must be replaced with a special token for unknown words (e.g. `'UNK'`) before estimating your models. When unknown words are encountered in the test corpora, they should be treated as that special token instead.

These preprocessing steps have been provided for you in the assignment code.

1.4.2 The `LanguageModel` classes

In order to compare the effects of using different-order n-grams and smoothing, we require you to implement the following five models:

1. `UnigramModel`: an unsmoothed unigram model, with probability distribution $\hat{P}(w)$
2. `SmoothedUnigramModel`: a unigram model smoothed using Laplace (add-one) smoothing, with probability distribution $P_L(w)$
3. `BigramModel`: an unsmoothed bigram model, with probability distribution $\hat{P}(w'|w)$

Laplace Smoothing For the `SmoothedUnigramModel` we want you to use Laplace smoothing, also known as add-one smoothing, on the unigram model $\hat{P}(w)$. Remember that in Laplace smoothing we increase the counts of all events by one and renormalize. This takes probability mass away from seen events and reserves it for unseen events (see Lecture 4, slide 17)

In order to smooth your unigram model, you will need the number of words in the corpus, N, and the number of word types, S. The distinction between these is meaningful: N indicates the number of word instances, where S refers to the size of our vocabulary. The sentence "the cat saw the dog" has four word types (*the*, *cat*, *saw*, *dog*), but five word tokens (the, cat, saw, the, dog). The token "the" appears twice in the sentence, but they share the same type *the*.

If $c(w)$ is the frequency of w in the training data, you can compute $P_L(w)$ as follows:

$$P_L(w) = \frac{c(w) + 1}{N + S}$$

1.4.3 Generating sentences

For each of the four language models, you need to implement the following methods:

> `generateSentence(self)`: returns a sentence `sent` that is generated by the language model. `sent` is a list of the form $[\texttt{<s>}, w_1, ..., w_n, \texttt{</s>}]$, where w_1 to w_n are words in your vocabulary (including `UNK`, but excluding `<s>` and `</s>`). You can assume that `<s>` starts each sentence (with probability 1). The following words $(w_1, ..., w_n, \texttt{</s>})$ are generated according to your language model's distribution. The number of words (n) is not fixed. Instead, you stop generating a sentence as soon as you generate the end of sentence symbol `</s>`.
>
> `getSentenceProbability(self, sen)`: returns the probability of the sentence `sen` (which is again a list of the form $[\texttt{<s>}, w_1, ..., w_n, \texttt{</s>}]$) according to the model.

Please use the provided `generateSentencesToFile` method and your unigram and bigram language models to generate 20 sentences (saved as `unigram_output.txt`, `smooth_unigram_output.txt` and `bigram_output.txt`).

Implementation hint In order to avoid underflow, your implementation may have to use log-probabilities internally. That is, once you have computed each (smoothed) probability from your relative frequency estimates (counts), you need to convert it to its logarithm (use `math.log(p)` for natural logs or `math.log(p, 2)` for base-2 log, but make sure you always use the same base).

1.4.4 Computing the perplexity of the test corpora

You need to compute the perplexity (normalized inverse log probability) of the two test corpora according to all five of your models (unsmoothed unigram, smoothed unigram, unsmoothed bigram, smoothed bigram ad and smoothed bigram kn). Implement the following method to evaluate a whole corpus:

`getCorpusPerplexity(self, corpus)`: given a corpus `corpus`, calculate and return its perplexity

For a `corpus` W with N words, Jurafsky and Martin tells you to compute perplexity as follows:

$$Perplexity(W) = \sqrt[N]{\prod_{i=1}^{N} \frac{1}{P(w_i|w_{i-1})}}$$

Since this is prone to underflow issues, you may be better off computing it using logarithms:

$$Perplexity(W) = \exp\left(-\frac{1}{N}\sum_{i=1}^{N} \log P(w_i|w_{i-1})\right)$$

Evaluate the models on the test corpora. Do you see a difference between the two test domains?

1.5 What you will be graded on

You will be graded based on the files you submit and an autograder would run on your models. Each of your models is worth 2 points. Keep in mind that the auto-grader must be able to compile and run your submitted code **or you will receive 0 points**.

To help you identify errors, we have included a test script `hw1_lm_test.py`. This script will use the classes you implement in `hw1_lm.py` and test their methods against a simple corpus, `test.txt`, which we have also included.

1.6 What to submit

See the end of this document for full submission guidelines, but the files you must submit for this portion are:

1. `hw1_lm.py`: your completed Python module for language models (see all of section 1.4)

Additionally, we provide you some questions to think about. **You do not need to answer these questions. These questions are helpful while studying for your midterms.**

1. When generating sentences with the unigram model, what controls the length of the generated sentences? How does this differ from the sentences produced by the bigram models?
2. Consider the probability of the generated sentences according to your models. Do your models assign drastically different probabilities to the different sets of sentences? Why do you think that is?
3. Generate additional sentences using your bigram and smoothed bigram models. In your opinion, which model produces better / more realistic sentences?
4. For each of the four models, which test corpus has a higher perplexity? Why? Make sure to include the perplexity values in the answer.

Part 2: Finite-State Transducers (4 points)

2.1 Goal

Your task is to implement a finite-state transducer (or FST; see Lecture 2, slide 38) which transduces the infinitive form of verbs to their correct *-ing* form. You will be graded according to how many verbs your transducer handles correctly.

2.2 Data

The input to your program is a text file with a single (infinitive) verb per line. For each verb, your program should compute the correct translation (e.g. `walk ==> walking`) and print it as a line in the output file.

1. `360verbs.txt`: sample input file containing a list of 360 infinitive verbs, one verb per line.
2. `360verbsCorrect.txt`: sample output file containing the correct translation (infinitive to *-ing* form) for each of the 360 verbs in order, one translation per line.

2.3 Provided Code

To help you with your task, we've provided `fst.py`, a module for constructing and evaluating FSTs. You shouldn't modify any of the code in this file. Below, we describe the most useful methods provided by this module:

> `FST(self, initialStateName)`: instantiate an FST with an initial (non-accepting) state named `initialStateName`
>
> `addState(self, name, isFinal)`: add a state named `name` to the FST; by default, `isFinal`=false and so the state is not an accepting state.
>
> `addTransition(self, inStateName, inString, outString, outStateName)`: adds a transition between state `inStateName` and state `outStateName`, where both of these states already exist in the FST. The FST can traverse this transition after reading `inString`[1], and outputs `outString` when it does so.
>
> `addSetTransition(self, inStateName, inStringSet, outStateName)`: adds a transition between state `inStateName` and state `outStateName` for each character in `inStringSet`. For each transition, the FST outputs the same character it reads.

You will need to implement your solution by modifying the file `hw1_fst.py`. This file uses the function `buildFST()` to define a rudimentary (and buggy) FST for translating verbs. Your task is to modify this function (using the methods in `fst.py`) to produce a better FST. You are free to define your own character sets (we've started out with some useful ones at the beginning of the file) and any helper methods you require. The main method evaluates the sample FST on an input file. When implementing your FST, you can save your results to another file (e.g. `360verbsGuess.txt`) and run `diff` against `360verbsCorrect.txt` to see how well you're doing:

```
python hw1_fst.py 360verbs.txt > 360verbsGuess.txt

diff -U 0 360verbsCorrect.txt 360verbsGuess.txt | grep ^@ | wc -l
```

This will print the number of incorrect translations.

```
diff -U 0 360verbsCorrect.txt 360verbsGuess.txt | grep ^@
```

This command should print the commands which you get wrong.

[1] `inString` can be at most one character long

2.4 What you need to implement

Your task is to fix the implementation of `buildFST()` by replacing the sample FST with a better FST for translating infinitive verbs to their correct *-ing* form. Your FST can be non-deterministic (that is, a state may have multiple transitions for the same input character), but each accepted string should only have one analysis.[2] It will accept a string if it reaches a final state after having read its last character. If it accepts a string, it will output its translation. If it does not accept a string, it will output FAIL. All verb stems are at least three letters long. In many cases, the *-ing* form just consists of *stem* + *ing*:

```
walk ==> walking     fly  ==>  flying
```

But there are a number of exceptions. In particular, in order to correctly translate the 360 verbs we've provided, your FST must implement the following rules:

1. **Dropping the final *-e*:** Drop the final *-e* **if and only if** it is preceded by a consonant or by the letter *u*:

   ```
   ride  ==> riding        make  ==> making        see ==> seeing
   argue ==> arguing       seize ==> seizing
   ```

2. **Double the final consonant:** Double a final single *-n, -p, -t, -r* **if and only if** it is preceded by a single vowel[3]:

   ```
   stop  ==> stopping      admit ==> admitting        stoop ==> stooping
   occur ==> occurring     set ==> setting            leap  ==> leaping
   halt  ==> halting       bar ==> barring
   ```

   Exceptions to this rule are verbs ending in *-er, -en*:[4]:

   ```
   gather ==> gathering    happen ==> happening
   ```

3. **Change final *-ie* to *-y*** :

   ```
   die  ==> dying
   ```

Notes: When you define a transition, the input strings can only be zero ("") or one character long, and your FST can't have more than 30 states (you shouldn't need that many).

2.5 What to submit

For this portion, you only need to submit one file:

1. `hw1_fst.py`: your completed Python module for translating verb forms using FSTs (see section 2.4)

We will use our own copy of the `fst` module and input files to test your program during grading.

2.6 What you will be graded on

You will be graded according to the number of verbs in `360verbs.txt` that your FST correctly translates (100% is 4 points; the provided FST starts at 78%). You need to make sure that the function definition of `buildFST()` remains the same, so that we can check your results and verify that your FST adheres to the specification.

[2] If your FST accepts a string along multiple paths, all of the outputs will all be printed out and the translation will be marked incorrect: e.g. `appeal --> appealingappealling`

[3] This is a simplified version of the actual spelling rule. Our data set does not contain verbs like *pardon*, where this rules does not apply. It also does not contain verbs that end in other letters like *stab, slam*, where this rule also applies.

[4] We are assuming American English spelling for *labeling*, etc.

Setting up Gradescope

1. You will need to make sure you are enrolled in the course and are able to view **CS447** on gradescope for **Fall 2019**. For that click on **Add a Course** on your gradescope dashboard. Your 6-character code is **98DYG8**. If you have any issues please let the TAs know. **Make sure you use your illinois.edu email**
2. Once you can, you should see **HW1** on the portal. Once you click on that you will have an option to **Upload Submission**.
3. Click on the **Upload Submission** button and upload the files asked for below in **Submission guidelines**.

Submission guidelines

You need to submit your code to Gradescope for autograding. Do not include the corpora themselves, but do include the following files:

1. `hw1_lm.py`: your completed Python module for language models (see all of section 1.4)
2. `hw1_fst.py`: your completed Python module for translating verb forms using FSTs (see section 2.4)

Make sure you only implement the functions provided to you in these files. Do not change the names of the functions or any things which might lead to the Gradescope autograder not being able to compile your code. Such compilation issues (if your fault) might lead to **a 0 on the whole assignment.**