CS159 - Language Modeling Lab

Today, we're going to be playing with a publicly available language modeling toolkit. The two most common ones are:

- SRI toolkit: `http://www-speech.sri.com/projects/srilm/`
- CMU toolkit: `http://www.speech.cs.cmu.edu/SLM_info.html`

1 Getting started

We're going to be using the SRI toolkit. The toolkit comes with many, many different scripts, etc. but the two main ones we will use are:

- `ngram-count`: This is a program that runs through a file, aggregates counts and outputs a trained language model
- `ngram`: This program takes the output from ngram-count and applies the language model either for generating random sentences or for asking perplexity questions.

I have precompiled versions of these for Mac OS. If you have a Mac laptop, you can work on that or work on one of the lab machines.

Download the following zip file and put it somewhere where you can work:

`https://cs.pomona.edu/classes/cs159/lectures/lm_lab.zip`

2 Word modeling

To get started, let's try out some character language models where our n-grams are over characters instead of words. In the data directory there are two data files that contain one word per line and where the characters have been split apart by spaces (to trick the lm toolkit into treating them as words)

- `char.data` contains all the unique words from a smallish data set

- `char_full.data` contains all of the words *with repetition* from the same data set

Let's start and train a basic model. Open up "Terminal" and then `cd` into your `lm_lab` directory and type:

```
./ngram-count -text data/char.data -order 3 -write char.count -lm char.lm
```

Here's a quick explanation of the parameters:

`-text` tells it which file to read the text data from

`-order 3` tells it to train a language model using trigrams (3-grams)

`-write` tells it to write the raw frequency counts to the file. This isn't necessary, but can be interesting to look at sometimes. Also, for larger data sets, you can pass this to `ngram-count` again when calculating smoothing, which can be faster than reading the original text file again.

`-lm` this file is the file we'll feed into `ngram` and is in a file format called ARPA format. It has the logprobs of the all of the n-grams (from 1 to `order`) and also backoff parameters depending on which model is used.

(Note, you'll get a few warnings when you run this, which you can ignore.)

Take a look in both of the generated files (they're both text files).

Now that we have a trained model, we can try a few things.

Generating random words

We can sample randomly from the language model and generate random "words". Type:

```
./ngram -lm char.lm -gen 10
```

`-lm` tells it which language model to use

`-gen 10` tells it to generate 10 random words

Questions/experiments

Write down a few notes/observations for each of the questions below and we'll discuss at the end of class

1. How do the words look? Any interesting ones?
2. Retrain the language model with different orders (e.g. unigrams, bigrams, 4-grams, etc.) and look at the output. Can you see a difference?

3. Right now, we're using any letter we've seen in the training data in our words/vocabulary. We can constrain this with the `-vocab` flag for `ngram-count` which takes a file with one word per line. Make a file with just the letters a through z in it (one per line) and retrain the model. Do the results improve? How do things change?

4. Try some of the same things with the `char_full.data` file. How do the results differ? If you were asked to model words in English, which would you use?

Perplexity

We can also measure the perplexity (average log prob) on data to try and see how well our model is doing. To get started, let's just enter some by hand:

```
./ngram -lm char.lm -debug 1 -ppl -
```

- `-debug` specifies the debugging level. 1 is a good start but try higher levels and you'll get more information

- `-ppl -` the -ppl flag tells the program to calculate the perplexity on the specified file. By specifying '-' we're telling it to read from STDIN, so you can just type in words

Try typing in some known words and compare the perplexities to unknown words (you'll have to type spaces in between each character). `ppl` is the perplexity and `ppl1` is the perplexity without the end of sentence token.

How well does it work? Try adjusting the debugging level. What extra information do you get?

Once you've tried a few things, lets see if we can compare some models. Take the `char.data` files and split it into two files say 90/10. Train on the 90% file and then calculate the perplexity of the test file. When you calculate the perplexity omit the `debug` flag (or it will print out lot of information you probably don't want) and instead of '-' put your filename.

Make sure that you pick a random split from this file. One way to do this is to:

- Randomly shuffle the first. There are many ways to do this. One way that will work on the command-line is:

  ```
  cat data/char.data | perl -MList::Util=shuffle -e 'print shuffle(<STDIN>);' > char.data.randomized
  ```

  (You have have to type this since for some reason it has trouble being copy and pasted... sorry!)

- Once you've done this, you can take the first 90% of the file as training and the last 10% of the file as testing.

  - Use `wc -l` to figure out how many lines are in the file.
  - Use `head` to grab the first 90% (calculate how many this is and then specify this number using the `-n` flag).
  - Use `tail` to grab the last 10%.

Questions/experiments

1. How does the order of the n-gram model affect performance? Did this seem to correlate with your qualitative experiments from looking at the generated output?
2. Pick an n-gram order (say 4 or 5) and then vary the amount of training data you give it. Make sure to specify the `-vocab` flag when comparing sizes or it won't be a valid comparison. Do you see a trend? Do you think you have enough data to get a reasonable result?

3 Language modeling: what is the best model?

Now lets take a look at the real language modeling task. In the data directory I've also included a file called `train.sentences` which has 100,000 sentences.

The language modeling toolkit has a lot of different parameters you can try. The full documentation can be found online at:

`http://www.speech.sri.com/projects/srilm/manpages/`

Split off some of the sentences from your data and use them for development testing. Run some experiments (some ideas below) examining the performance impact of different features. To be consistent, make sure, like above, that you use the same vocabulary between experiments. I've provided a file `train.vocab` that contains all of the words in the training sentences. This should get you started, but you may want to experiment with other versions.

Investigate some of the following:

1. How does the order affect the model performance? How does this compare to your results from the word model?
2. How does the size of the training data affect the model performance? How does this compare to your results from the word model?
3. How does the smoothing affect the performance? Some smoothing methods to try (all specified for `ngram-count`):
   - `-cdiscount [num]`: absolute discounting with discount `[num]`
   - `-wbdiscount`: use Witten-Bell discounting
   - `-kndiscount`: use modified Kneser-Ney discounting
   - `-gtmax [num]`: n-grams that occur less than [num] times will be adjusted using the Good-Turing estimate. If we're using a closed vocabulary, you should also specify `-gtmin [num2]` which effectively throws all words that occur < [num2] times.
   - `-interpolate`: Read the documentation :)