Lab 3: Language Modeling in Racket

Due October 7th at 10pm

A language model is a model of sentence probability. One common kind of language model is an *n-gram* model, where the probability of a sentence is estimated as the joint probability of each n-length sequence of words. For instance, in a unigram model, the probability of the sentence is simply the probability of each of the words, multiplied together.

1. Unigram model example:
   p(“A cat meows") $\approx$ p(“A") p(“cat") p(“meows")

In a bigram model, the probability of each word in the sentence is estimated by the probability of it occurring after the word that precedes it.

2. Bigram model example:
   p(<s> A cat meows <s>) $\approx$ p(“A"|<s>) p(“cat"|“A") p(“meows" | “cat") p(<s> | “meows")

Given some text, we can estimate the probability of each bigram by counting how many times it occurs relative to every other bigram that begins with the same word. This is called a *maximum likelihood estimate*.

3. Maximum likelihood estimate for the bigram "cat meows":
   $\frac{count(\text{“cat” “meows”})}{\sum_{w \in vocab} count(\text{“cat”} w)}$

In a bigram model, we can simplify this equation further. The total count of bigrams that begin with the word “cat" is equally to the total number of times that the word “cat" appears in the training data.

4. Maximum likelihood estimate for the bigram "cat meows" (simplified):
   $\frac{count(\text{“cat” “meows”})}{count(\text{“cat”})}$

Thus, in order to estimate the probability of each bigram, you’ll need to count two things: the total number of occurrences for each word, and the total number of occurrences of each bigram.

In this lab, you will write a bigram language model in Racket. Your model will estimate the probability of sentences in English, and it will also be capable of generating likely English sentences.

This chapter in Jurafsky & Martin’s Speech and Language Processing textbook is a good reference on n-gram language models: https://lagunita.stanford.edu/c4x/Engineering/CS-224N/asset/slp4.pdf.

Data

I have provided you with two data files. The first contains the text of *Alice in Wonderland*.[1] I recommend reading in only the first 10,000 characters; otherwise your program will take a long time to run.

[1] Made available by Project Gutenberg

The second file is a smaller test file containing the first three paragraphs of *The Hobbit*. I recommend testing your code on the smaller file as you work; I have given sample output for the smaller data file for many of the functions that you will write.

You should make sure that all of your code also works on the larger *Alice in Wonderland* file.

1 Reading in the data

The first step is to read in data from a text file. Use the read-string function to read in 10,000 characters from the text file, and store the result in a variable.

2 Preprocessing the data

In order to train your model, you will have to write some text processing functions in order to get your training data into the right format.

2.1 Replace

Before text data can be fed into a language model, we often want to clean up the text by removing punctuation marks. This is because we might want to include both 'Alice:' and 'Alice,' in our counts for how frequently the word 'Alice' occurs.

Write a function called **replace** has three arguments: a string, a second string called before, and a third string called after. The function should go through the string and replace any occurrences of before with after.

An example of how replace should work is shown below:

```
> (replace “pigs are pink" “i" “u")
“pugs are punk"
```

Once your replace function is working, write a function called **strip-punctuation** that uses replace to make the following substitutions:

- Replace newlines with the empty string
- Replace commas with the empty string
- Replace colons with the empty string

2.2 Add sentence tags

Since we are interested in modeling sentence probabilities, we need some way of identifying sentences. We are going to assume that sentences boundaries occur whenever one of three punctuation marks is present: a period, a question mark, or an exclamation mark.

We will add an end-of-sentence boundary tag to mark where the end of each sentence is. This is useful too in order to properly calculate the bigram probability of the first word in the sentence: we assume that it is preceded by the special end-of-sentence tag, and calculate its probability accordingly.

Our end-of-sentence tag will be "<s>".

Using the replace function that you wrote, write a function called **add-tags** which takes a string and adds an end-of-sentence tag after every period, exclamation point, and question mark. **Make sure to add a space between the punctuation mark and the end-of-sentence tag**.

If you apply the add-tags function to the Hobbit text (having stripped out the punctuation specified in Question 2.1), the beginning of the text will be as follows:

“In a hole in the ground there lived a hobbit. <s> Not a nasty dirty wet hole filled with the ends of worms and an oozy smell nor yet a dry bare sandy hole with nothing in it to sit down on or to eat; it was a hobbit-hole and that means comfort. <s> It had ...

2.3 Splitting into words

In this section, you will write a wrapper function called **get-words**.

I have given you a function called **split**, which is the inverse of the join function that you wrote in Lab 1 and Lab 2. Split takes two strings named *text* and *sep* and returns a list of strings. Every time that sep appears in text, the preceding portion of text is added to the return list.

An example is shown below:

>(split “hello-world" "a") ‘(“hello" “world")

One common application of split is to take a sentence and split it into a list of words by providing a single space as the separator string.

Write a function called **get-words** that takes a string and splits it into a list of words.

If you apply your get-words function to the tagged text you created in the previous question, the first part of the result will be as follows:

‘(“In" “a" “hole" “in" “the" “ground" “there" “lived" “a" “hobbit." “<s>" “Not" ...

3 Creating bigrams

Now that we have a list of words, with sentence boundaries, it’s time to turn them into bigrams.

3.1 Zip

First, let’s define a useful helper function when working with lists. Write a function called **zip** which takes two lists as arguments, and returns a list of lists, where each nested list at index i contains the ith item of list 1 and the ith item of list 2.

For instance:

>(zip ‘(1 2 3) ‘(4 5 6)) ‘((1 4) (2 5) (3 6))

If the lists are different sizes, you can truncate the longer list to be the same length as the shorter one (return as soon as either list is empty).

3.2 Make bigrams

Now, use your zip function to write a function that takes a list of words and returns a list of bigrams. Call this **make-bigrams**.

Hint: you can add an end-of-sentence token to the beginning of your list so that the first word of the first bigram is “<s>".

If you apply your make-bigrams function to the list of words we generated from the Hobbit text, the first few bigrams should be as follows:

‘((“<s>" “In") (“In" “a") (“a" “hole") (“hole" “in") (“in" “the") (“the" “ground") ...

4 Calculating bigram probabilities

4.1 Find unique bigrams

In this step, you’ll write functions that produce two lists: one of unique bigrams, and the other, of the first word in each of the unique bigrams.

Can use this built-in function

First, write a function called **get-unique-bigrams**. This function will simply call the **remove-duplicates** method on your list of bigrams to get a list of unique bigrams.

Next, write a function called **get-matching-unigrams**. This function should map (lambda (x) (first x)) over the list of unique bigrams in order to return a list containing the first word of each bigram.

You now have a list of unique bigrams, and a list of the first word in each of those bigrams.

4.2 Get n-gram counts

In this step, you will write a function to get n-gram counts for a list of unique n-grams. Your solution should generalize to different values of n.

The input to your **get-ngram-counts** function will be two lists: a list of unique n-grams, and a list of all of the n-grams in the text. For instance, in the unigram case, your input will be a list of unique unigrams and a list of all of the words in the text.

Your function will produce a list with one item for each item in the unique n-gram list. For each unique n-gram, your function will count how many times it appears in the list of all n-grams.

For instance, if the unique list is ’(“cat" “mat" “sat" “on" “the") and the all-n-grams list is ’(“the" “cat" “sat" “on" “the" “mat" “on" “the" “mat" “sat" “the" “cat"), get-ngram-counts would work as shown below:

```
> (define unique (list “cat" “mat" “sat" “on" “the"))
> (define all (list “the" “cat" “sat" “on" “the" “mat" “on" “the" “mat" “sat" “the" “cat"))
> (get-ngram-counts unique all)
```

```
‘(2 2 2 2 4)
```

Once you have written your get-n-gram-counts function, use it to get the unigram counts and the bigram counts for your text.

For the Hobbit text, the beginning of the unigram counts will look like:

‘(11 1 15 3 5 17 ...

You can verify these counts by looking at the original text. For instance, the fourth unigram in our list is “hole", which occurs 3 times in the original text (plus one time where it is followed by a hyphen, which we didn’t remove).

The beginning of the bigram counts for the Hobbit will look like:

‘(1 1 1 1 4 1 1 ...

(Bigrams re-occur much less often. The bigram ‘in the’ is the first bigram in our list that occurs more than once.)

4.3 Estimate bigram probabilities

Now that you have a list of bigram counts, you’re ready to calculate bigram probabilities. Write a function called **calc-bigram-prob** which takes a list of bigram counts and list of unigram counts, and returns a list of bigram probabilities (using the formula given in the beginning of this assignment, repeated below).

5. Maximum likelihood estimate for the bigram "cat meows" (simplified): $\frac{count(\text{“}cat\text{”}\ \text{“}meows\text{”})}{count(\text{“}cat\text{”})}$

Remember: you can rely on the fact that the list of unigram counts is in the same order as the list of bigram counts. That is, our unigram counts were calculated given the list of unigrams that correspond to the first word of each bigram in the unique bigram list.

If you use your calc-bigram-prob function on your Hobbit data, the result should start:

‘(1/11 1 1/15 1/3 4/5 ...

5 Generating likely sentences

We now have all the components of our language model; we just have to put them together!

5.1 Store bigrams and their probabilities

I have provided you with a function called **store-probabilities**. This function takes a list of bigrams and a list of their probabilities, and stores them in a hash table.

This will allow you to look up the probability of a given bigram.

Create a bigram hash table using the store-probabilities function, your list of unique bigrams, and your list of their probabilities.

You can look up a given bigram by calling the function **hash-ref** as shown below:

```
> (define bigram-table (store-probabilities bigrams probs))
> (hash-ref bigram-table ‘(“hobbit" “was"))
1/2
```
Should be a list of (1/2)

As you can see, the first argument to hash-ref is your hash table; the second argument is your bigram, which you provide in the form of a list.

5.2 Get relevant bigrams

Write a function that takes a word and a list of unique bigrams and returns a list of all bigrams that begin with that word. Call this function **get-relevant-bigrams**.

For instance, when you call get-relevant-bigrams with “hobbit", the following list is returned:

```
‘((“hobbit" “was")(“hobbit" “bedrooms") (“hobbit" “and"))
```

5.3 Get most likely next word

The final step is to write a function that returns the most likely next word and its probability, given the preceding word. Call this function **get-likely-next-word**. Its arguments will be a word, a list of unique bigrams, and a hash table.

This function should find all bigrams that begin with the word, and then return the bigram from that list that has the highest probability along with its probability. The return type should be a list whose first item is the probability and whose second item is a list representing the bigram.

For instance, when given the input word "tunnel", the function returns a list containing the bigram “tunnel" “a" along with its probability, 1/3:

```
>(get-likely-next-word “tunnel" unique-bigrams table)
‘(1/3 (“tunnel" “a"))
```

5.4 Generate sentences

Size of sentence: 15 words

Once you have your get-likely-next-word function written, you can run the **generate-sentence** function that I have provided. Its arguments are a word, a list of unique bigrams, a hash table, and a maximum sentence length.

The function generates the most likely sentence that starts with the specified word according to the language model. Because the language model can predict infinitely long sentences, it curtails the sentence at the specified maximum length and returns.

For instance, given the input word “hobbit" and a maximum length of 20, the model generates:

“hobbit was a very comfortable tunnel a very comfortable tunnel a very comfortable tunnel a very comfortable tunnel a very".

According to your language model, what is the most likely sentence generated from the *Alice in Wonderland* data when the first word is the end-of-sentence tag ("<s>")?

6 Extra credit: Estimating the probability of sentences

A language model can also be used to estimate the likelihood of a given sentence. Write a function that takes in a sentence and estimates its probability according to the language model. Call this function **estimate-sent-prob**.