LING/COMP 445, LING 645
Problem Set 3

Name: Mathieu-Joseph Magri , **McGill ID**: 260928498

Due before 8:35 AM on Thursday, February 16, 2023

Please enter your name and McGill ID above. There are several types of questions below.

- For questions involving answers in English or mathematics or a combination of the two, put your answers to the question in an answer box like in the example below. You can find more information about LaTeX here https://www.latex-project.org/.
- For programming questions, please put your answers into a file called `ps3-lastname-firstname.clj`. Be careful to follow the instructions exactly and be sure that all of your function definitions use the precise names, number of inputs and input types, and output types as requested in each question.

  For the code portion of the assignment, **it is crucial to submit a standalone file that runs**. Before you submit `ps3-lastname-firstname.clj`, make sure that your code executes correctly without any errors when run at the command line by typing `clojure ps3-lastname-firtname.clj` at a terminal prompt. We cannot grade any code that does not run correctly as a standalone file, and if the preceding command produces an error, the code portion of the assignment will receive a 0.

  To do the computational problems, we recommend that you install Clojure on your local machine and write and debug the answers to each problem in a local copy of `ps3-lastname-firstname.clj`. You can find information about installing and using Clojure here https://clojure.org/.

  A template Clojure file will be provided with the helper functions described below.

Once you have entered your answers, please compile your copy of this LaTeX into a PDF and submit

(i) the compiled PDF renamed to `ps3-lastname-firstname.pdf`
(ii) the raw LaTeX file renamed to `ps3-lastname-firstname.tex` and
(iii) your `ps3-lastname-firstname.clj`

to the Problem Set 3 folder under 'Assignments' on MyCourses.

Example Problem: This is an example question using some fake math like this $L = \sum_0^\infty \mathcal{G}\delta_x$.

Example Answer: Put your answer in the box provided, like this:

Example answer is $L = \sum_0^\infty \mathcal{G}\delta_x$.

Problem 1: In these exercises, we are going to be processing some natural linguistic data, the first paragraph of Moby Dick. We will first write some procedures that help us to manipulate this corpus. We will then start analyzing this data using some probabilistic models.

We'll start by defining the variable `moby-word-tokens`, the tokens in our corpus as a Clojure list. This variable is defined for you already in the provided template Clojure file.

```
(def moby-word-tokens '(CALL me Ishmael . Some years ago never mind how long
  precisely having little or no money in my purse , and nothing particular to
  interest me on shore , I thought I would sail about a little and see the
  watery part of the world . It is a way I have of driving off the spleen ,
  and regulating the circulation . Whenever I find myself growing grim about
  the mouth whenever it is a damp , drizzly November in my soul whenever I
  find myself involuntarily pausing before coffin warehouses , and bringing up
  the rear of every funeral I meet and especially whenever my hypos get such
  an upper hand of me , that it requires a strong moral principle to prevent
  me from deliberately stepping into the street , and methodically knocking
  people's hats off then , I account it high time to get to sea as soon as I
  can . This is my substitute for pistol and ball . With a philosophical
  flourish Cato throws himself upon his sword I quietly take to the ship .
  There is nothing surprising in this . If they but knew it , almost all men
  in their degree , some time or other , cherish very nearly the same feelings
  toward the ocean with me .))
```

In the template Clojure file, we have also defined the function `member-of-list?`, with the following code.

```
(defn member-of-list? [w l]
  (if (empty? l)
    false
    (if (= w (first l))
      true
      (member-of-list? w (rest l)))))
```

It has two arguments, `w` and `l`, and returns `true` if `w` is a member of the list `l`, and `false` otherwise. For example, `(member-of-list? 'a '(a ship is))`, will return `true`, and `(member-of-list? 'the '(a ship is))` will return `false`.

The template Clojure file contains a skeleton for the function `get-vocabulary`, which you must implement. This function takes two arguments, `word-tokens` and `vocab`, and it should return a list of all unique words occurring in `word-tokens`. For example, if `word-tokens` is `'(the ship is the ship)`, then get-vocabulary should return `'(the ship is)`. Implement this function by filling in the missing parts of this provided code.

When you call `(get-vocabulary moby-word-tokens '())`, you will get back a list of all of the unique words occurring in `moby-word-tokens`. Give this the name `moby-vocab`.

Note: there are a lot of choices that go into processing text when doing work with a corpus. That is not the point of this problem set. To make things easier:

- Don't worry about case. So, treat `With` and `with` as different words.
- Don't worry about punctuation. Treat `.` as a word just like any other. Also, note that commas are treated as whitespace in Clojure so they will be ignored by your code. We left them in the list `moby-word-tokens` just for readability.

Answer 1: Please put your answer in `ps3-lastname-firstname.clj`.

Problem 2: Define a function `get-count-of-word`. This function should take three arguments, `w`, `word-tokens`, and `count`, where `w` is a word, `word-tokens` is a list of words, and `count` is a number.

When you call `(get-count-of-word w word-tokens 0)`, the function should return the number of occurrences of the word `w` in the list `word-tokens`. For example

- `(get-count-of-word 'the (list 'the 'the 'whale) 0)` should return `2`.
- `(get-count-of-word 'the (list 'the 'whale) 0)` should return `1`.

Write `get-count-of-word` as a recursive function. You can use the `count` argument to accumulate the words counted so far.

Answer 2: Please put the answer in `ps3-lastname-firstname.clj`.

Problem 3: In the template Clojure file, we have provided a function `get-word-counts`, which takes two arguments, `vocab` and `word-tokens`, where `vocab` is assumed to be a list of the unique words that occur in the list `word-tokens`.

```
(defn get-word-counts [vocab word-tokens]
  (let [count-word
        (fn [w] (get-count-of-word w word-tokens 0))]
    (map count-word vocab)))
```

This function returns the number of times each word in `vocab` occurs in `word-tokens`. For example, suppose `vocab` is `'(whale the is)`, and `word-tokens` is `'(the is whale is)`. Then the function will return the list `(1 1 2)`, corresponding to the number of times `'whale`, `'the`, and `'is` occur in `word-tokens`, respectively.

Use this function and the other variables we have defined, to define a variable named `moby-word-frequencies`. This variable should contain the number of times each word in `moby-vocab` occurs in `moby-word-tokens`.

Answer 3: Please put your answer in `ps3-lastname-firstname.clj`.

Problem 4: In class we defined the functions `normalize`, `flip`, and `sample-categorical`. These functions will be very useful for us, and are included below as well as in the Clojure template file.

```
(defn flip [p]
  (if (< (rand 1) p)
    true
    false))

(defn normalize [params]
  (let [sum (apply + params)]
    (map (fn [x] (/ x sum)) params)))

(defn sample-categorical [outcomes params]
  (if (flip (first params))
    (first outcomes)
    (sample-categorical (rest outcomes) (normalize (rest params)))))
```

We have also provided a function that returns a particular probability distribution, the *uniform distribution*. The uniform distribution is the distribution which assigns equal probability to every possible outcome. The function `create-uniform-distribution` takes a single argument, `outcomes`, which is a list of length n. The function returns a list containing the number $1/n$ repeated n times. For example, if `outcomes` is

`'(the a every)`, then this function will return `'(1/3 1/3 1/3)`. This list can be interpreted as a probability distribution over the outcomes, which assigns equal probability to each of them.

```
(defn create-uniform-distribution [outcomes]
  (let [num-outcomes (count outcomes)]
    (map
      (fn [x] (/ 1 num-outcomes))
      outcomes)))
```

Using functions `create-uniform-distribution` and `sample-categorical`, write a function `sample-uniform-BOW-sentence` that takes two arguments: a number `n` and a list `vocab`, and returns a sentence of length `n`. Each word in the sentence should be generated independently from the uniform distribution over vocab. For example, if `n` is `4` and `vocab` is `'(the a every)`, a possible return value for this function is `'(a the the a)`.

Note that this is a bag of words model, as defined in class. That is, we assume every element of the list is generated independently. We will call this the uniform bag of words model.

Answer 4: Please put your answer in `ps3-lastname-firstname.clj`.

Problem 5: Define a function `compute-uniform-BOW-prob`, which takes two arguments, `vocab` and `sentence`. `vocab` is the list of all words in the vocabulary, and `sentence` is a list of observed words. The function should return the probability of the sentence according to the uniform bag of words model.

For example, if `vocab` is `'(the a every)`, and sentence is `'(every every)`, then the function should return the number $\frac{1}{9}$.

Answer 5: Please put your answer in `ps3-lastname-firstname.clj`.

Problem 6: Using `sample-uniform-BOW-sentence` and `moby-vocab`, sample a 3-word sentence from the vocabulary of our Moby Dick corpus. This will be a sample from the uniform bag of words model for this vocabulary. Repeat this process a handful of times. For each of these 3-word sentences, use `compute-uniform-BOW-prob` to compute the probability of the sentence according to the uniform bag of words model. Are the different sentences you sampled assigned different probabilities under this model? Explain why this is (or isn't) to be expected.

Answer 6: Please put your answer in the box below.

After running (def mob-sentence1 (sample-uniform-BOW-sentence 3 moby-vocab)) once, we got the sentence '(way myself high). After computing its probability with (compute-uniform-BOW-prob moby-vocab mob-sentence1), we get a value of 3.6443148688046643e-7, which is equal to 1/2744000. After computing the probabilities of other sentences such as '(throws such all) and '(long shore With), we get the exact same probability as the first sentence. This is to be expected since the words are distributed as a uniform distribution, meaning that every word has the exact same chance at being picked to form a sentence. Therefore, it is normal that we get the same probabilities for all 3-word sentences since every word has the same likelihood of being picked to form a sentence. To further confirm our answer, if we do (count moby-vocab), we get the value 140, so every word has a (1/140) chance to be picked. Then, if we do 1/140 to the power of 3, we get 1/2744000, further proving that the probabilities calculated are the right ones and will all be identical.

Problem 7: In class we looked at a more general version of the bag of words model, in which different words in the vocabulary can be assigned different probabilities. We defined a function `sample-BOW-sentence`, which returns a sentence sampled from the bag of words model that we have specified. Below we have included a slight variant of the function which we defined in class. Previously the variables vocabulary and probabilities were defined outside of the function. In the current version, they are passed in as arguments. The function is identical otherwise.

```
(defn sample-BOW-sentence [len vocabulary probabilities]
  (if (= len 0)
    '()
    (cons (sample-categorical vocabulary probabilities)
          (sample-BOW-sentence (- len 1) vocabulary probabilities))))
```

The function `sample-BOW-sentence` allows us to sample a sentence given arbitrary probabilities for the words in our vocabulary. Let's make use of this power and define a distribution over the vocabulary which is better than the uniform distribution. We will use the word frequencies for our Moby Dick corpus to *estimate* a better distribution.

Above we defined the variable `moby-word-frequencies`, which contains the frequency of every word that occurs in our Moby Dick corpus. Using `normalize` and `moby-word-frequencies`, define a variable `moby-word-probabilities`. This variable should contain probabilities for every word in `moby-vocab`, in proportion to its frequency in the text. A word which occurs 2 times should receive twice as much probability as a word which occurs 1 time.

Answer 7: Please put your answer in `ps3-lastname-firstname.clj`.

Problem 8: Using `sample-BOW-sentence`, sample a 3-word sentence from a bag of words model, in which the probabilities are set to be those in `moby-word-probabilities`. Repeat this process at least three times, and write down the sentences that you collect through this process.

Answer 8: Please put the output sentences in the box below.

After running (sample-BOW-sentence 3 moby-vocab moby-word-probabilities) 5 times, we get the following sentences:

'(it nearly meet)

'(that time of)

'(find as I)

'(is part soul)

'(it the .)

Problem 9: Define a function `lookup-probability`, which takes three arguments, `w`, `outcomes`, and `probs`. `probs` represents a probability distribution over the elements of `outcomes`. For example, if outcomes is `'(the a every)`, then `probs` may be `'(0.2 0.5 0.3)`. The first number in `probs` is the probability of the first element of outcomes, the second number in probs is the probability of the second element of outcomes, and so on.

`lookup-probability` should look up the probability of the element `w`. For example, if `w` is `'the`, then look-up probability should return `0.2`. If `w` is `'a`, then `lookup-probability` should return `0.5`. If `w` is not in the list of outcomes, the function should return that its probability is zero.

Answer 9: Please put your answer in `ps3-lastname-firstname.clj`.

Problem 10: Using `lookup-probability`, define a function `compute-BOW-prob` which takes three arguments, `sentence`, `vocabulary`, and `probabilities`. The arguments `vocabulary` and `probabilities` are used to define a bag of words model with the associated probability distribution over vocabulary words. The function should compute the probability of the sentence (which is a list of words) according to the bag of words model.

This function is a generalization of the function `compute-uniform-BOW-prob` that you defined above.

Answer 10: Please put your answer in `ps3-lastname-firstname.clj`.

Problem 11: In problem 8, you collected a number of 3-word sentences. These sentences were generated from a bag of words model in which the probabilities were set to those in `moby-word-probabilities`, which reflect the relative frequency of the words in the Moby Dick corpus. Use `compute-BOW-prob` to compute the probability of these sentences according to the bag of words model. How does your answer differ from problem 6?

Choose one of the 3-word sentences that you have generated. Can you construct a different sentence which has the same probability according to the bag of words model? When computing the probability of a sentence under a bag of words model, what information about the sentence suffices to compute this probability?

Answer 11: Please put your answer in the box below.

The probability for '(it nearly meet) is now 4.381483020274546e-7, the probability for '(that time of) is 8.762966040549092e-7, the probability for '(find as I) is 0.000003943334718247091, the probability for '(is part soul) is 4.381483020274546e-7, and the probability for '(it the .) is 0.00003505186416219636 4. As we can see, all probability values have changed for each sentence given that the probability values associated to each word has now been weighted based on the amount of times it actually appears in the initial corpus. Moreover, most probabilities are different from one another except for '(it nearly meet) and '(is part soul).

If we choose the sentence '(it nearly meet), we can definitely compute another sentence with the same probability. For example, another randomly generated sentence like '(is part soul) has the exact same probability as well. Furthermore, if we just mix and match the three words in '(it nearly meet) to something like '(meet nearly it), we will also get a sentence that has the same probability as '(it nearly meet) since both sentences are constructed with the same words.

The information that is needed to compute a probability of a sentence is the probability of each individual word in the sentence. We then can do the product of all words in a given sentence to compute the sentence's designated probability.