CS 38003
Homework 2: Language Modeling

Release date: Wed 02/27
Due date: Wed 03/06

Introduction

Almost all devices around us today have a built in language understanding and generation module as Apple's Siri, Amazon's Alexa, Google Assistant ...etc.
These applications use machine learning and natural language processing techniques to achieve that goal. One technique that's heavily used in all of these applications is called "Language Modeling". Language Modeling is a technique that machines uses to model human languages (E.g. a model for English, another model for French, Spanish, Arabic ... etc.).

Language models performs two tasks:

1. ***Score text:***
   Given in input sentence, the language model should be able to score this sentence depending how well it fits the grammar and vocabulary learned by the language model.

2. ***Generate text:***
   Similar to the text prediction in your phone, the language model should be able to predict one word at a time, generating running text.

In this Homework, you will be implementing a simple language called "N-gram" Language Model (N-gram LM). The "N" in N-gram is the number of contextual words considered during training. Words depend on the context they appear in; N is the length of such context. For example, 3-gram LM is the model which restricts itself to context of length equals 3 words, such that each word depends on the previous 2 words, so more generally an n-gram model assumes each word depend on the previous n-1 words.

How can a machine build a language model?

The idea is to show the machine a sample running text of the language of interest. A language model algorithm would analyze the given text and identify the key language constructs, grammar and vocabulary which we call a language model.
Example: if we train the machine using texts from sports news articles, it would be able to identify key words from the sports domain as "score", "win", "lose", "foul", "touchdown", "dripple" … etc.

We call this phase a ***training phase***, which is exactly as it sounds, the program is training to identify the key language constructs, grammar and vocabulary of the given text. The output of the training phase is the language model, which we can use later to generate our own text. It's worth noting here that the language model is as good as the text used to train it. I.e. if the text used is written poorly and with grammatical mistakes and typos, the language model would learn these mistakes and it would make them later on during text generation.

Now that program is trained and we have our language model, we'd like to test our model.
We call this phase ***testing phase***. In this phase, we want to test our model on new data.
Using the language model, we should be able to assess how likely (represented in score) any input sentence to belong to the trained language model.
For example, if the language model is built using text from sports news articles, then a good language

model would give better score to a sentence from a sports article than a sentence coming from an article on economics.

Data Description

Although your code should work with any textual dataset. A famous dataset for language modeling is called text8. It's a cleaned dataset collected from Wikipedia. Although, this dataset is cleaned, you will be required to implement your own text cleaning function that would be used for test sentences and for any other training/testing datasets that might not be cleaned.

Task 1: Reading Data

The first task is to read data from files. This function should take the path to a file (training or testing) and return a list of lines read for the input file.

You are required to implement the function:

def **read_data**(fileName):

Inputs:

> 1) fileName: the name of the data file (e.g. "TRAINING.txt").
> Type: string

Returns:

> 1) lines: a list of lines read from the file.
> Type: A list of strings

Task 2: Data Cleaning

If you look at the abundant textual data that you find online, you will notice that people write in lower case, upper case, use exclamation marks, smiley faces … etc. These different types of writing can confuse your program, so in this task you are required to clean these inconsistencies from the reviews.

Below are the cleaning instructions, each instruction should be fulfilled regardless of the order of cleaning instructions used.

1. Numbers as prefix or suffix must be separated from the word:
   xyz12 → xyz 12
   12xyz →12 xyz

2. Convert all numbers to token 'num':
   xyz 12 → xyz num

3. Numbers inside words should be left alone:
   seq2seq → seq2seq

4. Only single hyphen separating words should be removed:
   power-wise → powerwise
   well--it's looking good → well it's looking good

5. Continuous numbers should be just one ‘num’ token:
   1 2.2 35 1.3 → num

6. All special non-alphanumeric characters should be removed (except for '):
   it's good! → it's good

7. Only one single space should separate words:
   the   water → the water

8. All letters should be converted to lower case.

You are required to implement the function:

def **clean_data**(my_data):

Inputs:

1) my_data: A list of lines.
Type: A list of strings

Returns:

1) cleaned_data: A list of cleaned lines.
Type: A list of strings

Note: Data cleaning step should be applied to both training dataset and any testing dataset.

Training Phase

In the training phase, you will be building N-gram language model.

Task 3: Building N-gram Language Model

How to build n-gram language model for a given an input text?

The n-gram model is simply a dictionary with parts of the input text being the keys and their frequencies are the values.

Below are the steps to build a language model:

1. Tokenize the text: Tokenization is the process of extracting words from text. Since the input text assumed to be cleaned, then splitting the text by space would give the words.
2. Add starting symbol <s> as the starting word in the beginning of all sentences. (Assume that every line is a new sentence).
3. Scan the words in order with n-sized window, forming sentences.
4. Calculate the frequencies of the n-sized sentences.

For example:
Assume the input text was “if it talks like a duck walks like a duck it’s a duck”
For n=1 (we’re building a 1-gram (uni-gram) language model). We would scan the text with a window of 1, then we advance the window by 1 and keep counting the resulting words.
In this case: the words scanned are:

<s>, if, it, talks, like, a, duck, walks, like, a, duck, it's, a, duck.

1-gram vocabulary	count
<s>	1
if	1
it	1
talks	1
like	2
a	3
duck	3
walks	1
it's	1

For n=2 (we're building a 2-gram (bi-gram) language model). We would scan the text with a window of 2, then we advance the window by 1 and keep counting the resulting words.
In this case: the words scanned are:

<s> if, if it, it talks, talks like, like a, a duck, duck walks, walks like, like a, a duck, duck it's, it's a, a duck

2-gram vocabulary	count
<s> if	1
if it	1
it talks	1
talks like	1
like a	2
a duck	3
duck walks	1
walks like	1
duck it's	1
it's a	1

You are required to implement the function:

def **build_n_gram_dict** (n, cleaned_data)

Inputs:

1) n:
"n" in the n-gram
Type: int

2) cleaned_data:
A list of cleaned sentences.
Type: A list of strings

This function takes n and the cleaned_data and builds an n-gram dictionary with the keys being the n-grams and the values are their corresponding frequencies.

Returns:

1) ngram_dict: Dictionary<key:string, value:int>

Testing Phase

In this phase, we are concerned with measuring and quantifying how good our models are.

Task 4: Perplexity

Perplexity (PP) is a metric used to measure the performance of language models. The PP of M words-sentence $(w_1\, w_2\, w_3 \ldots\, w_M)$ is:

$$PP(w_1\, w_2\, w_3 \ldots\, w_M) = P(w_1\, w_2\, w_3 \ldots\, w_M)^{-1/M}$$

Where w_i is the word at position i in the sentence, and P is probability of this sentence being generated by n-gram language model. The lower the perplexity, the better.

Let's see an example of how to calculate the probability of a sentence. let's assume we have the following sentence, and we have a 3-gram LM:

"<s> if it talks like a duck walks like a duck it's a duck"

Where P is probability of this sentence being generated by a 3-gram language

$$\begin{aligned}
&P(<s> \text{ if it talks like a duck walks like a duck it's a duck}) = \\
&P(if \mid <s>) \times \\
&P(it \mid <s>\, if) \times \\
&P(like \mid it\ talks) \times \\
&P(a \mid talks\ like) \times \\
&P(duck \mid like\ a) \times \\
&P(walks \mid a\ duck) \times \\
&\ldots \\
&P(duck \mid it's\ a)
\end{aligned}$$

Where

$$P(if \mid <s>) = \frac{count(<s>\, if) + 1}{count(<s>) + V}$$

$$P(it \mid <s>\, if) = \frac{count(<s>\, if\ it) + 1}{count(<s>\, if) + V}$$

$$P(like \mid it\ talks) = \frac{count(it\ talks\ like) + 1}{count(it\ talks) + V}$$

$$P(a \mid talks\ like) = \frac{count(\text{“}talks\ like\ a\text{”}) + 1}{count(\text{“}talks\ like\text{”}) + V}$$

$$\ldots$$

$$P(duck \mid it's\ a) = \frac{count(it's\ a\ duck) + 1}{count(it's\ a) + V}$$

Where V is the length of the vocabulary. I.e. V is the number of unique words in the training dataset. It's worth noting that using 3-gram LM requires to build a 2-gram LM (to compute the counts of 2-word sentences as "$talks\ like$") and a 1-gram LM (to compute the counts of 1-word sentences as "$< s >$ ", and to compute V)

You are required to implement the function **calculate_PP** which calculate the PP for each test sentence and returns the average PP
E.g. if we have T sentences in the test set, then the average PP is:

$$Average\_PP = \frac{PP_1 + PP_2 + \cdots + PP_T}{T}$$

def calculate_PP(test_sentences, ngram_models)

Inputs:

1) test_sentences:
A list of testing sentences
Type: list<string>

2) ngram_models:
The n-gram models built by the function **build_n_gram_dict.** Since if we are using 3-gram LM we need also 2-gram and 1-gram LM. Then the n_gram_models is a dictionary of these language models, where n is the key and the value is the n-gram LM.
Type: A dictionary of dictionaries: dictionary <key: n, value: dictionary<key:string, value:int>>

Returns:

1) avg_PP: float
The average PP of the test data.

Task 5: Text Generation

Language models can be used to generate text similar to the predictive text of your phone. It starts with a starting seed word, then it generates the most likely word to follow that seed word, then generates the most likely word following those 2 words and so on.

More concretely, assume that we start with the seed word w_{seed}, use your LM calculate the probability $P(w_i \mid context)$ for all words in the vocabulary, and pick the word with maximum probability.
Context is the past n-1 words generated so far by the n-gram language model.

Example:
For a 3-gram LM, assume our vocabulary is "like, I, pizza, music, and" and our starting seed word is "like", so our context is "<s> like". To generate the next word, we calculate for each word in the vocabulary w_i the probability $P(w_i \mid < s > \ like)$:

$$P(like \mid < s > \ like), P(I \mid < s > \ like), P(pizza \mid < s > \ like),$$
$$P(music \mid < s > \ like), P(and \mid < s > \ like)$$

Then we pick the word with maximum probability. Let's assume that word was "pizza". Now our generated text is "<s> like pizza". To generate the next word, the context would be "like pizza" (because we have 3-gram LM), and we calculate for each word in the vocabulary the probability $P(w_i \mid like\ pizza)$:

$$P(I \mid like\ pizza),\ P(like \mid like\ pizza),\ P(pizza \mid like\ pizza),$$
$$P(music \mid like\ pizza),\ P(and \mid like\ pizza),$$

Then we pick the word with maximum probability, and so on.

You are required to write the function:
def generate_text(ngram_models, text_length, seed_word)

Inputs:

1) ngram_models:
The n-gram models built by the function **build_n_gram_dict.** Since if we are using 3-gram LM we need also 2-gram and 1-gram LM. Then the n_gram_models is a dictionary of these language models, where n is the key and the value is the n-gram LM.
Type: A dictionary of dictionaries: dictionary <key: n, value: dictionary<key:string, value:int>>

2) text_length
The length of the text to be generated
Type: Int

3) seed_word
A starting seed word
Type: string

Returns:

1) generated_text
The text generated, whose number of words should match text_length (including the <s> and the seed_word)
Type: string

Turning in Your Work

- Submit your completed file (HW2.py) on Blackboard.

Rubric

HW 2	
Task 1: Reading Data	**5%**
Task 2: Cleaning Data	**25%**
Task 3: Building n-gram model	**30%**
Task 4: Evaluation	**20%**
Task 5: Text Generation	**20%**
TOTAL	**100%**