CptS 215 Data Analytics Systems and Algorithms (https://apps.aoi.wsu.edu/coursemoreinfo/course_more_iuid=2203-25600)

Washington State University (https://wsu.edu)

Srini Badri (https://school.eecs.wsu.edu/people/faculty/) (Instructor)

Gina Sprint (http://eecs.wsu.edu/~gsprint/) (Original Author)

PA4 Keyboard (100 pts)

Due:

Learner Objectives

At the conclusion of this programming assignment, participants should be able to:

- Implement hash tables and hash functions
  - Linear probing
  - Chaining
- Implement maps
- Understand a unigram language model

Prerequisites

Before starting this programming assignment, participants should be able to:

- Write object-oriented code in Python
- Work with lists

Acknowledgments

Content used in this assignment is based upon information in the following sources:

- Predictive keyboard assignment (https://katie.mtech.edu/classes/archive/s13/csci136/assign/keyboard/) from Montana Tech

Overview and Requirements

Natural language processing (NLP) refers to computational technique involving language. It is a broad field. For this assignment, we will learn a bit about NLP to build a predictive keyboard. The keyboard learns to make predictions by text given at startup and by things the user types into the program. As users type, the program predicts the most likely word given the currently entered word prefix. So for example, if the user has typed "th", the keyboard will probably predict "the".

We will use a hash map to implement the keyboard, which is crucial to making the keyboard's learning and predictions fast and efficient. For a hash map, we will use our own `Map` class.

Note: for this assignment, *do not use Jupyter Notebook* to code your solution. Use standard .py files.

Program Details

Background: Language Modeling 101

Our predictive keyboard will make use of a very simple model of language (https://en.wikipedia.org/wiki/Language_model), a *unigram language model* (https://en.wikipedia.org/wiki/Language_model#Unigram_models). Our keyboard will keep track of every unique word it has seen. For each unique word, it will count how many times that word has occurred. A simple *maximum likelihood* estimate for the unigram probability of a word is to take the number of times you've seen that word occur divided by the total number of words you've encountered. For example, here is a small bit of text:

> The cat is in the corner of their garage.

There are 9 total words in our text. Ignoring case and punctuation, the word the appears twice. The unigram probability of "the" is thus $P(the) = 2/9$. The unigram probability of "cat" is $P(cat) = 1/9$. The other six words: is, in, corner, of, their, garage all also have a probability of 1/9. If a word has never been seen, its probability is zero: $P(zebra) = 0/9 = 0.0$.

Given the data, if the user were to type in "c", both "cat" and "corner" are equally likely. But if they type "ca", we would predict "cat", while if they typed "co" we'd predict "corner". If the user typed "ci", we would make not prediction (since none of our words start with "ci").

Datasets

Download the following text files (keyboard_files.zip (https://raw.githubusercontent.com/gsprint23/cpts215/master/progassignments/files/keyboard_files.zip)):

1. moby_start.txt (https://raw.githubusercontent.com/gsprint23/cpts215/master/progassignments/files/moby_start.txt): The first paragraph from Moby-Dick (202 words).
2. moby_end.txt (https://raw.githubusercontent.com/gsprint23/cpts215/master/progassignments/files/moby_end.txt): The last two paragraphs from Moby-Dick (232 words).
3. mobydick.txt (https://raw.githubusercontent.com/gsprint23/cpts215/master/progassignments/files/mobydick.txt): The entire book Moby-Dick (209K words).
4. wiki_200k.txt (https://raw.githubusercontent.com/gsprint23/cpts215/master/progassignments/files/wiki_200k.txt): 200,000 sentences from Wikipedia (3.8M words).

Classes to Define

Map

Implement a `Map` class that inherits from a `HashTable` class (see the class notes for much of this code). Your `HashTable` class should implement chaining for collision handling.

DictEntry

We need a class to represent a word and its unigram probability. Here is the API to `DictEntry`:

```
# create a new entry given a word and probability
__init__(word, prob) # string, float
# getter for the word
get_word() # returns string
# getter for the probability
get_prob() # returns float
# does this word match the given pattern?
match_pattern(pattern) # (optional) returns string
```

WordPredictor

The brains of this whole operation is the class `WordPredictor`. This class learns how to make word predictions based on being shown training data. It can learn from all the words in a file (via the `train()` method) or from a single individual word (via the `train_word()` method). After new training data is added, the `build()` method must be called so the class can recompute the most likely word for all possible prefixes. Here is the API for the `WordPredictor` class:

```
# train the unigram model on all the words in the given file
train(training_file) # string
# train on just a single word
train_word(word) # string
# get the number of total words we've trained on
get_training_count() # returns integer
# get the number of times we've seen this word (0 if never)
get_word_count(word) # string. returns integer
# recompute the word probabilities and prefix mapping
build()
# return the most likely DictEntry object given a word prefix
get_best(prefix) # string. returns DictEntry
```

Training the Model

Model training occurs in the `train()` and `train_word()` methods. `train()` should parse out each word in the specified file on disk. If the file cannot be read, it should print out an error, "Could not open training file: file.txt". All training words should be converted to lowercase and stripped of any characters that are not a-z or the single apostrophe. During training, you need to update the instance variables:

- `word_to_count` : `Map` of string (word key) to integer (count value) pairs
- `total` : Integer

The `word_to_count` instance variable is a `Map` where the keys are the unique words encountered in the training data. The values are the integer count of how many times we've seen each word in the data. Only words seen in the training data will have an entry in the `word_to_count` `Map` . The total instance variable tracks how many words of training data we have seen. That is, total should equal the sum of all integer counts stored in your map. Training is cumulative for repeated calls to `train()` and `train_word()` , so you just keep increasing the counts stored in your `word_to_count` map and your total count of training words.

Making Predictions

The class needs to be able to predict the most probable word for a given word prefix give the statistics found during training. For example, after training on mobydick.txt (https://raw.githubusercontent.com/gsprint23/cpts215/master/progassignments/files/mobydick.txt), if the user types the prefix "wh", "wha", "whal" or "whale", the most likely word is "whale". The job of mapping word prefix strings to the most probable dictionary entry falls to the third instance variable:

- `prefix_to_entry` : `Map` of string (prefix key) to `DictEntry` (value) pairs

A `Map` can map multiple keys to the same value. This is exactly what we need in this case since a set of strings (such as "wh", "wha", "whal" and "whale") may all need to map to a single `DictEntry` object. You need to ensure that each prefix maps to its most likely word. So even if the word "thespian" was encountered first in the training data, for any sensible training text, "th" should probably map to "the" and not "thespian".

Every time the `build()` method is called, you need to re-initialize the contents of `prefix_to_entry` . This ensures that you take into account any new words or updated counts present in `word_to_count` . Here is an overview of the algorithm you should be aiming for in `build()` :

1. Loop over all possible (key, value) pairs in `word_to_count` .
    A. For each pair: Calculate the probability of that word given its count and the number of training words. Create a new `DictEntry` object representing this word and its probability.
    B. For each pair: Loop over all possible prefixes of the word. That is from a prefix of only the first letter to a "prefix" that is the entire word.
        a. For each prefix, if the current key (word) has a probability strictly greater than what is currently stored for that prefix, replace the old value (a `DictEntry` object) with the new object.

This results in our map containing a mapping from every valid prefix of every training word to the most probable complete word under the unigram language model. Your `get_best()` method can be a one-liner!

Testing the Prediction Class

There is a fairly extensive test `main()` available to in keyboard_test.py (https://raw.githubusercontent.com/gsprint23/cpts215/master/progassignments/files/keyboard_test.py). The first part of `main()` tests out your training routines, while the second half focuses on the prediction part. Here is the output you can use to compare with for correctness (not all output will match exactly, think about this):

```
bad1 = null
training words = 202
bad2 = null
Could not open training file: thisfiledoesnotexist.txt
training words = 202

count, the = 10
count, me = 5
count, zebra = 0
count, ishmael = 1
count, savage = 0

count, the = 32
count, me = 5
count, zebra = 0
count, ishmael = 1
count, savage = 1

a -> and    0.03917050691244239
ab -> about 0.004608294930875576
b -> bird   0.004608294930875576
be -> before    0.002304147465437788
t -> the    0.07373271889400922
th -> the   0.07373271889400922
archang -> archangelic  0.002304147465437788
training words = 434

before, b -> bird   0.004608294930875576
before, pn -> null
after, b -> bird    0.004576659038901602
after, pn -> pneumonoultramicroscopicsilicovolcanoconiosis  0.002288329519450801
training words = 437

a -> and    0.030079417288752873
ab -> about 0.001472985727769356
b -> but    0.008580499385064212
be -> be    0.0048908846494866
t -> the    0.06757143265738066
th -> the   0.06757143265738066
archang -> archangel    3.3368608719694154E-5
training words = 209778

elapsed (s)           : 0.459

Random load test:
elapsed (s)           : 3.447
Hit % = 30.0537
```

Bonus (10 pts)

1. (5 pts) Create a more advanced predictive keyboard that can show the top *N* possible completions for the current word. The value for N is passed as the first argument on the command-line. The interface should show up to N words that are consistent with the currently typed letters. Less than N words may be shown for some prefixes. The letters should be labeled with the numbers 0 up to N-1. If the user hits a number, the coorresponding predictions (if any) is output. The return key should select the best prediction. Try this using the wiki_200k.txt (https://raw.githubusercontent.com/gsprint23/cpts215/master/progassignments/files/wiki_200k.txt) file.
2. (5 pts) Make your predictions depend on not only the current word prefix, but also on the prior word (i.e. a *bigram language model* (https://en.wikipedia.org/wiki/Language_model#n-gram_models)). Note that you might not always be able to make a prediction using the prior word (e.g. at the start of the sentence or if the prior word was never seen in the training data). This is a common problem in language modeling called *sparsity*. One way to handle this problem is to *backoff* to a lower-order language model. In your case, this would mean backing off to the unigram model which can always make a prediction as long as the current prefix is consistent with some word in the training data.

Submitting Assignments

1. Use the Blackboard tool https://learn.wsu.edu (https://learn.wsu.edu) to submit your assignment. You will submit your code to the corresponding programming assignment under the "Content" tab. You must upload your solutions as `<your last name>_pa4.zip` by the due date and time.
2. Your .zip file should contain your .py files and your .txt files used to test your program.

Grading Guidelines

This assignment is worth 100 points + 10 points bonus. Your assignment will be evaluated based on a successful compilation and adherence to the program requirements. We will grade according to the following criteria:

- 15 pts for correct `Map` implementation
- 15 pts for correct chaining collision handling
- 15 pts for correct `DictEntry` implementation
- 40 pts for correct `WordPredictor` implementation using `Map`. `WordPredictor` class should include the following methods:
  - `train()`
  - `train_word()`
  - `build()`
  - `get_best()`
  - `get_word_count()` and `get_training_count()`
- 10 pts for correct definition and use of the `word_to_count` and `prefix_to_entry` map objects
- 5 pts for adherence to proper programming style and comments established for the class