Final Project

MPCS 51042 – Python Programming

Due: December 10th 2020 @ 11:59pm CT (No Extensions)

Initial Setup

Make sure to perform a pull upstream inside your repository. This will grab the distribution code for the **final_proj**. The command is the following:

```
$ git pull upstream master
```

Style Guide

For this homework and all future homework assignments, we will follow the style guide used by the undergraduate Python course. It's located here: `https://classes.cs.uchicago.edu/archive/2020/fall/12100-1/style-guide/index.html`

Introduction

In this final project, you will develop a modeling system using Markov Models. You don't have to understand them completely (or at all) for this assignment but here are some helpful slides if you are curious:

`http://cecas.clemson.edu/~ahoover/ece854/refs/Ramos-Intro-HMM.pdf`

Markov models can be used to capture the statistical relationships present in a language like English. These models allow us to go beyond simplistic observations, like the frequency with which specific letters or words appear in a language, and instead to capture the relationships between words or letters in sequences. As a result, we can not only appreciate that the letter "q" appears in text at a certain rate, but also that it is virtually always followed by a "u." Similarly, "to be" is a much more likely word sequence than "to is."

One application of Markov models is in analyzing actual text and assessing the likelihood that a particular person uttered it. That is one objective of this assignment.

The other objective for this assignment is to create a hash tables module. Most of you should be familiar with Hash tables, which are data structures that store associations between keys and values (exactly like dictionaries in Python) and provide an efficient means of looking up the value associated with a given key. Hash tables find a desired entry rapidly by limiting the set of places where it can be. They avoid "hot spots," even with data that might otherwise all seem to belong in the same place, by dispersing the data through hashing.

Apart from developing an appreciation of how hashing and hash tables work, you will also be better prepared if ever you need to write your own hash table in the future. For instance, you may use a language (like C) that does not feature a built-in hash table. Or, the hash table that is used by a language, like Python, may interact poorly with your particular data, obligating you to design a custom-tailored one. After completing this assignment, you should consider hash tables to be in your programming repertoire.

Hash tables and Linear Probing

In the interest of building and testing code one component at a time, you will start by building a hash table. Once you have gotten this module up and running, you will use it in your construction of a Markov model for speaker attribution.

There are different types of hash tables; for this assignment, we will use the type that is implemented with **linear probing**.

Hashtable Class

Please look at `final_proj/hash_table.py`. You will modify this file and **must** implement a hash table using the linear probing algorithm. The class `Hashtable` must have:

- It inherits the `Map` abstract base class (`final_proj/map.py`) and implements all abstract methods specified in its definition.
- Assume all keys are strings.
- It defines an `__init__(self,capacity,defVal,loadfactor,growthFactor)` method that takes in the following:
  - `capacity` - the initial number of cells to use. It must create a list of empty cells with the specified length. You can assume the value passed in for the initial number of cells will be greater than 0.
  - `defVal`- a value to return when looking up a key that has not been inserted. (see prior section for an example)
  - `loadfactor` - a floating point number ($(0, 1]$). If the fraction of occupied cells grows beyond the `loadfactor` after an update, then you must perform a rehashing of the table. Rehashing is described below.
  - `growthFactor` - an integer greater than or 1 that represents how much to grow the table by when rehashing. For example, if `growthFactor = 2` then the size of the hash table will double each time we rehash.
- It must use a single Python `list` to hold the key and value as tuple in the table. Tuple can contain any additional components as needed.
- A `_hash(self,key)` method takes in a string and returns a hash value. Use the standard string hashing function (i.e., horners method) discussed in lecture.
- **Rehashing**: A hash table built on linear probing does not have unlimited capacity, since each cell can only contain a single value and there are a fixed number of cells. But, we do not want to have to anticipate how many cells might be used for a given hash table in advance and hard-code that number in for the capacity of the hash table. Therefore, we will take the following approach: the cell capacity passed into the `__init__` will represent the initial size of the table. If the fraction of occupied cells grows beyond `loadfactor` (i.e., parameter passed into `loadfactor`) after an update, then we will perform an operation called **rehashing**: We will expand the size of our hash table, and migrate all the data into their proper locations in the newly-expanded hash table (i.e., each key-value pair is hashed again, with the hash function now considering the new size of the table). We will grow the size of the table by `growthFactor` (i.e., parameter passed into `__init__`) ; for instance, `growthFactor = 2`, the size of the hash table will double each time it becomes too full.
- **Deletion**: deleting an item in a hash table that uses linear probing is tricky. Simply removing the key-value pair could potentially invalidate the entire table. For our implementation, we will "logically" delete the key-value pairings. This means that you should have marker for each key-value pair that indicates whether or not it's been "deleted" (e.g., having a third component of the tuple to represent the marker). If the marker is `True` then the key-value pair is still inside the table; otherwise a `False` marker says that the pair was deleted at some point. You should think about how this helps with implementing

your insertion and lookup methods... You will "physically" remove the key-value pairings that have been "logically" removed during rehashing.

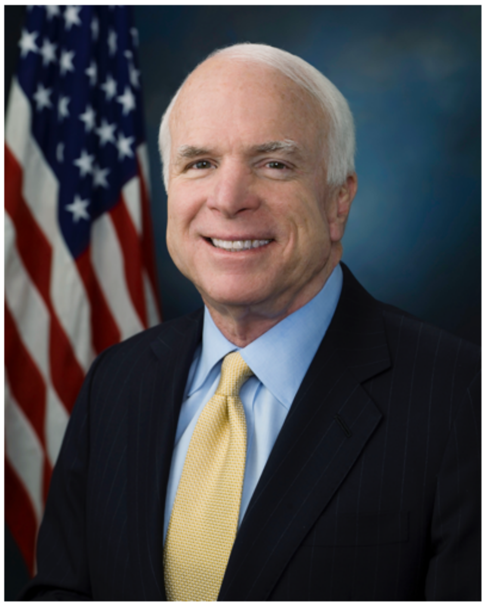
- The syntax to add, update, delete, etc. are the same as a Python dictionary. You should be able to know how to modify and access the data inside the `Hashtable` instances.
- You are free to implement additional properties, attributes, and methods as needed to implement this class.
- I have provided you with a pytest file, `final_proj/test_hash_table.py` that you can use to verify your implementation is correct. You must pass these tests in less than 120 seconds. You still be given credit for passing the tests if you could takes longer than 120 seconds but there will be a penalty.

A Speaker Recognition System

Speaker A: Barack Obama

"Well, thank you very much, Jim, and thanks to the commission and the University of Mississippi, Ole Miss, for hosting us tonight. I can't think of a more important time for us to talk about the future of the country.
You know, we are at a defining moment in our history. Our nation is involved in two wars, and we are going through the worst financial crisis since the Great Depression…"

Speaker B: John McCain



"Well, thank you, Jim. And thanks to everybody.
And I do have a sad note tonight. Senator Kennedy is in the hospital. He's a dear and beloved friend to all of us. Our thoughts and prayers go out to the lion of the Senate.
I also want to thank the University of Mississippi for hosting us tonight.
And, Jim, I -- I've been not feeling too great about a lot of things lately. So have a lot of Americans who are facing challenges. But I'm feeling a little better tonight, and I'll tell you why…"

Speaker C (Unidentified): Who ***most likely*** is the speaker of the below text? **Obama** or **McCain**

"I think we can, for all intents and purposes, eliminate our dependence on Middle Eastern oil and Venezuelan oil. Canadian oil is fine."

Markov models are used significantly in speech recognition systems, and used heavily in the domains of natural language processing, machine learning, and AI.

Markov model defines a probabilistic mechanism for randomly generating sequences over some alphabet of symbols. A **k**-th order Markov model tracks the last k letters as the context for the present letter. We will build a module called **Markov** that will work for any positive value of k provided. This module, naturally, resides in **markov.py**.

While we use the term "letter," we will actually work with all characters, whether they be letters, digits, spaces, or punctuation, and will distinguish between upper- and lower-case letters.

Building the Markov Model (Learning Algorithm)

Given a string of text from an unidentified speaker, we will use a Markov model for a known speaker to assess the likelihood that the text was uttered by that speaker. Thus, the first step is building a Markov Model on known text from a speaker. For this assignment, the Markov Model will be represented as a `Hashtable` (**Note:** You will not use all the methods you defined in its implementation). **You must use your implementation and not the built-in dictionaries from Python!** You will be given an integer value of k at the start of the program. Each key-value pairing inside the table contains string keys with length k and $k+1$ and values set to the number of times those keys appeared in the text. For example, let's say you have a file called **speakerA.txt** that contains the following text:

```
This_is_.
```

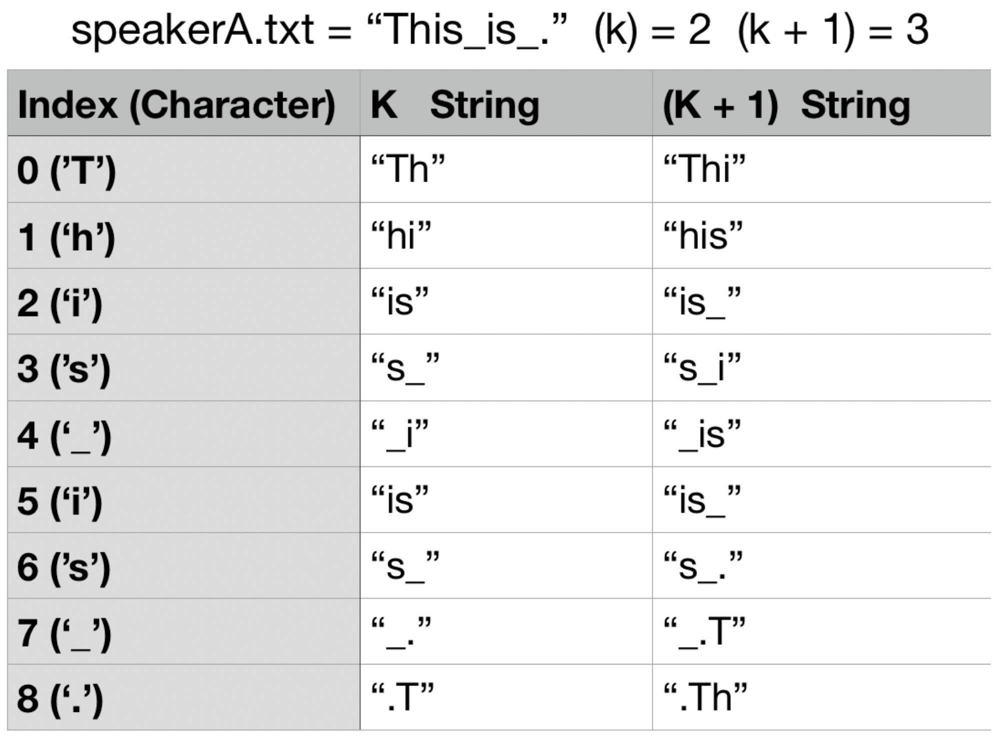
Note: You consider all characters in the text. Regardless if the characters are punctuation, special characters, whitespace characters, etc., all characters in the text are considered valid. Even if the text does not make any sense, especially in this case, you will still generate a Markov Model.

We will use this text (i.e., `"This_is_."`) to create a Markov Model for Speaker A. The algorithm proceeds as follows:

Starting from the beginning of the text for some known speaker:

1. For each character in the known text, you generate a string of length k that includes the current character plus $k-1$ succeeding characters (**Note**: The model actually works by finding the k preceding letters but our way works too because we are using a wrap-around effect.).
2. For each character in the known text, you generate a string of length $k+1$ that includes the current character plus k succeeding characters.
3. For certain characters, they will not have k or $k+1$ succeeding characters. For example, what are the succeeding characters for the character `'.'` if $k = 2$ in the **speakerA.txt** text? We will **wrap around**: we will think of the string circularly, and glue the beginning of the string on to the end to provide a source of the needed context. For instance, if $k = 2$, and we have the string `"ABCD"`, then the letters of context for `'D'` will be `"A"` (for $k-1$) and `"AB"` (for k).

Below is a diagram of all of the k and $k+1$ length strings that are generated from the **speakerA.txt** file given that $k = 2$:



speakerA.txt = “This_is_.” (k) = 2 (k + 1) = 3

Index (Character)	K String	(K + 1) String
0 (’T’)	“Th”	“Thi”
1 (‘h’)	“hi”	“his”
2 (‘i’)	“is”	“is_”
3 (’s’)	“s_”	“s_i”
4 (‘_’)	“_i”	“_is”
5 (‘i’)	“is”	“is_”
6 (’s’)	“s_”	“s_.”
7 (‘_’)	“_.”	“_.T”
8 (‘.’)	“.T”	“.Th”

The Markov Model (i.e., `Hashtable`) will contain the number of times those k and $k+1$ were generated via the known text. Thus, for the **speakerA.txt** file, the Markov Model generated will be the following:

Markov Model (i.e., Hashtable)

Key (K & K+1 Strings)	Value (Counts)
“Th”	1
“hi”	1
“is”	2
“s_”	2
“_i”	1
“_.”	1
“.T”	1
“Thi”	1
“his”	1
“is_”	2
“s_i”	1
“_is”	1
“s_.”	1
“_.T”	1
“.Th”	1

Most of the k and $k+1$ strings were only generated once but some such as `"is"` were generated by processing the character at index 0 and index 5.

Determining the likelihood of unidentified text (Testing Algorithm)

As we stated earlier, given a string of text from an unidentified speaker, we will use the Markov model for a known speaker to assess the likelihood that the text was uttered by that speaker. Likelihood, in this context, is the probability of the model generating the unknown sequence. **If we have built models for different speakers, then we will have likelihood values for each, and will choose the speaker with the highest likelihood as the probable source**.

These probabilities can be very small, since they take into account every possible phrase of a certain length that a speaker could have uttered. Therefore, we expect that all likelihoods are low in an absolute sense, but will still find their relative comparisons to be meaningful. Very small numbers are problematic, however, because they tax the precision available for floating-point values. The solution we will adopt for this problem is to use **log probabilities** instead of the probabilities themselves. This way, even a very small number is represented by a negative value in the range between zero and, for instance, -20. If our log probabilities are negative and we want to determine which probability is more likely, will the greater number (the one closer to zero) represent the higher or lower likelihood, based on how logarithms work?

Note that when we use the Prelude's `math.log` function (i.e., we will calculate natural logarithms). Your code should use this base for its logarithms. While any base would suffice to ameliorate our real number precision problem, we will be comparing your results to the results from our implementation, which itself uses natural logs.

The process of determining likelihood given a model is similar to the initial steps of building the Markov Model in the previous section.

Starting from the beginning of the text for some unknown speaker:

1. For each character in the unknown text, you generate a string of length k that includes the current character plus $k - 1$ succeeding characters.
2. For each character in the known text, you generate a string of length $k + 1$ that includes the current character plus k succeeding characters.
3. For certain characters, they will not have k or $k+1$ succeeding characters. You will use the same **wrap around** mechanism as described previously.
4. We need to keep in mind that we are constructing the model with one set of text and using it to evaluate other text. The specific letter sequences that appear in that new text are not necessarily guaranteed ever to have appeared in the original text. Consequently, we are at risk of dividing by zero.

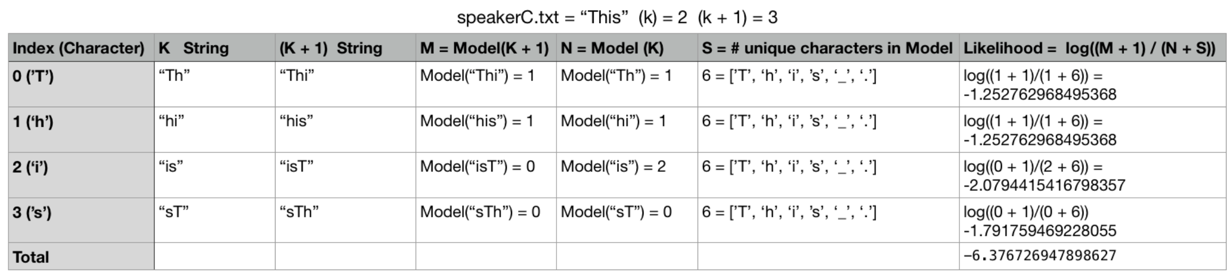
   It turns out that there is a theoretically-justifiable solution to this issue, called **Laplace smoothing**. We modify the simple equation above by adding to the denominator the number of unique characters that appeared in the original text we used for modeling. For instance, if every letter in the alphabet appeared in the text, we add 26. (In practice, the number is likely to be greater, because we distinguish between upper- and lower-case letters, and consider spaces, digits, and punctuation as well.) Because we have added this constant to the denominator, it will never be zero. Next, we must compensate for the fact that we have modified the denominator; a theoretically sound way to balance this is to add one to the numerator. Symbolically, if N is the number of times we have observed the k succeeding letters and M is the number of times we have observed those letters followed by the present letter, and S is the size of the "alphabet" of possible characters, then our probability is :math:

$$log((M + 1)/(N + S))$$

For example, lets say you have file called **speakerC.txt** (i.e., the unidentified text):

```
This
```

Calculating the total likelihood will be done in the following way using the model we built from the **speakerA.txt** text:



speakerC.txt = "This" (k) = 2 (k + 1) = 3

Index (Character)	K String	(K + 1) String	M = Model(K + 1)	N = Model (K)	S = # unique characters in Model	Likelihood = log((M + 1) / (N + S))
0 ('T')	"Th"	"Thi"	Model("Thi") = 1	Model("Th") = 1	6 = ['T', 'h', 'i', 's', '_', '.']	log((1 + 1)/(1 + 6)) = -1.252762968495368
1 ('h')	"hi"	"his"	Model("his") = 1	Model("hi") = 1	6 = ['T', 'h', 'i', 's', '_', '.']	log((1 + 1)/(1 + 6)) = -1.252762968495368
2 ('i')	"is"	"isT"	Model("isT") = 0	Model("is") = 2	6 = ['T', 'h', 'i', 's', '_', '.']	log((0 + 1)/(2 + 6)) = -2.0794415416798357
3 ('s')	"sT"	"sTh"	Model("sTh") = 0	Model("sT") = 0	6 = ['T', 'h', 'i', 's', '_', '.']	log((0 + 1)/(0 + 6)) -1.791759469228055
Total						−6.376726947898627

Note:

- When we say "Model(K)" we are looking at the hash table inside the model and retrieving the counts for that string.
- S is the number of unique characters that we encountered when building the model. When building the model you need to create a set of all the unique characters.

Markov Class

Inside the `final_proj/markov.py`, you will a implement the `Markov` class with the following instance methods:

1. An `__init__` method that takes in a value of "k" and a string of text to create the model and a `state` variable. The markov class will use a hash table as its internal state to represent the markov model. If the variable `state` is

   - `state == 0`: You will use your class you defined inside `hash_table.py`. You will define one or more hash tables with `HASH_CELLS` many cells; we have provided this constant to be a suitable number that is a good starting size for your hash tables, although they will have to grow significantly to accommodate all the statistics you will learn as you scan over the sample text we have provided. You can define the hash table as follows: `Hashtable(HASH_CELLS, 0, 0.5, 2)`.
   - `state == 1`: You will use the built-in `dict` data structure to represent the internal markov model.

   You will run performance tests on whether your implementation of the hash table is faster or slower than using the built-in `dict` type. We will discuss this performance testing in a later section.

2. `log_probability` is a method that takes in a new string and returns the log probability that the modeled speaker uttered it, using the approach described in a prior section.

Inside the `markov.py` file define the following functions:

- `identify_speaker(speech1, speech2, speech3, order, state)` - This function is called by the `main` function with three strings (i.e., `(speech1, speech2, speech3` and a value of k (i.e., `order`). `state` represents whether the markov model object should use your implementation of a hash table or `dict` type. You must learn models for the speakers that uttered the first two strings, calculate the **normalized** log probabilities that those two speakers uttered the third string, and return these two probabilities in a tuple (with the first entry being the probability of the first speaker). Finally, you must compare the probabilities and place in the third slot of your returned tuple, a conclusion of which speaker was most likely. This conclusion should be either the string `"A"` or `"B"`.

  While the `log_probability` function yields the likelihood that a particular speaker uttered a specified string of text, this probability may be misleading because it depends upon the length of the string. Because there are many more possible strings of a greater length, and these probabilities are calculated across the universe of all possible strings of the same length, we should expect to see significant variance in these values for different phrases. To reduce this effect, we will divide all of our probabilities by the length of the string, to yield normalized ones. To be clear, this division does not occur in `log_probability`, but rather in `identify_speaker`. Also, note that we will be calculating log probabilities, under different models, for the same string. Thus, string length differences are not a factor during a single run of our program. Rather, we choose to normalize in this fashion because we want outputs for different runs to be more directly comparable when they may pertain to different length quotes being analyzed.

Note: You can test your implementation of this file by running the pytests inside the `final_proj/test_markov` file. You must pass these tests in less than 120 seconds. You still be given credit for passing the tests if you could takes longer than 120 seconds but there will be a penalty.

Driver File

For this assignment, You will need to implement a driver for the speaker recognition system inside (`final_proj/driver.py`). We have provided a set of files containing text from United States presidential debates from the 2004 and 2008 general elections. In the 2004 election, George W. Bush debated John Kerry; in the 2008 debates, Barack Obama went up against John McCain. We have provided single files for Bush, Kerry, Obama, and McCain to use to build models. These files contain all the text uttered by the corresponding candidate from two debates. We have also provided directories from the third debates of each election year, containing many files, appropriately labeled, that have remarks made by one of the candidates.

Inside the `final_proj/driver.py` file, define a `if __name__ == "__main__"` block that reads in command-line arguments. The driver will have two modes:

- **Normal mode**: In normal mode, the driver will read in **four** command-line arguments where the first two file names represent the text files that will be used to build the two models, and the third file name is that of the file containing text whose speaker we wish to determine using the Markov approach. The third argument is the order (k) of the Markov models to use, an integer. the final argument is "state" argument where `state == 0` then the program builds the Markov models using your `Hashtable` implementation. `state == 1` the program builds the Markov models using the built in `dict` type.

  Your code should then read in text from the text files and call the the `identify_speaker` function to retrieve the log probabilities and conclusion string tuple. Your program will then print out the the log probabilities and the conclusion as follows:

  ```
  Speaker A: -2.1670591295191572
  Speaker B: -2.2363636778055525

  Conclusion: Speaker A is most likely
  ```

  Here's a full sample use of calling the program from the terminal window:

  ```
  $ ipython driver.py speeches/bush1+2.txt speeches/kerry1+2.txt speeches/bush-kerry3/BUSH-0.txt 2 1
  Speaker A: -2.1670591295191572
  Speaker B: -2.2363636778055525

  Conclusion: Speaker A is most likely
  ```

- **Performance measurement mode**: In performance measurement mode, you will run performance tests on your two different internal states for your Markov model class. The driver will read **five** arguments. To help your driver determine which mode to use (i.e., performance measurement mode or normal mode), if the first argument is simply a `"p"` then you will run the performance measurement mode. Similar to the normal mode, the next command-line arguments will be two file names represent the text files that will be used to build the two models, and the third file name is that of the file containing text whose speaker we wish to determine using the Markov approach. The fifth argument will be the "k" range and the last argument is the number of "runs". How to execute and use these arguments to do the testing is specified in the section below.

- You do not need to do error checking on the command line. You can assume we will provide the arguments with valid values and in the correct order.

Performance Measurement Mode

You will perform time measurements between your Markov model states to determine which one is faster and record these timings inside a **single `pandas`** dataframe. Specifically, you will time how fast the `identify_speaker` returns based on specifying a range of k values starting at 1 to "k" values (inclusive) and running it multiple times (i.e., "run" command line arguments (inclusive)). For example, assume we run the program as follows,

```
$ ipython driver.py p speeches/bush1+2.txt speeches/kerry1+2.txt speeches/bush-kerry3/BUSH-0.txt 2 3
```

Then the pandas dataframe could **potentially** look like this:

Implementation	K	Run	Time
Hashtable	1	1	0.345
Hashtable	1	2	0.356
Hashtable	1	3	0.386
Hashtable	2	1	0.567
Hashtable	2	2	0.601
Hashtable	2	3	0.598
dict	1	1	0.045
dict	1	2	0.056
dict	1	3	0.076
dict	2	1	0.167
dict	2	2	0.101
dict	2	3	0.198

Where you will run these timings on the same three speech file each time you call `identify_speaker`. For each `k`, you will need to run it multiple times, which is determined by "runs" command-line argument. Why do we need to run `identify_speaker` with the same k multiple times?

Unfortunately, running and timing a program once can give misleading timings because:

- A process may create a cache on its first execution; therefore, running faster subsequently on additional executions
- Other processes may cause the command to be starved of CPU or I/O time
- There might be random interrupt that causes the timing to be an outlier

You will use these timings to make a seaborn graph to show how times fluctuation based on the implementation and the increasing k value. Thus, for each timing indicated above you will actually take the average of those timings for each k.

Here's another example:

```
$ ipython driver.py p speeches/bush1+2.txt speeches/kerry1+2.txt speeches/bush-kerry3/BUSH-0.txt 3 2
```

Then the pandas dataframe could look like this:

Implementation	K	Run	Time
Hashtable	1	1	0.345
Hashtable	1	2	0.356
Hashtable	2	1	0.686
Hashtable	2	2	0.567
Hashtable	3	1	1.001
Hashtable	3	2	1.198
dict	1	1	0.045
dict	1	2	0.056
dict	2	3	0.176
dict	2	1	0.167
dict	3	1	0.301
dict	3	2	0.308

I'm not saying this should be the structure of your dataframe! You will need to think clearly about how to structure your dataframe. **You can only use one pandas dataframe**. After receiving a time measurement you will need to update that pandas dataframe. Thus, you must think about how you will structure this dataframe when you define it. You will be graphing the data inside this dataframe. I would highly recommend you think about how to structure this code based on how seaborn will need the data.

Helpful Pandas Advice

- Think about using `df.groupby`
- Think about using `df.loc`

Timing your code

You can use the `time` package to take your code. Specifically you want to get the elapsed time between calling the `identify_speaker`. Here's the code you could use:

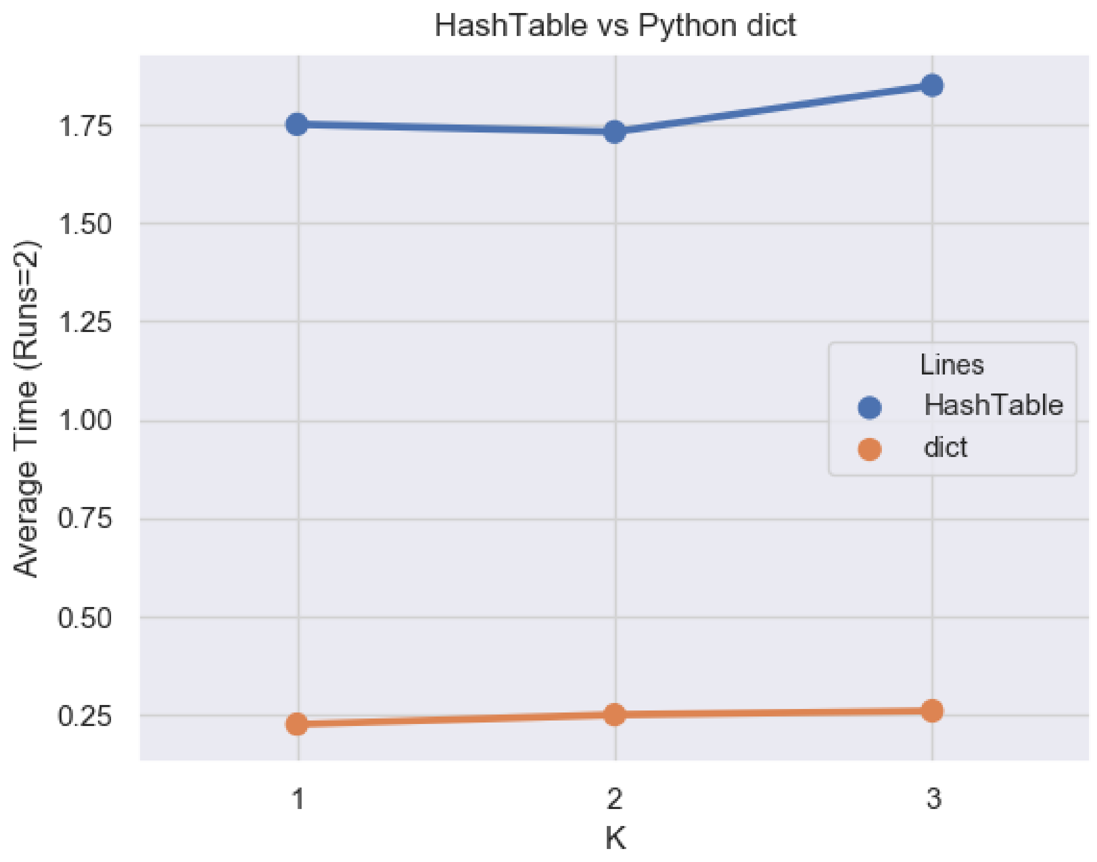
```
import time
...
start = time.perf_counter()
tup = identify_speaker(speech1, speech2, speech3, k, state)
end = time.perf_counter()
elapsed_time = float(f'{start - end:0.4f}') #0.4f formats the time to 4 decimal places.
# Now store "elapsed_time" inside the pandas dataframe
```

Graphing

After producing your pandas dataframe, you will produce a seaborn (or matplotlib) graph where the x values will be the k values and the y values will be **average time of the runs** for each k value per implementation type. Here's an example graph of running the following program:

```
$ ipython driver.py --p speeches/bush1+2.txt speeches/kerry1+2.txt speeches/bush-kerry3/BUSH-0.txt 3 2
```

The execution graph should have the similar structure:



Similar to the graph shown above. Make make sure to title the graph, and label each axis. Make sure to adjust your y-axis range so that we can accurately see the values. That is, if most of your values fall between a range of [0,1] then don't make your graph range [0,14]. You can use a point plot with the following arguments:

```
sns.pointplot(..., linestyle='-', marker='o')
```

You will save the graph as `execution_graph.png`

Debugging suggestions

A few debugging suggestions:

- Make sure you have chosen the right data structure for the Markov model!
- Check your code for handling the wrap-around carefully. It is a common source of errors.
- Test the code for constructing a Markov model using a very small string, such as `"abcabd"`. Check your data structure to make sure that you have the right set of keys and the correct counts.
- Make sure you are using the correct value for S the number of unique characters that appeared in the *training* text.

Acknowledgment

This assignment's text and documentation originated from CAPP 30122 Team @ The University of Chicago. The original development of the assignment was done by Rob Schapire with contributions from Kevin Wayne.