COMP 200 & COMP 130
Assignment 4: Data Structures, Markov Chains, PCA

Be sure to read the course policies, as posted on the course web site:

http://www.clear.rice.edu/comp200/policies.shtml

Work in assigned pairs on this assignment. Each pair should put all assignment answers in one Python file called *`netid1_netid2_4.py`*, substituting in the students' NetIDs. Put the answers to non-programming problems in comments. When submitting, only one of the member should turn in the pair's answers.

Total points: 100 for COMP 200, 150 for COMP 130.

Wherever reasonable, use functions that have been defined previously on this assignment, on a previous assignment, or in class.

Create a separate folder for this assignment and place all your code in it. This folder should also contain a subfolder called `texts`, which will contain all the sample text files that you use. You may use any of the provided samples, as well as any other text files. Download sample text files.

"Zip" up your assigment folder, and submit the resulting zip file.

Data Structures – COMP 200 (25 points) & COMP 130 (30 points)

1. (10 points)

   Briefly describe the similarities and differences between Python lists, dictionaries, and tuples.

2. In the class notes, when defining `countSequences`, we repeatedly took slices to get the length-n subsequences of the data. E.g., if we had the data `aList = [0,1,2,3,4,5]`, we used `aList[0:3]`, `aList[1:4]`, `aList[2:5]`, and `aList[3,6]` to get the length-3 tuples that we used as dictionary keys.

   In this problem, we will motivate and explore an alternative approach.

   Python tends to obscure the idea of how long each operation takes. But, computing a length-n slice requires times proportional to n, i.e., "linear" time, because each of those n list elements is copied.

   (a) **COMP 130 only:**
   (5 points)
   Give code that illustrates that a slice actually copies the data, rather than referring to the same pieces of data.

   As an alternate approach, consider the following idea. We first take linear time to get our first subsequence by using a slice, as before. Then for each next subsequence, we want to take advantage of the natural overlap between subsequences. We simply need to add the next list element to the back of our subsequence, and remove the one at the front. For example, when we have the subsequence `[0,1,2]`, we add `3` at the end and remove `0` at the front and get `[1,2,3]`. Since we are only changing two elements, this takes "constant" time.

   (b) (15 points)
   Define a version of `countSequences` that uses this approach.

Markov Chains – COMP 200 & COMP 130 (75 points)

We separate the problem of generating text into two halves. First, we construct a Markov chain model of our input text. Second, we use this model to generate "riffs", i.e., new text based upon the original. We keep these parts separate because we often want to use the same Markov chain for multiple riffs.

3. Following the outline provided in the class notes, define the function `makeMarkovChain(filename,n)`. It takes a filename and a positive integer. It returns an n-th order Markov chain, i.e., a dictionary mapping n-word tuples to a probabilistic description of the possible successor words.

   (a) (25 points)

   Define a function `countSequenceSuccessors(words,n)` that creates a dictionary. Each key would be an n-word tuple subsequence of the input. Each value would be a dictionary or `Counter`, mapping successor words to their counts.

   For example, the class notes illustrate that the lyrics to *Johnny B. Goode*, with $n = 2$, results in a dictionary that includes the key/value pair `(".","Goode"): {"!": 3, "tonight": 1, ",": 1}`.

   Hint: As its name and description implies, this function is similar to that of `countSequences`. To get each n-word sequence, you can use either the approach of the original `countSequences` in the notes or the approach of the previous assignment problem.

   Hint: The code is simplest if you use a `defaultdict` of `Counter`s. These built-in data structures let you avoid the hassle of checking whether a key is already present before accessing it.

   (b) (15 points)

   Define the function `makeChain(sequenceDict)` that takes the output of the previous function and returns a new dictionary. Each key of the returned dictionary should be an n-word tuple subsequence of the original input, i.e., the same keys as in the given dictionary. Each value should be a dictionary, mapping successor words to their probabilities.

   For example, the class notes illustrate that the lyrics to *Johnny B. Goode*, with $n = 2$, results in a dictionary that includes the key/value pair `(".","Goode"): {"!": 0.6, "tonight": 0.2, ",": 0.2}`.

   In other words, for each key such as `(".","Goode")`, it is looking at the corresponding value `{"!": 3, "tonight": 1, ",": 1}`, and converting the counts to probabilities.

   The function `makeMarkovChain` then just calls these functions.

```
def makeMarkovChain(filename,n):
    """Return an nth-order Markov chain, given a filename."""
    return makeChain(countSequenceSuccessors(inputText(filename),n))
```

4. Following the outline provided in the class notes, define the function `makeRiff(chain,n,length)`. It takes an n-th order Markov chain (created by `makeMarkovChain`) and a positive integer `length`. As described in class, it returns a "riff" string with the indicated number of words randomly generated from the chain.

   (a) (15 points)

   Define a function `chooseWordFromChain(chain,words)` that takes a Markov chain and a list of words. It uses the list of words as a key to choose a random next word based upon the Markov chain. It returns `None` if the list doesn't have any successor word in the chain.

   Hint: You might want to use `random.uniform(0,1)`, or something similar.

(b) (17 points)

Define `makeRiff(chain,n,length)`. It starts the riff with a n-word sequence chosen randomly from among the chain's keys.

To generate each next riff word, we consider the n previous words. Those n previous words of the riff will likely be present as a key in the chain/dictionary. We get the corresponding value, which gives the probabilities of each possible successor word. Respecting those probabilities, we pick among the possible words and add it to the riff.

We stop when we have generated `length` words in the riff. We also stop if the n previous words of the riff are not a key in the chain/dictionary. (This only occurs if those n words only occurred at the end of the original text that we used to create the Markov chain.)

Return the resulting riff as a string with whitespace. If you have the riff words as a list, you can simply use `" ".join(words)`, which puts a space in between each word and punctuation.

5. (3 points)

Use the previous functions on the various supplied input texts. Provide an example riff (preferably a funny or interesting one), and indicate what file it was based on.

PCA Analysis – COMP 130 (45 points total)

6. Complete the functions listed in the Principle Component Analysis (PCA) prep and usage lecture notes and perform explorations of the ability of PCA to distinguish authorship of unknown texts. You will need the text parsing and word tuple counting functions from the text analysis lecture notes. You are free to create additional helper functions as well and use any provided functions, as desired. You are expected to use other functions whenever possible when writing each of your functions so that your code most directly expresses exactly what your function does. Proper documentation strings are required for all functions.

   (a) (27 pts total) Create the following functions from the lecture notes.

      i. (3 pts) `combine_dicts(dicts)`
      ii. (3 pts) `getTotalValues(dict)`
      iii. (3 pts) `makeNormalizedDict(dict)`
      iv. (3 pts) `makeChunks(aList, nElts)`
      v. (3 pts) `countChunks(chunks)`
      vi. (3 pts) `getTopVallues(aDict, keyList)`
      vii. (3 pts) `getTopValMatrix(aDicts, keyList)`
      viii. (3 pts) `makePCADataFromFile(??)` – you decide what input parameters this should take
      ix. (3 pts) `getNormalizedTopValsFromFile(??)` – you decide what input parameters this should take

   (b) (18 pts total) Perform the following analyses on at least two sets of "known" text data compared against multiple "known" and "unknown" texts and make conclusions as to the effect of varying each parameter. Your code should automatically execute, computing the results on which you base your conclusions. *Be specific in your analysis*, clearly stating exactly what you did, your results and why you came to your conclusions. Put your concluding remarks as a comment after your supported calculations.

      i. (4 pts) Vary the chunk size.
      ii. (4 pts) Vary the number of top values used.
      iii. (4 pts) Vary the word tuple size.

iv. (3 pts) Create a set of functions that you automatically perform what you believe the optimal analysis and comparison of known and unknown texts. Create two separate functions (plus any desired supporting functions), one to prepare the known data and one to compare the unknown data to the prepared known data. Run your functions on various data to show your results.

v. (3 pts) In your opinion, is the PCA analysis that you performed a valid technique for determining text authorship? Is so, why. If not, why not? Could your PCA analysis be enhanced or is there something completely different that you think should be done? There is no "right" answer here. You will be graded on the clarity of your discussion.

Feedback – COMP 200 & COMP 130 (0 points)

1. Roughly how many hours did you spend on the two sets of finger exercises?
2. On a scale of 1 (very easy) to 5 (very difficult), how difficult were the finger exercises?
3. Roughly how many hours did you spend on this homework?
4. On a scale of 1 (very easy) to 5 (very difficult), how difficult was this homework?
5. Which material did you find most challenging?
6. Did you feel that the class material adequately prepared you for the homework?