Homework 01 STAT/CS 287

This assignment is primarily a collection of programming exercises to prepare us for bigger challenges. Aim for short, simple, readable code. Try to choose useful variable names and code comments. Avoid redundant or duplicated code ("code clones") as much as possible!

My goals for writing scientific code are, in order of decreasing importance:

1. Correctness
2. Clarity
3. Conciseness
4. Efficiency/speed

Forming good habits for ***reproducible*** code is a crucial skill, so it's included in the grading criteria.

Instructions:

Read these instructions completely before you begin.

Along with this file is the writeup file and several empty Python script files for you to use as templates problems given below. Please fill these script files with your code as you answer these problems. *Follow any and all directions for full credit!*

- Please rename the included files using your UVM NetID. For example, I would rename `problem1_NETID.py` to `problem1_jbagrow.py`. Please put your name and date into the appropriate locations at the top of each `.py` file.
- Please place any code necessary for answering a given problem within its respective `.py` file.
- Recall for this and all assignments that any **BONUS** questions are optional for undergraduates but are required for graduate students (and those receiving graduate credit).
- You can include *equations* as needed inside your writeup using Word's equation editor or by inserting a scanned image of your handwritten (and legible!) equations.
- Please show all your work. This means providing as part of your submission the Python code, mathematical derivations, or written arguments as appropriate for any and all problems in this assignment.
- When finished, please upload your finished `.py` files and your completed **writeup**, to the Blackboard course website. See the "Preparing and submitting homework" slides for submission details.

You are expected to encounter errors in code while working on assignments. Dealing with broken code (for example, a script trying to import a module that isn't installed on your machine) is something you frequently confront in practice. Fixing such errors is something you'll need to be able to do.

Good luck!

Problem 1:

Python's **set** data structure supports mathematical set operations (union, intersection, difference, etc.)

1. Using online resources, IPython exploration, and other means, learn how to use **Python sets**, then answer the following questions:

    a. How do sets differ from dictionaries?

b. How are sets similar to dictionaries?
c. If you did not have a builtin `set` data type in Python, could you create one from the other data types? If you cannot create sets, why not? If you can create sets, briefly describe how.

2. Please write a function `similarity(A,B)` that computes a *similarity* between two sets A and B, defined as:

$$\text{similarity}(A, B) = \frac{|A \cap B|}{|A \cup B|}$$

where $|S|$ is the cardinality of set S, $A \cap B$ is the intersection of sets A and B, and $A \cup B$ is the union of sets A and B.

3. Create two or three small example sets within your `.py` file. Call your similarity function on all pairs of these sets. Print the sets and their similarities in a readable summary format, and paste this into your writeup.

Problem 2:

Flipping a biased random coin.

Using the Python `random` module, write a function `coin_flip(p)` that takes a number $0 \leq p \leq 1$ and returns either **heads** or **tails**. If $p = 1/2$ then both heads and tails are equally likely and the coin is fair. If $p = 0.9$ then there is a 90% chance of a head and only a 10% chance of a tail.

- You can use `'H'` and `'T'` to represent heads and tails, respectively.

Now, `for p in [0.2,0.4,0.6,0.8]`, do the following:

1. Flip the coin 1000 times, storing the heads or tails result to a list after each flip.
2. Compute the **run statistics** for heads for this coin flip history. A run of heads is a contiguous sequence of all heads. For example, `['H','H','H','T','H','T','H','H']`, has a run of 3, then a run of 1, then a run of 2. The run statistics count how many runs there are of length 1,2,3,4,5, etc.

To be clear: do the above two steps once for each value of `p`. Meaning, you set `p = 0.2`, flip 1000 coins, etc., then set `p = 0.4`, flip 1000 coins, etc.

Collate your run statistics into a **table in your writeup**. Include in this table the counts of each run length up to runs of length 10 (include counts of zero for unobserved lengths). Each column of this table should correspond to a value of p and each row a run length. Include a final row in the table containing the *average run length*. You can generate this table in Python and paste it into your writeup. Include appropriately labeled header rows and columns to make this table readable.

Bonus: Determine or estimate the probability for a run of length L as a function of L (and p). Answer this mathematically as best you can, by writing down a formula for this probability; do not write code to answer this bonus question.

Problem 3:

Within the `problem3` script template there is a function that downloads and parses *A Tale of Two Cities* from Project Gutenberg. This function returns a list of all words in the book. Study how this function works.

1. Modify the download section of this function to **cache** your book download. This means you first try to read the book as a text file saved to your homework's enclosing folder and only download the book if this file does not exist. If the book needs to be downloaded, then create the text file by saving the book. Use relative paths. Why is caching the book download a useful and important step?
2. Write a function called `count_most_common(word_list)` that takes the book's word list and counts the number of times each word occurs. Use this function to print the 100 most common words and the number of times each occur, in descending order. Paste this printout into your writeup.

- For P3.2, you must use and document an appropriate and efficient data structure. The final code should require only a few seconds to complete. Please use `.sort()` or `sorted()` in your solution.

Bonus: Try to find a special Python library or package to do P3.2 quickly. Describe this library/package and how to use it for P3.2. However, please use this library only after you have implemented your own "homegrown" solution.

Problem 4:

Coin dependence

Building off the `coin_flip` function from P2, implement a pair of **dependent** coins. For two coins, each flip is a tuple of two heads/tails outcomes. For simplicity, assume the coins are always flipped one following the other. Let X be the (random) variable representing the first coin (or the outcome of flipping the first coin), let Y be second coin, and let p be the probability that the first coin gives heads (`'H'`) when flipped.

Now, suppose the second coin depends on the first coin (assume it is flipped *after* the first coin). Specifically, let the second coin give heads with probability q_1 when the first coin gave heads, and otherwise let the second coin gives heads with probability q_2 when the first coin gave tails.

1. Implement this dependent coin pair in a Python function `dependent_coin_flip(p, q1, q2)`. Note the three probabilities passed as input. Make sure that `dependent_coin_flip` returns a `(X,Y)` tuple representing the outcomes of the two coins, respectively. We can think of the output of this function as representing an *event pair* for the two coins being flipped. Build this new function by reusing the `coin_flip` function from P2 within `dependent_coin_flip`. (Do not write any code that is ***redundant*** with `coin_flip`; instead, call `coin_flip` from within `dependent_coin_flip` as needed.) Why is it important and useful to reuse the code from P2?
2. Can `dependent_coin_flip` represent a pair of independent coins? If so, how? If not, why not?
3. Flip the dependent coins 1000 times to generate a list of event pairs. Use two non-equal combinations of parameters (p, q_1, q_2) when flipping the coins. Count the number of times each unique event occurred. Print these counts into a readable table and paste into your writeup, along with a justification for your choice of parameters.
4. Given a list of events for a random variable, what is the simplest way to estimate the probability of each possible event? Can this method miss events?
5. Using the event pairs generated by your 1000 flips of the dependent coins, estimate the **joint probability** $P(X, Y)$. Write code to compute this and represent $P(X, Y)$ in a small table printed out by your code and pasted into your writeup.

Bonus 1: Using the definition of dependence and the 1000 event pairs, check whether the dependent coins are dependent or not. If you can represent independent coins (see previous question), flip 1000 *in*dependent coin pairs, and check whether they are independent or not. Please provide both a written derivation of your answer and a Python implementation.

Bonus 2: Suppose you flipped the dependent coins n times (instead of 1000) and consider n to be small (maybe even as small $n = 5$ or $n = 10$). With such few observed event pairs, it may be difficult to reliably determine if the two coins are independent or dependent. At what value of n can you be reasonably confident you can distinguish between the independent and dependent cases? You can answer this question computationally, with simulations and analysis of those simulations, or mathematically, whichever you prefer. Provide a written argument supporting and motivating your answer in the writeup.