Homework 3: Files and Objects

Due October 3, 11:59 pm

Worth 15 points

Read this first. A few things to bring to your attention:

1. **Important:** If you have not received a Cavium username, please request one here: `http://myumi.ch/6pn5d` (you will need to be on the Michigan network to access this form). List me (Keith Levin, unique name `klevin`) as your "advisor".
2. Start early! If you run into trouble installing things or importing packages, it's best to find those problems well in advance, not the night before your assignment is due when we cannot help you!
3. **Make sure you back up your work!** I recommend, at a minimum, doing your work in a Dropbox folder or, better yet, using `git`, which is well worth your time and effort to learn.

Instructions on writing and submitting your homework.

Failure to follow these instructions will result in lost points. Your homework should be written in a jupyter notebook file. I have made a template available on Canvas, and on the course website at `http://www-personal.umich.edu/~klevin/teaching/Fall2019/STATS507/hw_template.ipynb`. You will submit, via Canvas, a `.zip` file called `yourUniqueName_hwX.zip`, where `X` is the homework number. So, if I were to hand in a file for homework 1, it would be called `klevin_hw1.zip`. Contact the instructor or your GSI if you have trouble creating such a file.

When I extract your compressed file, the result should be a directory, also called `yourUniqueName_hwX`. In that directory, at a minimum, should be a jupyter notebook file, called `yourUniqueName.hwX.ipynb`, where again `X` is the number of the current homework. You should feel free to define supplementary functions in other Python scripts, which you should include in your compressed directory. So, for example, if the code in your notebook file imports a function from a Python file called `supplementary.py`, then the file `supplementary.py` should be included in your submission. In short, I should be able to extract your archived file and run your notebook file on my own machine. Please include all of your code for all problems in the homework in a single Python notebook unless instructed otherwise, and please include in your notebook file a list of any and all people with whom you discussed this homework assignment. Please also include an estimate of how many hours you spent on each of the sections of this homework assignment.

These instructions can also be found on the course web page at `http://www-personal.umich.edu/~klevin/teaching/Fall2019/STATS507/hw_instructions.html`. Please direct any questions to either the instructor or your GSI.

1 Counting Word Bigrams (7 points)

In your previous homework, you wrote a function for counting character bigrams. Now, let's write a function for counting word bigrams. That is, for each pair of words, say, `cat` and `dog`, we want to count how many times the word "cat" occurred immediately before the word "dog". We will represent this bigram by a tuple, `('cat', 'dog')`. For our purposes, we will ignore all spaces, newlines, punctuation and capitalization in our counting. So, as an example, the fragment of poem,

> Half a league, half a league,
> Half a league onward,
> All in the valley of Death
> Rode the six hundred.

includes the bigrams `('half', 'a')` and `('a', 'league')` both three times, the bigram `('league', 'half')` appears twice, while the bigram `('in', 'the')` appears only once.

1. Write a function `count_bigrams_in_file` that takes a filename as its only argument. Your function should read from the given file, and return a dictionary whose keys are bigrams (given in the tuple form above), and values are the counts for those bigrams. Again, your function should ignore punctuation, spaces, newlines and capitalization. The strings in your key tuples should be lower-case. Your function should use a try-catch statement to raise an error with an appropriate message to alert the user in the event that the given file cannot be opened, and a different error in the event that the provided argument isn't a string at all. **Hint:** you will find the Python function `str.strip()`, along with the string constants defined in the string documentation (`https://docs.python.org/3/library/string.html`), useful in removing punctuation. **Hint:** be careful to check that your function handles newlines correctly. For example, in the poem above, one of the `('league', 'half')` bigrams spans a newline, but should be counted nonetheless. **Note:** be careful that your function does not accidentally count the empty string as a word (this is a common bug if you aren't careful about splitting the input text). Solutions that merely delete "bad" keys from the dictionary at the end will not receive full credit, as all edge cases can handled by correctly splitting the input.

2. Download the file `WandP.txt` from the course webpage: `http://www-personal.umich.edu/~klevin/teaching/Fall2019/STATS507/WandP.txt`. This is an ASCII copy of all of Tolstoi's novel *War and Peace*. Run your function on this file, and pickle the resulting dictionary in a file called `mb.bigrams.pickle`. Please include this file in your submission, along with `WandP.txt`, so that we can run your notebook directly from your submission.

3. We say that word A is *collocated* with word B in a text if words A and B occur immediately one after another (in either order). That is, words A and B are collocated if and only if either of the tuples `(A, B)` or `(B, A)` are present in the text. Write a function `collocations` that takes a filename as its only argument and returns a dictionary. Your function should read from the given file (raising an appropriate error if the file cannot be opened or if the argument isn't a string at all) and return a dictionary whose keys are all the strings appearing in the file (again ignoring case and stripping away all spaces, newlines and punctuation) and the value of word A is a Python set containing all the words collocated with A. Again using the poem fragment above

as an example, the string `'league'` should appear as a key, and should have as its value the set `{'a', 'half', 'onward'}`, while the string `'in'` should have the set `{'all', 'the'}` as its value. **Hint:** we didn't discuss Python sets in lecture, because they are essentially just dictionaries without values. See the documentation at `https://docs.python.org/3/tutorial/datastructures.html#sets` for more information.

4. Run your function on the file `WandP.txt` and pickle the resulting dictionary in a file called `mb.colloc.pickle`. Please include this file in your submission.

2 More Fun with Vectors (8 points)

In this exercise, we'll encounter our old friend the vector yet again, this time taking an object-oriented approach.

1. Define a class `Vector`. Every vector should have a dimension (a non-negative integer) and a list or tuple specifying its entries. The initializer for your class should take the dimension as its first argument and a list or tuple of numbers (ints or floats), representing the vector's entries, as its second argument (note: your initializer should work correctly given *either* a list or a tuple of numbers). Choose sensible default behavior for the case where the user applies only a dimension and no entries. The initializer should raise an appropriate error in the case where the dimension is invalid (i.e., wrong type or a negative number), and should also raise an error in the event that the dimension and the number of supplied entries disagree.

2. Did you choose to make the vector's entries a tuple or a list? Defend your choice. (There is no wrong answer here, although I would say one is better than the other in this context.)

3. Are the dimension and entries class attributes or instance attributes? Why is this the right design choice?

4. Implement the necessary operator(s) to support comparison (equality, less than, less or equal to, greater than, etc) of `Vector` objects. We will say that two `Vector` objects are equivalent if they have the same coordinates. Otherwise, comparison should be analogous to tuples in Python, so that comparison is done on the first coordinate first, then the second coordinate, then the third, and so on. So, for example, the two-dimensional vector $(2, 4)$ is ordered before (less than) $(2, 5)$. Attempting to compare two vectors of different dimensions should result in an error.

5. Implement a method `Vector.dot` that takes a single `Vector` as its argument and returns the inner product of the caller with the given `Vector` object. Your method should raise an appropriate error in the event that the argument is not of the correct type or in the event that the dimensions of the two vectors do not agree.

6. We would also like our `Vector` class to support scalar multiplication. Left- or right-multiplication by a scalar, e.g., `2*v` or `v*2`, where `v` is a `Vector` object, should result in a new `Vector` object with its entries all scaled by the given scalar. We will also follow R and numpy (which you will learn in a few weeks), and use `*` to denote entrywise vector-vector multiplication, so that for `Vector` objects `v` and `w`, `v*w` results in a new `Vector` object, with the i-th entry of `v*w` equal to the i-th

entry of `v` multiplied by the i-th entry of `w`. Implement the appropriate operators to support this multiplication operation. Many languages have a convention for dealing with multiplication of vectors that differ in their dimension, but we will punt on this matter. Your method should raise an appropriate error in the event that `v` and `w` disagree in their dimensions.

7. For a real number $0 \leq p \leq \infty$, and a vector $v \in \mathbb{R}^d$, the p-norm of v, written $\|v\|_p$, is given by

$$\|v\|_p = \begin{cases} \sum_{i=1}^{d} 1_{v_i \neq 0} & \text{if } p = 0 \\ (\sum_{i=1}^{d} |v_i|^p)^{1/p} & \text{if } 0 < p < \infty, . \\ \max_{i=1,2,\ldots,d} |v_i| & \text{if } p = \infty \end{cases}$$

   Strictly speaking, this is only a norm for $p \geq 1$, but that's beside the point. Implement a method `Vector.norm` that takes a single int or float `p` as an argument and returns the `p`-norm of the calling `Vector` object. Your method should work whether `p` is an integer or float. Your method should raise a sensible error in the event that p is negative. **Hint:** see `https://docs.python.org/3/library/functions.html#float` for documentation on representing positive infinity in Python.