Jorgo Qirjaj (jq16)
COMP614 - Statistics

Python code

The CodeSkulptor file can be accessed using the link here.

2.C. Moving Averages

Referring to ***m*** as in the *length of the input list*, ***n*** as the *input number*, and assuming ***n > 0***.

Q2.C.i: In the case where *n* < *m*, how many pieces of data from the input list should be used to compute an individual element in the output list? Express your answer in terms of *m* and/or *n*.

If n < m, and n > 0, **then each individual element in the output list is always computed using n pieces of data,** regardless of the length of m.

If we take a look at the code, the **moving_avg** function works by creating sublists (or windows) of length **n.** At position *i*, we take the slice *data[i : i + n].* If we look closely, the length of the slice will always be *n*, regardless of the value of *i*.

So it would not matter where this sublist/window started, it **would always contain n values.** The value of m only helps us determine how many sublists/windows will be created in total.

In the example in question, we have *m = 6* and *n = 3,* with *data = [5, 2, 8, 3, 7, 4].* This creates a total of 4 sublists/windows, **each with 3 items** (i.e. **n** items).

Q.2.C.ii: Without using iteration, how could you extract a contiguous sequence of elements from the input list? Provide a Python expression in terms of data *i, n* and or *m* that will give you the sequence of elements that will be used to compute the *i-th* element in the output list in the case that *n* < *m*. Your expression should work for any arbitrary *n* < *m* and any arbitrary value of *i* that is greater than or equal to zero and less than the size of the output list.

If *n* < *m*, **then each individual element in the output list is always computed using n pieces of data** (relating it back to Q1).

To extract a sequence of elements from a list we can use **slicing** in Python. The code to extract specifically the *i-th* element of the output list, would be:

data[i : i + n]

We see that the slice starts at index *i* and ends at *i + n*. SincePython slicing will always include the first element (starting index) and exclude the last element (ending index), then this slice would include **exactly n values**. In other words, it includes the elements that correspond to index *i, i + 1, i + 2, …, i + n -1.*

This holds true for any *m* or *n* as long as ***0 =< i < (m - n + 1)*** and ***n < m****.* To make it clearer, let's look at the same example as Q1, where *m = 6, n = 3,* and *data = [5, 2, 8, 3, 7, 4].* We see that for all values of *i* (from 0 to 3), for each of the 4 sequences created (sublists), we have a total of n = 3 items per sequence (sublist).

Q.2.C.iii: In the case where *n < m*, how many pieces of data will be in your output list? Why? Express your answer in terms of *m* and/or *n.*

As I briefly mentioned in the first question, the output list would always contain **m - n + 1 elements**. Let's break down why that's the case.

Firstly, if the list has m elements, knowing the properties of lists then that means the last element will be at index *m - 1* (i.e. if m = 6, the last index is 5).

Each element of the output list corresponds to a sublist of size *n*. We have a new sublist for each index, starting at index 0 **data[0 : n]**, then index 1 **data[1 : n + 1],** and so on until the last sublist.

So where does the last window (with size n, or a total of *n - 1* elements) start? The starting index would be:

(m - 1) - (n - 1) = m - n

If we take the same example as the other questions, where *m = 6* and *n = 3,* with *data = [5, 2, 8, 3, 7, 4],* we get **3** as the last valid starting index – ***data[3 : 6].***

The number of sublists (or windows) is the total number of indices, which range from 0 to (*m - n)*, meaning we have a total of **(m - n + 1) indices, and thus sublists.**

2.D. Cleaning Data

Q.2.D.i: For clean_with_deletion, describe in English a strategy for generating a deep copy of the original list while excluding rows that originally contained `None`.

To create a **deep copy** of the **original** list, I would first create a **new list**, like so:

clean_list = [data[0][:]]

This ensures that the new clean list will hold only the cleaned data (whatever we choose to append to it), without changing the original list. It also specifies that a **copy** of the **header** will be added to the list, since that should always remain the dataset.

Next, I would write a for loop that would go through all the rest of the rows, while skipping the header, which would look like this:

for row in data[1:]

Inside the loop, the best strategy would be to check if the row has any “None” values, and if so we ignore it and move on to the next row. Only when **the row does not have** any “**None**” values, we can **append** a **copy to** our “clean_list”.

Ultimately, the new “clean_list” would contain only a copy of the header row and a copy of the rows that do NOT have “None” in them, not impacting/mutating the original dataset.

Q.2.D.ii: For column_avgs, provide a Python expression for the number of columns in the data set. Explain your answer.

The simplest way to calculate the number of columns in any dataset is to calculate the **number of elements in a row**, considering that every row corresponds with a column.

In our case, we have a fixed header row that will always remain the same, which includes one string (column name) per column. If we calculate the **length** of the **header row**, we should get exactly the number of columns for the dataset. The code would be:

len(data[0])

We know our “data” is a list of lists, and every inner list is the row and every outer list is the column. Thus, data[0] would return the first row (header). Considering that in the assignment it is specified that each row will have the same number of elements in it, we can be safe in knowing that this is a safe and simple way to find the number of columns.

Q. 2.D.iii: For column_avgs, describe a strategy for building a list of all of the numeric elements in the column. You may include Python expression(s) to help explain your answer, but your description should be in English. If iteration is involved, you should clearly specify what you are iterating over *(i.e.* what structure and/or what indices, including whether they are inclusive or exclusive).

For this function, the first thing we need to do is build a new list with only the numeric elements of a column and then use a previous helper function to calculate the mean of that new list.

To collect all the numeric elements (not "None"), we **fix the column index** (col) and **iterate through each row,** skipping the header. For each row index from 1 to m - 1 (inclusive), we check **row[col]**. If the value is not "None", we append that number to a list. We repeat this process for all the columns with **range(0, *len(data[0])*)**. In the question before, we discussed we can find the number of columns using *len(data[0]), s*o that helps us write our first for loop:

```
for col in range(len(data[0])):
```

This will loop over each column index, from 0 to len(data[0] - 1). Then, we proceed with the inner loop, which goes from 1 to row m - 1 (last row):

```
data_values = []
for row in data[1:]:
    if row[col] is not None:
        data_values.append(row[col])
```

The code creates a new empty list, then goes through each row (except the header) and adds all the numeric values (i.e. not "None") to the new data_values list. At this point, our *data_values* list will contain all numeric elements which can be passed by the first helper function *arithmetic_mean* to compute the average.

Reflection

Q1. The main concepts this week included applying new Python concepts, specifically for lists and conditionals, within helper functions. We focused more on slicing, loops, conditionals and list manipulation (append method), and more helper function practice to help us avoid duplicate or repetitive code. These concepts are quite important in Data Science as they are the foundation for more complex functions (and sometimes longer ones, like the ‘bubble sort’ example in class) where all of these concepts have to be applied together to reach a solution.

Q2. The lists, conditionals, and specifically slicing in Python are crucial for visualization, which is a key part of Data Science and a big reason why Python is used. I think the knowledge I gained would help me think more conceptually about how to analyze, manipulate, or clean datasets in the future.

Q3. I think what I did well was revisiting the HW1 concepts before trying to go into HW2. That helped me understand how to clearly and effectively call helper functions to avoid repetition. Moreover, it was some class exercises that I revised that helped me with conditionals and lists, which was quite useful for the HW. However, if there was one thing I would do differently is to first write all the documentation/writeup and then focus on the code. While I did write a simple draft of the write-up, it was clear to me after a while that being more specific with the write up first would have helped me with a sort of plan for the code and take me less time to write it.

Q4. I believe that I am finally very comfortable with helper functions and can confidently say that I understand list manipulation quite well, which was also my favorite part of these past few weeks. I think I still need to practice a bit more with more complex exercises that involve applying all of these topics in an exercise, however, I think I can offer practical advice and tips when sharing this knowledge with a peer.