Midterm Assessment

ME 371: Data-Driven Problem Solving

Fall 2023

(Last Update: October 31, 2023)*

This document provides a short description about the midterm assessment for the course Data-Driven Problem Solving in Mechanical Engineering for Fall 2023. The project has two parts. The final submission should be two notebooks, one for each part. The deadline to submit the notebooks is Thursday November 2^{nd}. **Make sure you read the Submission section at the end of this document**.

- For Part I, ensure that you rely on the Python standard library and refrain from importing additional packages unless explicitly instructed in the problem. Unauthorized package usage may result in a deduction of up to 20% from your grade.
- Please be mindful of submission deadlines. Late submissions may incur a penalty of up to 10%.
- The primary objective of this assessment is to enhance your Python programming skills, not to practice crafting ChatGPT prompts. Avoid utilizing ChatGPT for assistance, as it can hinder your Python coding learning experience. If it is determined that ChatGPT was used, a deduction of up to 20% may apply to your grade.
- Collaboration with others is okay but please submit your original work.
- Ten points will be awarded for the organization and cleanliness of your notebook. This means it should not contain random print outputs, and each question, along with its corresponding answer, should be clearly presented with well-organized and clear code. Furthermore, all the requirements outlined in the 'Submission' section have been met.

Part I - Python Programming [35 points]

1. Import the random library into your notebook. The function $random.randint(a, b)$ returns a random integer between a and b, inclusive. Let's use this function to study the Law of Large Numbers.

   Write a function called $random\_avg(a, b, n = 100, seed = 1)$, where a , b , n and $seed$ are integers. Use the value of $seed$ to seed the random number generator, inside the function, using $random.seed(seed)$. This should be the first line of your function.

   Then using a for-loop, compute the sum of n random integers between a and b . Finally, divide the sum by n to compute the mean, and let your function return that value. You can use the following lines to check your function:

*If you see any mistakes or typos in this document, please email Dr. Masoumi at masoud.masoumi@cooper.edu

```python
# checking the function
assert random_avg(0, 10) == 5.28
assert random_avg(0, 10, n=1000) == 5.03
assert random_avg(0, 10, n=1000, seed=123) == 4.966
assert random_avg(0, 10, n=1_000_000, seed=123) == 4.997245
```

2. For this exercise you will need to import math library.

   Write a function that determines whether two lists of the same length that contain float values are "close" to each other. The function should have the following structure:

   ```python
   is_almost_equal(left, right, tolerance=1e-12)
   ```

   For example consider these lists: left=[1, 2, 3] and right=[1.0001, 2.0001, 2.9999] , then the function *is_almost_equal* will return True if $tolerance = 0.001$ because the root mean squared error, RMSE, is less than tolerance, i.e.

   ```python
   se = (1 - 0.0001) ** 2 + (2 - 2.0001) ** 2 + (3 - 2.9999) ** 2
   rmse = math.sqrt(se / 3) < 0.00
   ```

   but it would return False if tolerance was set to the default value.

   You can use the following test cases to check your function:

   ```python
   # checking the function
   assert is_almost_equal([1, 2, 3], [1.0001, 2.0001, 2.9999], tolerance=0.001)
   assert not is_almost_equal([1, 2, 3], [1.0001, 2.0001, 2.9999])
   ```

3. Write a function $prefix\_finder(my\_list, suffix)$ where my_list is a list of strings and $suffix$ is a string. The function iterates through the list of strings and identifies the strings that end with the provided suffix. It returns a list containing the portion of those strings up to the suffix. The comparisons should ignore case.

   You can use the following test cases to check your function:

   ```python
   # checking the function
   assert prefix_finder(["Forest", "Cat", "Fastest"], "EsT") == ["For", "Fast"]
   assert prefix_finder(["Forest", "Fastest", "Cat"], "AsT") == []
   ```

4. Consider the following list of 2-element tuples:

   ```python
   data = [
       (1, 23),
       (104, -3)
   ]
   ```

   Write a function named *tuple_to_dict* that converts the list of tuples into a list of dictionaries with the following structure: the first element in the tuple gets mapped to the dictionary key "col0" and the second element gets mapped to the key "col1":

```
[
    {"col0": 1, "col1": 23},
    {"col0": 104, "col1": -3}
]
```

You can use the following test cases to check your function:

```python
assert tuple_to_dict([(1, 40)]) == [{"col0": 1, "col1": 40}]
list_1 = [(9, 3), (12, -4), (0, 2)]
out_1 = [{"col0": 9, "col1": 3}, {"col0": 12, "col1": -4},
        {"col0": 0, "col1": 2}]

assert tuple_to_dict(list_1) == out_1
list_2 = []
out_2 = []
assert tuple_to_dict(list_2) == out_2
```

5. Consider a list of tuples, where the first element in the tuple stores a string identifier, and the remaining elements in the tuple are attributes for some entity that corresponds to the string identifier.

   Write a function named *attributes_to_dict* that takes as input the list of tuples and converts it to a dictionary, where each key of the dictionary stores the identifier and the value stores the corresponding list of the attributes. For instance, this tuple:

```python
("123xyz", 1, "green", "apple")
```

   is converted into this key-value pair:

```python
"123xyz": [1, "green", "apple"]
```

   And this tuple:

```python
("456uvw", 0)
```

   is converted into this key-value pair:

```python
"456uvw":[0]
```

   You can use the following test case to check your function:

```python
# checking the function
data = [
    ("123xyz", 1, "green", "apple"),
    ("456uvw", 0),
    ("209abc",),
    ("845lmn", 2, "blue", "ocean", "salmon"),
]
out = {
    "123xyz": [1, "green", "apple"],
    "456uvw": [0],
    "209abc": [],
    "845lmn": [2, "blue", "ocean", "salmon"],
}

assert attributes_to_dict(data) == out
```

6. Suppose you have a dictionary named COUNTS of string-integer pairs, for instance:

```python
COUNTS = {
    "u13": 10,
    "a41": 12,
    "c20": 2,
}
```

Construct a function named lookup that takes a list of strings as input. For each string in the list, the function should look up the corresponding value in the COUNTS dictionary and return the sum of the mapped values. If a string is not found in COUNTS, the corresponding value in the summation should default to 0.

You can use the following test cases to check your function

```python
# checking the function
assert lookup(["d23"]) == 0
assert lookup(["u13"]) == 10
assert lookup(["u13", "a41"]) == 22
assert lookup(["u13", "a41", "d23"]) == 22
```

7. Write a function that accepts a dictionary as input whose keys are strings and whose values are tuple pairs, where the first element of the tuple is a function, and the second element of the tuple is a list of numbers. For instance:

```python
data = {
 "c1X2": (max, [10, 0, -2]),
 "d4Y0": (sum, [-3, -2, -5, 4, 6]),
 "x1C9": (min, [0, -2, -9, 3]),
}
```

The function should be called *reducer*(*data*). It should iterate through each item in the dictionary and update the value with the result of applying the function in the tuple to the list of integers.

Use the following tests to validate your implementation:

```python
assert reducer(data) == { "c1X2": 10, "d4Y0": 0,  "x1C9": -9}

def square(my_list, q=2):
    s = 0
    for x in my_list:
        s += x ** q
    return s

data = {
 "r126": (square, [1, 0, -2]),
 "p1H2": (square, [-2, 1, -1, 0]),
}
assert reducer(data) == { "r126": 5, "p1H2": 6}
```

8. For this question, you can use numpy library.

   Given a NumPy array data of shape (m, n) containing random integers, write a function $custom\_max\_value\_indices(data)$ that returns the indices of the maximum values for each row in the array.

   The function should return a 1D NumPy array of shape $(m,)$ where each value represents the column index of the maximum value in the corresponding row.

   You can use the following test cases to check your function:

```python
data1 = np.array([[1, 2, 3], [4, 5, 6], [7, 8, 9]])
result1 = custom_max_value_indices(data1)
assert np.array_equal(result1, np.array([2, 2, 2]))

data2 = np.array([[3, 2, 1], [4, 5, 4], [7, 8, 8]])
result2 = custom_max_value_indices(data2)
assert np.array_equal(result2, np.array([0, 1, 1]))
```

9. You can use numpy library for this question.

   Write a Python function called $normalize\_array(arr)$ that takes a NumPy array as input and normalizes it, such that the mean of the normalized array is 0, and the standard deviation is 1.

Part II - Data Analysis via Manufacturing Industry Database [55 points]

The dataset for this part of the project and the detailed explanations of the variables in the dataset can be found here: `https://github.com/MasoudMiM/MECG_542/tree/master/Manufacturing_Industry_Database`. The data set is the NBER-CES Manufacturing Industry Database, containing the annual data from the United States manufacturing sector for the period from 1958 to 2018. The data set page provides all the information needed to become familiar with the variables and how they were collected or calculated. You can also see the document here for detailed information about the industries.

1. How many unique NAICS industries are represented in the dataset?
2. What is the total employment for each NAICS industry in the year 2018? Sort them in descending order.
3. Find the NAICS industry with the highest total payroll in 2018.
4. What was the overall trend in total capital expenditure (investment) from 1958 to 2018? Create a line plot to visualize this trend.
5. Which industry has the highest 5-factor TFP index (tfp5) in 2015?
6. How does the distribution of production worker wages (prodw) change from 1958 to 2018? Create a boxplot to visualize this.

7. Which industry experienced the highest increase in production worker hours (prodh) from 1990 to 2000?

8. For the industry with the highest total cost of materials (matcost) in 2015, calculate the percentage of matcost relative to total value added (vadd) for that year.

9. Create a bar plot to visualize the top 15 industries with the highest 4-factor TFP index (tfp4) in 2010.

10. Create a histogram to visualize the distribution of total real capital stock (cap) in the year 2010.

11. Calculate the industry with the highest average 5-factor TFP annual growth rate (dtfp5) over the years. Consider industries with data available for at least 20 years.

12. Create a line plot to visualize the trends in 5-factor TFP annual growth rates (dtfp5) for the top 5 industries with the highest average dtfp5 values over the years.

Submission

Your final submission will be two notebooks, one for part I and another for part II. When creating your notebooks:

- Make sure the notebook is well-organized and it includes the codes and the outputs.
- Make sure the answers to the questions asked are explicitly either printed or displayed in the notebook in the same order as the questions. Take advantage of the "print" options to provide an organized output.
- Make sure your name is in the notebook, preferably as a text in your notebook.
- Make sure you do not print very large chunks of data and variables with so much data. They just make your code unclean and hard to understand. They also mostly serve no purpose.
- Use proper variable names that help anyone reading your code understand what each variable might be representing.
- Comment your code so others can understand your thought process and debug your code in case of an error. Also, it helps you be able to easily use your code in the future.
- Once you are done writing your code, your notebook most probably includes code cells that you used for experimenting with some functions or testing some code lines. Organize your notebook and clean it up. Remove those unneeded sections/cells, restart the kernel, and then run the notebook from the beginning. Then submit the notebook with the results.
- Keep the length of code lines in your code short. Break the long lines into multiple lines. The suggested length of a code line is 79 characters.