Lab 9: Numpy and Randomness

Part 0: Submission/Reference Materials

Remember to read (and write!) good documentation and use the internet to find code examples. Finding and using appropriate/accurate reference materials is hard to teach directly, but tends to be what separates bad scientific programmers from good ones!

If you get stuck:

- Review the lecture material
- Check the docstrings of functions you are trying to use (hint: use ?function in jupyter)
- The pandas official documentation includes all of the commands you need to complete this week's practical https://pandas.pydata.org/docs/user_guide/index.html#user-guide
- Use online resources like stackoverflow, w3schools, realpython etc. (but make sure you don't blindly copy code without working out HOW it works).
- Looking up materials is totally fine but remember if you copy code (or autocomplete it) directly from any source, you MUST cite where you got it from in a comment next to the code.

Submit this assignment as a formatted notebook - include an explanation of your answers and make sure every function has a clear docstring that explains what it does, its arguments, and what it returns. This will be part of the grading of each of your answers.

Part 1: Set-Up and Random Number Generation

In today's assignment, you are going to use NumPy's random module to simulate, model, and analyze statistical data. You'll apply these methods to solve problems in optimization, hypothesis testing, and confidence interval calculation.

Let's start by importing the necessary libraries and creating our random number generator object with a random seed for reproducibility:

```
import numpy as np
rng = np.random.default_rng(42)
```

Q1 [4 points]: Create a function called `generate_sample_data` that takes the following parameters:

- distribution_type: A string specifying the distribution ('normal', 'uniform', 'exponential')

- sample_size: An integer specifying the number of samples to generate
- params: A dictionary containing the distribution parameters (e.g., for normal: 'loc' for mean and 'scale' for standard deviation)

The function should return a 1D NumPy array containing the generated random samples. Include at least 3 different distributions.

Q2 [4 points]: Create a function called `calculate_statistics` that takes a sample and returns the mean, median, standard deviation, and interquartile range. Test this function on three different samples (normal, uniform, exponential) of size 1000 generated using your function from Q1. Compare the calculated statistics with the theoretical values for each distribution.

Part 2: Monte Carlo Estimation

In this section, we'll use Monte Carlo methods to estimate the probability of complex events without having to solve them analytically. We do this by randomly simulating event occurring and measuring the outcomes over and over again. The distribution of outcomes can then be used to calculate the probability of specific observed outcomes.

Q3 [6 points]: We are interested in calculating the probability of rolling a total value or greater across multiple dice using Monte Carlo estimation (simulations). Create a function called `estimate_dice_probability` that calculates the probability of rolling >= a target value (`target`) on 1 or more dice (`n_dice`) by simulating a user-specified number of rolls (`num_sims`).

The function should:

- Accept parameters for the number of dice, the target total value, and the number of simulations
- Simulate rolling these dice as many times as the `num_sims` and calculating the total values each time
- Estimate the probability of rolling a total greater than or equal to the target value on that number of dice (i.e., what proportion of simulated rolls resulted in a total >= the `target`)

Use your function to estimate the probability of getting a sum greater than 8 when rolling three six-sided dice. How does this compare to the exact probability (115 possible rolls on 3 dice >= 8 out of 216 possible outcomes = 53.7%)

Part 3: Random-search Optimization

Q4 [6 points]: Create a function called `optimize_function` that uses random search to find the global minimum of a complex function which takes 2 parameters (i.e., if z = f(x,y) find the values of x and y that result in the lowest possible value for z). Your optimisation function should:

- Accept the following parameters:
    - a function to minimize with 2 parameters (x,y) that returns a single value (z)
    - bounds of the x,y parameter space (i.e., if n = 10, then you are searching for the optimal x between x=0 and x=10 and the optimal y between y=0 and y=10)

    - How many random x and y values to try
- Generate randomly distributed x and y values within the bounds
- Calculate the value of z for your input function for each of these x and y values
- Return the value of x and y that resulted in the lowest value of z

You can test your optimisation function by finding the `x` and `y` that cause `test_function` to return the lowest value :

```
def test_function(x, y):
    """
    A complex 2D function with multiple local minima.

    Parameters:
    x (float or array): Coordinate to evaluate
    y (float or array): Coordinate to evaluate
    Returns:
    float or array: Function value at the given coordinate
    """
    return x**2 * np.sin(5 * x/y) + 0.2 * np.cos(16 * x) * y
```

Search within the bounds (0, 10), using 10000 random points and report the found minimum value and the associated x and y values.

Part 4: Hypothesis Testing Using Simulation

In this section, you'll use random sampling to perform hypothesis tests when analytical solutions are difficult to derive.

Q5 [6 points]: Create a function called `permutation_test` that performs a two-sample permutation test for the difference in means. The function should:

- Accept two samples of equal size (i.e., same number of values in each) and the number of permutations to perform
- Calculate the observed difference in means between the two samples
- Randomly permute the two samples to shuffle values between them (this can be done by combining both samples into one long array, shuffling this long array, and then splitting the long array in half).
- Calculate the difference in mean between these 2 permuted sample arrays for each permutation
- Return the p-value (the proportion of permuted differences that are more extreme than the observed difference)

Test your function on samples generated from:

1. Two normal distributions with the same mean ($\mu=0$) but different standard deviations ($\sigma_1=1$, $\sigma_2=2$)
2. Two normal distributions with different means ($\mu_1=0$, $\mu_2=0.5$) and the same standard deviation ($\sigma=1$)

Use a sample size of 50 for each group and 1000 permutations.

Part 5: Confidence Intervals Using Simulation

In this section, you'll use simulation methods to construct confidence intervals for statistics when analytical solutions are complex or unavailable.

Q6 [6 points]: Create a function called `bootstrap_ci` that calculates bootstrap confidence intervals for the mean of a sample. The function should:

- Accept a sample (i.e., an array of numbers), the confidence level (e.g., 0.95), and the number of bootstrap resamples
- Generate bootstrap resamples by sampling with replacement from the original sample
- Calculate the mean of each bootstrap resample
- Return the confidence interval using the percentile method

Test your function by calculating 95% confidence intervals for the mean of 1000 numbers drawn from a gamma distribution with a shape of 3.5 and a scale of 2.2.