Random Sequence Generation & Exploratory Analysis

DNAe Coding Test | Abhiraj Arora

DNAe

Background & Purpose

- The purpose of the exercise was to generate a DNA sequence of 100 bases and perform exploratory analysis on this dataset.

- This was done in Jupyter Notebook using Python, and used the libraries numpy, matplotlib and random.

```
# Importing the required libraries for this analysis:
    # Random library is imported for random choosing of nucleotides, and positions for introducing errors
    # Numpy is imported for mathematical analysis
    # Matplotlib.pyplot is imported for graph plotting and visualisation

import random
import numpy as np
import matplotlib.pyplot as plt
```

Methodology

The first step was to create a DNA sequence of length 100 bases, using the 'random' library, where the G and C nucleotides comprise of 60% of the sequence, and A and T comprise the remaining 40%.

The below function defines the bases, and the parameters for the GC content and remaining AT content, and combines them to create the template sequence, which will be used to introduce errors.

```python
# Creating a function to create a DNA template sequence
def create_template_sequence(length, gc_content):

    # Calculating the number of GC nucleotides required (60% content will be specified later)
    amount_gc = int(length * gc_content)

    # Calculate the number of AT nucleotides required (the length of the sequence (100) subtract the GC amount)
    amount_at = length - amount_gc

    # Creating a list of all the nucleotides (Adenine (A), Cytosine(C), Guanine(G) and Thymine(T))
    bases = ['G', 'C', 'A', 'T']

    # Creating the sequence with GC nucleotides only
    gc_sequence = ''.join(random.choice(bases[:2]) for i in range(amount_gc))

    # Creating the sequence with AT nucleotides only
    at_sequence = ''.join(random.choice(bases[2:]) for i in range(amount_at))

    # Joining the GC and AT sequences together to create the template sequence
    template_sequence = gc_sequence + at_sequence

    # Returning the template sequence
    return template_sequence
```

Methodology

The code to the right lists the error types (insertions, deletions and mismatches), and a loop to introduce them with If/Else If statements to specify the appropriate action based on which error type is randomly chosen.

The error positions are also tracked per error type.

```python
# Creating a function to add random errors to a sequence
def add_errors(sequence, error_rate):

    # Calculating how many errors to add based on the error rate
    amount_errors = int(len(sequence) * error_rate)

    # Listing the error types
    errors = ['deletion', 'insertion', 'mismatch']

    # Creating dictionaries to track the position of errors
    error_positions = {'deletion': [], 'insertion': [], 'mismatch': []}

    # Creating a loop to introduce errors
    for i in range(amount_errors):

        # Choosing a random error type (insertion/deletion/mismatch)
        error_type = random.choice(errors)

        # Choosing a random position in the sequence
        position = random.randint(0, len(sequence) - 1)

        # Specifying If/Else If statements to perform the appropriate functions based on the error type chosen
        # For instance, if the random error type chosen in the loop above is an insertion, random nucleotides need to be add

        if error_type == 'deletion':
            # If deletion is chosen, remove nucleotides at the chosen (random) position
            sequence = sequence[:position] + sequence[position + 1:]

        elif error_type == 'insertion':
            # If insertion is chosen, insert nucleotides at the chosen (random) position
            base = random.choice('ACGT')
            sequence = sequence[:position] + base + sequence[position:]

        elif error_type == 'mismatch':
            # If mismatch is chosen, replace with nucleotide that is different from the original one
            bases = [base for base in 'ACGT' if base != sequence[position]]
            base = random.choice(bases)
            sequence = sequence[:position] + base + sequence[position + 1:]

        # Tracking the position where the error(s) occurred
        error_positions[error_type].append(position)

    # Return the error-ridden sequence(s) and the respective error positions
    return sequence, error_positions
```

Methodology

```python
# Main function to call the previous subtasks, printing the sequences and plotting & displaying the required graphs
def main():
    # Generating a template sequence with specified length and GC content
    template_sequence = create_template_sequence(100, 0.6)

    # Defining the required error rates (2%, 5% and 10%)
    error_rates = [0.02, 0.05, 0.1]

    # Looping through each error rate
    for error_rate in error_rates:

        # Creating lists to store error-ridden sequences and respective error positions
        sequences_with_errors = []
        error_positions_by_type = {'deletion': [], 'insertion': [], 'mismatch': []}

        # Creating a loop to generate 100 sequences with errors and analyze
        for i in range(100):
            sequence_with_errors, error_positions = add_errors(template_sequence, error_rate)
            sequences_with_errors.append(sequence_with_errors)

            # Storing error positions by error type
            for error_type, positions in error_positions.items():
                error_positions_by_type[error_type].extend(positions)

        # Printing sequences with errors
        print(f"Error Rate: {error_rate}")
        for sequence in sequences_with_errors:
            print(sequence)

        # Printing error positions
        print("Error Positions:")
        for error_type, positions in error_positions_by_type.items():
            print(f"{error_type}: {positions}")
```

The task required a main function to be implemented, and the calling of the previous functions for the subtasks.

The main function is defined here, where the GC content (60%) for the template sequence is specified, and so are the error rates for each set of sequences.

A loop is made to create a set of 100 sequences per error rate, track the error positions per error type and print them.

Methodology

The following graphs are to be plotted, per error rate:

- Error Position Distribution
- Sequence Length Distribution
- GC Content Distribution
- Homopolymer Ratio vs Sequence Length (per Nucleotide (A,T,C,G)
- Homopolymer Positions (per Nucleotide)

```
# Plotting error position distribution (position vs frequency)
plt.figure()
for error_type, positions in error_positions_by_type.items():
    plt.hist(positions, bins=range(len(template_sequence)+1), alpha=0.5, label=error_type)
plt.title(f"Error Position Distribution (Error Rate: {error_rate})")
plt.xlabel("Position")
plt.ylabel("Frequency")
plt.legend()
plt.show()

# Plotting sequence length distribution per error rate (sequence length vs frequency)
sequence_lengths = [len(sequence) for sequence in sequences_with_errors]
plt.figure()
plt.hist(sequence_lengths, bins=range(min(sequence_lengths), max(sequence_lengths)+2), alpha=0.5)
plt.title(f"Sequence Length Distribution (Error Rate: {error_rate})")
plt.xlabel("Sequence Length")
plt.ylabel("Frequency")
plt.show()

# Plotting GC content per error rate (amount of GC nucleotide content vs frequency)
gc_contents = [(sequence.count('G') + sequence.count('C')) / len(template_sequence) for sequence in sequences_with_e
plt.figure()
plt.hist(gc_contents, bins=np.linspace(0, 1, 21), alpha=0.5)
plt.title(f"GC Content Distribution (Error Rate: {error_rate})")
plt.xlabel("GC Content")
plt.ylabel("Frequency")
plt.show()
```

Methodology

```
        # Plotting homopolymer ratios relative to sequence length, per nucleotide & error rate
        homopolymer_ratios = []
        for sequence in sequences_with_errors:
            homopolymer_counts = {base: 0 for base in 'ACGT'}
            for base in 'ACGT':
                homopolymer_counts[base] = max([len(run) for run in sequence.split(base)])
            total_homopolymers = sum(homopolymer_counts.values())
            homopolymer_ratios.append({base: count / len(sequence) for base, count in homopolymer_counts.items()})

        bases = 'ACGT'
        for i, base in enumerate(bases):
            plt.figure()
            ratios = [ratios[base] for ratios in homopolymer_ratios]
            plt.scatter(sequence_lengths, ratios, alpha=0.5)
            plt.title(f"Homopolymer Ratio ({base}) vs. Sequence Length (Error Rate: {error_rate})")
            plt.xlabel("Sequence Length")
            plt.ylabel("Homopolymer Ratio")
            plt.show()

        # Plotting homopolymer positions per error rate, for each nucleotide
        homopolymer_positions = {base: {'before': [], 'after': []} for base in 'ACGT'}
        for sequence in sequences_with_errors:
            for base in 'ACGT':
                runs = [run for run in sequence.split(base) if len(run) > 1]
                for run in runs:
                    start = sequence.index(run)
                    end = start + len(run) - 1
                    homopolymer_positions[base]['before'].append(start)
                    homopolymer_positions[base]['after'].append(end)

        for base in bases:
            plt.figure()
            plt.hist(homopolymer_positions[base]['before'], bins=range(len(template_sequence)+1), alpha=0.5, label='Before')
            plt.hist(homopolymer_positions[base]['after'], bins=range(len(template_sequence)+1), alpha=0.5, label='After')
            plt.title(f"Homopolymer Positions ({base}) (Error Rate: {error_rate})")
            plt.xlabel("Position")
            plt.ylabel("Frequency")
            plt.legend()
            plt.show()

if __name__ == "__main__":
    main()
```

Continued from previous page

The following graphs are to be plotted, per error rate:

- Error Position Distribution
- Sequence Length Distribution
- GC Content Distribution
- Homopolymer Ratio vs Sequence Length (per Nucleotide (A,T,C,G)
- Homopolymer Positions (per Nucleotide)

Finally, the main() function is called, displaying the sequences, the error positions and the relevant graphs as specified as above.