Department of Science

School of Health and Life Sciences

MSc. BIOINFORMATICS

ONWUCHEKWA BENJAMIN CHUKWUMA

E4294137

PYTHON

SCI4019-N- BF1-2024

Module Leaders Name

Dr Mengyuan Wang

Data Analytics Assessment Report.

13th January 2025.

Declaration of Generative AI:

I DID use Generative AI technology in the development, or editing of this assignment.

Word Count. 2940

Contents

Task one: Basics of Python

Task 1.1: DNA sequence Analysis

1.1.1 Task Overview

The aim of this task is to write a Python program that processes a DNA sequence provided by the user. The program should take input interactively and check that the sequence contains only valid nucleotide bases (A, T, C, G). To make it case insensitive, the input should be converted to uppercase before processing. The main tasks include counting how many times each nucleotide appears and calculating the GC-content, which is the percentage of Guanine (G) and Cytosine (C) bases in the sequence. This task tests the ability to use basic Python programming skills. It involves using loops to go through sequences, conditionals to check the input, and dictionaries to store and handle the counts of nucleotides. It also requires working with strings and performing calculations to find the GC-content. The assessment focuses on both programming skills and accuracy when working with biological data.

1.1.2 Program design

The program is designed to:

- Prompt the user to input a DNA sequence.
- Validate the sequence to ensure it only contains valid nucleotide characters (A, T, C, G).
- Convert the input to uppercase to account for any lowercase characters
- Standardise the input using the .upper() method and remove any whitespace.
- Use a dictionary to store the counts of each nucleotide (A, T, C, G).
- Calculate the GC-content as a percentage and format it to two decimal places.
- Display the nucleotide counts and GC-content to the user.

```
DEF analyze_dna_sequence()
    WHILE True
        Prompt user to input a DNA sequence
        Remove spaces and convert the sequence to uppercase
        IF input is empty THEN
            Display error message
            CONTINUE
        END IF
        IF sequence contains invalid characters (not A, T, C, G) THEN
            Display error message
            CONTINUE
        END IF
        Create dictionary to store counts of A, T, C, G
        FOR each nucleotide in ['A', 'T', 'C', 'G']
            Count occurrences of the nucleotide and store in the dictionary
        END FOR
        Calculate GC-content as (G + C) / total length * 100
        Display counts of A, T, C, G
        Display GC-content percentage
        BREAK
    END WHILE
END DEF
```

Figure 1.1 Pseudocode of program

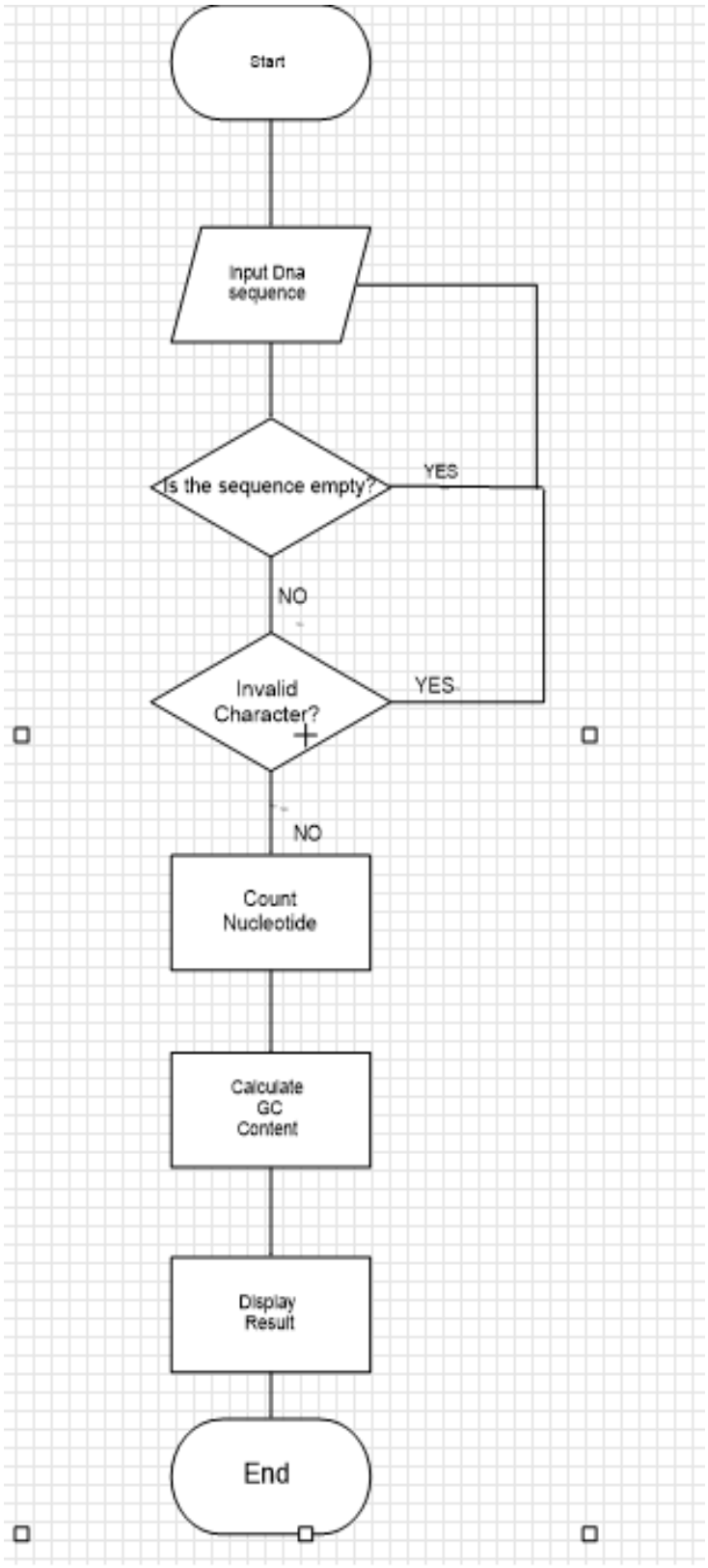

Figure1.2: Flowchart of the program

1.1.3 Program execution demonstration

This program was executed using Jupyter Notebook from the Anaconda distribution which is a Python compatible environment. Below is the demonstration of the program execution with input and outputs of the program.

```python
# Define the function 'analyze_dna_sequence' which will perform the DNA sequence analysis
def analyze_dna_sequence():

    # use the while loop to Start an infinite loop to repeatedly ask for input until valid input is provided
    while True:

        # use the input function to ask the user to input a DNA sequence.
        #The .strip() removes any extra spaces at the start and end of the input.
        dna_input = input("Enter a DNA sequence (only A, T, C, G): ").strip()

        # Check if the user input is empty (i.e., if the user pressed Enter without typing anything)
        if not dna_input:
            # If the input is empty, print an error message and prompt the user to try again
            print("Error: The sequence cannot be empty. Try again.")
            continue  # Restart the loop to ask for input again

        # Convert the input to uppercase and remove any spaces
        dna_input = dna_input.upper().replace(" ", "")

        # Define a set of valid nucleotide bases (A, T, C, G) in DNA
        valid_nucleotides = {'A', 'T', 'C', 'G'}

        # Find any characters in the input that are not in the set of valid nucleotides using set.
        invalid_characters = set(dna_input) - valid_nucleotides

        # If there are invalid characters, raise an error with a message showing the invalid characters
```


```python
# Define the function 'analyze_dna_sequence' which will perform the DNA sequence analysis
def analyze_dna_sequence():

    # use the while loop to Start an infinite loop to repeatedly ask for input until valid input is provided
    while True:

        # use the input function to ask the user to input a DNA sequence.
        #The .strip() removes any extra spaces at the start and end of the input.
        dna_input = input("Enter a DNA sequence (only A, T, C, G): ").strip()

        # Check if the user input is empty (i.e., if the user pressed Enter without typing anything)
        if not dna_input:
            # If the input is empty, print an error message and prompt the user to try again
            print("Error: The sequence cannot be empty. Try again.")
            continue  # Restart the loop to ask for input again

        # Convert the input to uppercase and remove any spaces
        dna_input = dna_input.upper().replace(" ", "")

        # Define a set of valid nucleotide bases (A, T, C, G) in DNA
        valid_nucleotides = {'A', 'T', 'C', 'G'}

        # Find any characters in the input that are not in the set of valid nucleotides using set.
        invalid_characters = set(dna_input) - valid_nucleotides

        # If there are invalid characters, raise an error with a message showing the invalid characters
```


Figure 1.3 input of the program

Figure 1.4 Output of the program

1.1.4 Program test, debug and optimization

Testing: The program was tested with various types of input to ensure it handles both valid and invalid cases correctly**.**

```
Enter a DNA sequence (only A, T, C, G):  TGGCGCCCCCGACAGCCATGCGTACGGCAGGC

        DNA Sequence Analysis:
        Number of occurrences of 'A': 5
        Number of occurrences of 'T': 3
        Number of occurrences of 'C': 13
        Number of occurrences of 'G': 11
        GC-content: 75.00%
```

```
Enter a DNA sequence (only A, T, C, G):  atcgatcgggccaatcta

        DNA Sequence Analysis:
        Number of occurrences of 'A': 5
        Number of occurrences of 'T': 4
        Number of occurrences of 'C': 5
        Number of occurrences of 'G': 4
        GC-content: 50.00%
```

```
Enter a DNA sequence (only A, T, C, G):  GGCGGCAGCGGGGGGA324562
---------------------------------------------------------------------------
AssertionError                            Traceback (most recent call last)
Cell In[9], line 52
     49         break  # End the loop since the sequence was valid and processed
     51 # Run the function to analyze a DNA sequence
---> 52 analyze_dna_sequence()

Cell In[9], line 27, in analyze_dna_sequence()
     24 invalid_characters = set(dna_input) - valid_nucleotides
     26 # If there are invalid characters, raise an error with a message showing the invalid characters
---> 27 assert not invalid_characters, f"Invalid characters found: {invalid_characters}. Only A, T, C, and G are allowed."
     29 # Count how many times each base ('A', 'T', 'C', 'G') appears in the DNA sequence
     30 base_counts = {base: dna_input.count(base) for base in valid_nucleotides}

AssertionError: Invalid characters found: {'5', '6', '2', '3', '4'}. Only A, T, C, and G are allowed.
```

```
Enter a DNA sequence (only A, T, C, G):
Error: The sequence cannot be empty. Try again.
Enter a DNA sequence (only A, T, C, G):
```

Figure 1. 5 program tested with a different DNA sequence(Valid and invalid Input) and empty input.

Debug Process:

• Fixed an issue with lowercase input by converting all characters to uppercase.

• Improved error handling to repeatedly prompt the user until a valid sequence is provided.

• Enhanced GC-content formatting to always show two decimal places.

Optimisation:

• Used a dictionary to efficiently store and retrieve nucleotide counts.

• Applied Python's f-string formatting for precise and accurate display of GC-content percentage.

1.1.5 Task summary

This Python program analyses DNA sequences by counting nucleotides, calculating GC-content, and validating input. It handles errors accurately, uses efficient data structures, and ensures accurate results while limiting to valid nucleotide sequences**.**

1.1.5.1 Program Advantages

• Easy to use, with prompts for input and clear error messages.

• Works with both uppercase and lowercase letters.

• Calculates nucleotide counts and GC-content accurately.

1.1.5.2 Program Limitations

• Only works with nucleotide sequences containing A, T, C, and G.

• Does not handle special characters or spaces in the input.

• Input checks are basic and may not catch complex mistakes like non-biological sequences.

Task 1.2: DNA Sequence Analysis

1.2.1 Task Overview

This task requires writing a Python program to perform three key operations on a given DNA sequence of 475 characters. First, the program will count the frequency of each nucleotide ('A', 'T', 'C', 'G'). Second, it will generate the reverse complement of the DNA sequence. Finally, it will identify the starting indices of the subsequence 'ATC' within the DNA sequence. The focus will be on efficiently handling string manipulations, ensuring accuracy, and employing basic Python programming

techniques. These operations will test the ability to process and analyse biological data effectively using fundamental coding skills.

1.2.2 Program design

The program design is divided into three main steps:

- **Frequency Count**: A dictionary is used to stores the count of each nucleotide ('A', 'T', 'C', 'G'), The program then loops through the sequence to update these counts.
- **Reverse Complement:** A dictionary maps each nucleotide to its complement ('A' - 'T', 'C' - 'G'). The reverse complement is created using list comprehension and the **reversed()** function.
- **Find Subsequence Occurrences**: The **find()** method is used in a loop to find starting positions of the subsequence ('ATC'). The indices are stored in a list for easy access.

```
# For Frequency Count:
DEF analyze_dna_sequence()
    Prompt user to input a DNA sequence
    Convert the sequence to uppercase
    Create a dictionary nucleotide_count
    FOR each base in dna_sequence
        IF base is in nucleotide_count THEN
            Increment nucleotide_count[base] by 1
        END IF
    END FOR
    Display nucleotide_count
```

Figure 1.2.1 pseudocode for frequency count

```
#For reverse compliment
   Create a dictionary complement
   Initialize reverse_complement_sequence as an empty string
   FOR each base in reversed(dna_sequence)
       IF base is in complement THEN
           Add complement[base] to reverse_complement_sequence
       END IF
   END FOR
   Display reverse_complement_sequence
```

Figure 1.2.2 pseudocode for reverse compliment

```
#For finding the occurence of a subsequence
    Define subsequence = 'ATC'
    Initialize subsequence_indices as an empty list
    Set start to 0
    WHILE start is less than the length of dna_sequence
        Set start to dna_sequence.find(subsequence, start)
        IF start equals -1 THEN
            BREAK
        END IF
        Add start to subsequence_indices
        Increment start by 1
    END WHILE
    Display subsequence_indices
END DEF
```

Figure 1.2.3 pseudocode for finding the occurrence of a subsequence

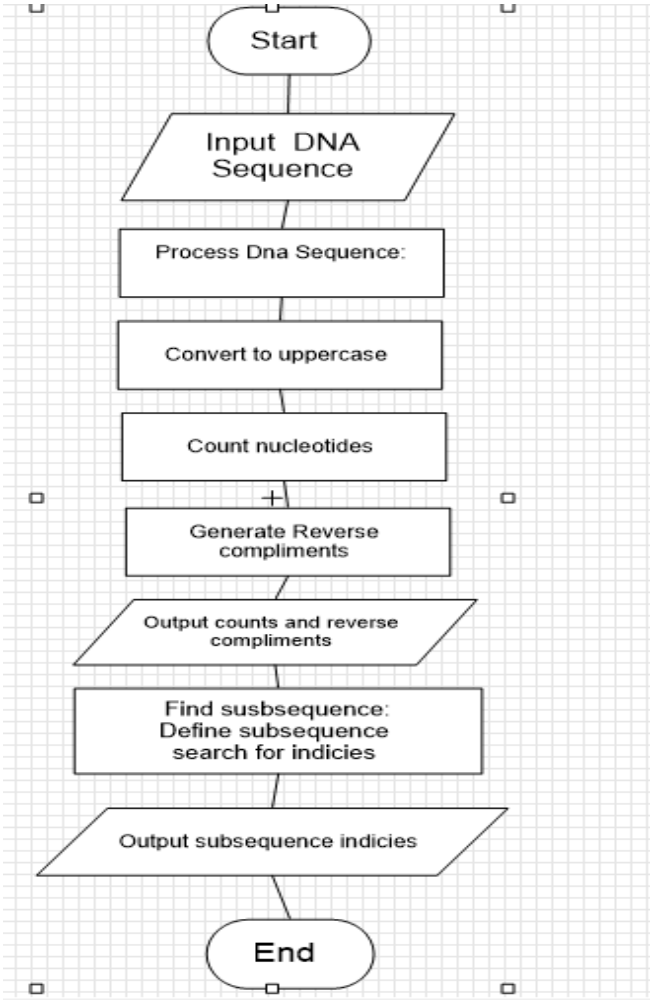

Figure 1.2.4 flowchart of the program

1.2.3 Program execution demonstration

This program was executed using Jupyter Notebook from the Anaconda distribution Python compatible environment. Below is the demonstration of the program execution with sample inputs.

```
dna_sequence ="ATCGATCGAACATCGATCATCATCGATCGAACATCGATCATCGATCGAAATCGATCGAACCGAACGATCGATCGAACATATCGATATCGATCGAACCGATCGAACATCGAACCATCGATCGAAATCG ATCGAACCGAA

# Convert the sequence to uppercase to prevent any case sensitivity issues
dna_sequence = dna_sequence.upper()

# count the frequency of each nucleotide (A, T, C, G) and Initialize a dictionary to hold nucleotide counts
nucleotide_count = {'A': 0, 'T': 0, 'C': 0, 'G': 0}

# Count each nucleotide in the sequence
for base in dna_sequence:
    if base in nucleotide_count:
        nucleotide_count[base] += 1

# Print the frequency of each nucleotide
print(" The Nucleotide frequencies are:", nucleotide_count)


# Step 2: Generate the reverse complement of the DNA sequence
# use Dictionary for base pairings
complement = {'A': 'T', 'T': 'A', 'C': 'G', 'G': 'C'}

# Using list comprehension to generate the reverse complement, only including valid bases
reverse_complement_sequence = ''.join(complement[base] for base in reversed(dna_sequence) if base in complement)

# Print the reverse complement of the sequence
```

```
# Print the reverse complement of the sequence
print("\n The Reverse complement of the DNA sequence is:", reverse_complement_sequence)


# Step 3: Find subsequence occurrences in the DNA sequence

subsequence = 'ATC'
subsequence_indices = []

# Iterate over the sequence and find subsequence that matches
start = 0
while start < len(dna_sequence):
    start = dna_sequence.find(subsequence, start)  # Find subsequence from current position
    if start == -1:
        break  # No more occurrences
    subsequence_indices.append(start)  # Store the start index of the match
    start += 1  # Move past the current match to search for further occurrences

# Print the indices of the subsequence
print(f"\nSubsequence '{subsequence}' found at indices:", subsequence_indices)
```

Figure 1.2.5 input of the program

```
 The Nucleotide frequencies are: {'A': 170, 'T': 90, 'C': 130, 'G': 85}

 The Reverse complement of the DNA sequence is: GATGGTTCGATGTTCGATCGGTTCGATCGATATCGATATGTTCGATCGATCGTTCGGTTCGATCGATTTCGATCGATGATCGATGTTCGATCGATGATGGGATGG
TTCGATGTTCGATCGGTTCGATCGATATCGATATGTTCGATCGATCGTTCGGTTCGATCGATTTCGATCGATGATCGATGTTCGATCGATTTCGATGTTCGATGATGGTTCGATGTTCGATCGGTTCGATCGATATCGATATGTTCGATCGAT
CGTTCGGTTCGATCGATTTCGATCGATGATCGATGTTCGATCGATCGGTTCGATCGATATCGATATGTTCGATCGATCGTTCGGTTCGATCGATTTCGATCGATGGTTCGATGTTCGATCGGTTCGATCGATATCGATATGTTCGATCGATCG
TTCGGTTCGATCGATTTCGATCGATGATCGATGTTCGATCGATGATGATCGATGTTCGATCGAT

Subsequence 'ATC' found at indices: [0, 4, 11, 15, 18, 21, 25, 32, 36, 39, 43, 49, 53, 66, 70, 79, 85, 89, 98, 105, 113, 117, 123, 128, 141, 145, 154, 16
0, 164, 173, 177, 184, 188, 191, 195, 201, 205, 218, 222, 231, 237, 241, 250, 258, 266, 269, 276, 282, 286, 293, 297, 300, 304, 310, 314, 327, 331, 340,
346, 350, 359, 366, 374, 379, 387, 394, 398, 401, 405, 411, 415, 428, 432, 441, 447, 451, 460, 467, 475]
```

Figure 1.2.6 output of the program

1.2.4 Program test, debug and optimization

Testing: The program was tested with multiple DNA sequences to ensure the accuracy of frequency counts, reverse complement generation, and subsequence search. Different subsequences were used to validate the search function.

Debugging

- The program handled invalid characters by ensuring that only 'A', 'T', 'C', and 'G' are counted.
- Ensured that the reverse complement function works even if the input contains lowercase letters by converting the sequence to uppercase.

Optimization

- Used a dictionary for nucleotide counts to allow efficient data access.
- Used a list comprehension for concise and efficient reverse complement generation.
- Improved the subsequence search function by using a loop and the **find()** method.

1.2.5 Task summary

This Python program analyses a DNA sequence by counting nucleotides, creating the reverse complement, and finding subsequence positions. It uses dictionaries and lists efficiently but doesn't handle invalid inputs well.

1.2.5.1 Program advantages

- Good use of data structures (dictionaries and lists).
- Clear and concise code with meaningful variable names.
- Accurate analysis of the DNA sequence, including nucleotide counts, reverse complement generation, and subsequence search.

1.2.5.2 Program limitations

- It assumes the input DNA sequence is already validated and cleaned of any non-nucleotide characters.
- The program currently does not handle sequences with invalid characters very well. Adding error handling for unexpected inputs would be beneficial.

Task 1.3 DNA Sequence Mutation Analysis

1.3.1 Task Overview

The objective of this task is to write a Python program that analyses a given DNA sequence of 700 characters by validating it, counting nucleotide runs, and identifying possible mutation positions. The program checks for valid nucleotides ('A', 'T', 'C', 'G') and corrects the sequence if any invalid characters are found. It then counts the longest consecutive run of a specified nucleotide ('A') and finds positions where a mutation to a target nucleotide ('G') can occur.

1.3.2 Program design

The program works in three main steps:

- **DNA Sequence Validation:** A set is used to effectively validate nucleotide bases, with a for loop iterating through the DNA sequence. Non-valid characters are filtered out.

- **Counting Nucleotide Runs:** A while loop, combined with conditional checks, determines the longest consecutive occurrences of a specified nucleotide(A).
- **Finding Mutation Positions:** using a for loop to Identify positions where a mutation could create a specific nucleotide (e.g., G) and shows these positions. using index-based comparisons

```
# for DNA Sequence Validation
START
    INPUT dna_sequence
    CONVERT dna_sequence TO uppercase
    INITIALISE nucleotide_count AS {'A': 0, 'T': 0, 'C': 0, 'G': 0}
    FOR EACH base IN dna_sequence
        IF base EXISTS IN nucleotide_count THEN
            INCREMENT nucleotide_count[base] BY 1
        END IF
    END FOR
    PRINT nucleotide_count
```

```
# for Counting Nucleotide Runs
    INITIALISE complement AS {'A': 'T', 'T': 'A', 'C': 'G', 'G': 'C'}
    INITIALISE reverse_complement_sequence AS empty string

    FOR EACH base IN REVERSED dna_sequence
        IF base EXISTS IN complement THEN
            ADD complement[base] TO reverse_complement_sequence
        END IF
    END FOR
    PRINT reverse_complement_sequence
```

```
# for finding the position of a mutation
    INITIALISE subsequence AS 'ATC'
    INITIALISE subsequence_indices AS empty list
    INITIALISE start AS 0

    WHILE start < LENGTH(dna_sequence)
        SET start TO POSITION OF subsequence IN dna_sequence STARTING FROM start
        IF start == -1 THEN
            BREAK
        END IF
        ADD start TO subsequence_indices
        INCREMENT start BY 1
    END WHILE

    PRINT subsequence_indices
```

Figure 1.3.1 showing the pseudocode for DNA mutation analysis

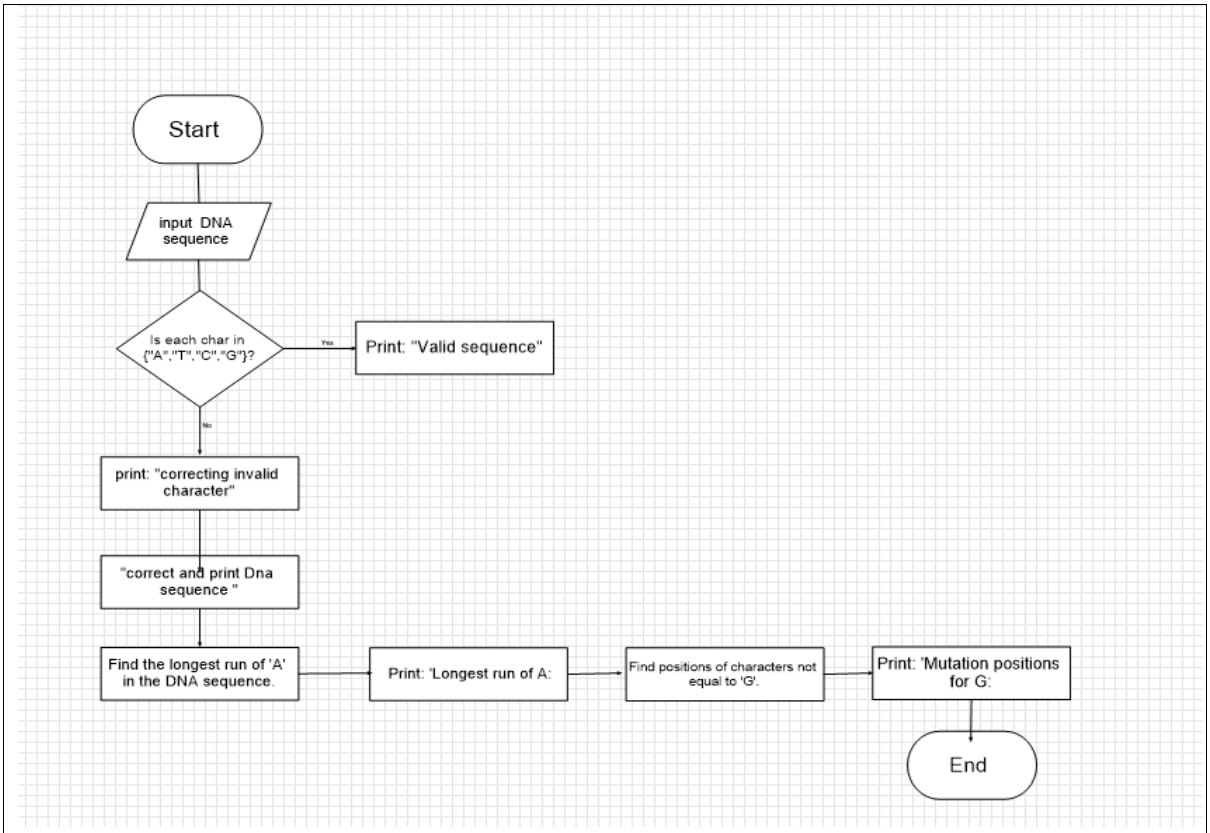

Figure 1.3.2 flow chart of the program

1.3.3 Program execution demonstration

```python
# Input DNA sequence (Length 700 as provided in the problem statement)
dna_sequence = ("ACTAACATCCAGTTAATAATTTATACATAGTTTAAGAATATACTACATTAAAAATACGAACTAAGACTGTATTACGCTCCCATCTAATCTATATTTTGTGTTCTATAAGATCCTTTATTAATATAG TATACACTG

# Step 1: Validate the DNA sequence
valid_bases = {'A', 'T', 'C', 'G'}
is_valid = True

# Check each character in the sequence
for char in dna_sequence:
    if char not in valid_bases:
        is_valid = False  # Invalid character found
        break

# If the sequence is invalid, correct it by removing non-valid characters
if is_valid:
    print("The DNA sequence is valid.")
else:
    print("Invalid characters found in the DNA sequence. Correcting...")
    corrected_sequence = ""
    for char in dna_sequence:
        if char in valid_bases:
            corrected_sequence += char  # Add only valid characters to the corrected sequence
    dna_sequence = corrected_sequence
    print("Corrected DNA sequence.")
```

```python
# Step 2: Count the longest run of 'A'
target_nucleotide = 'A'
longest_run = 0
current_run = 0
i = 0  # Initialize counter

# Use a while loop to check each character
while i < len(dna_sequence):
    if dna_sequence[i] == target_nucleotide:
        current_run += 1
        if current_run > longest_run:
            longest_run = current_run  # Update the longest run if needed
    else:
        current_run = 0  # Reset if the character is not 'A'
    i += 1  # Move to the next character

print(f"\nThe longest consecutive run of '{target_nucleotide}' is: {longest_run}")

# Step 3: Find possible mutation positions for 'G'
target_mutation_nucleotide = 'G'
mutation_positions = []
```

```python
# Step 3: Find possible mutation positions for 'G'
target_mutation_nucleotide = 'G'
mutation_positions = []

# Check each character's position
for i in range(len(dna_sequence)):
    if dna_sequence[i] != target_mutation_nucleotide:
        mutation_positions.append(i)  # If it's not 'G', it's a possible mutation position

print(f"\nPossible mutation positions for '{target_mutation_nucleotide}':")
print(mutation_positions)
```

Figure 1.3.3 showing the input of the program

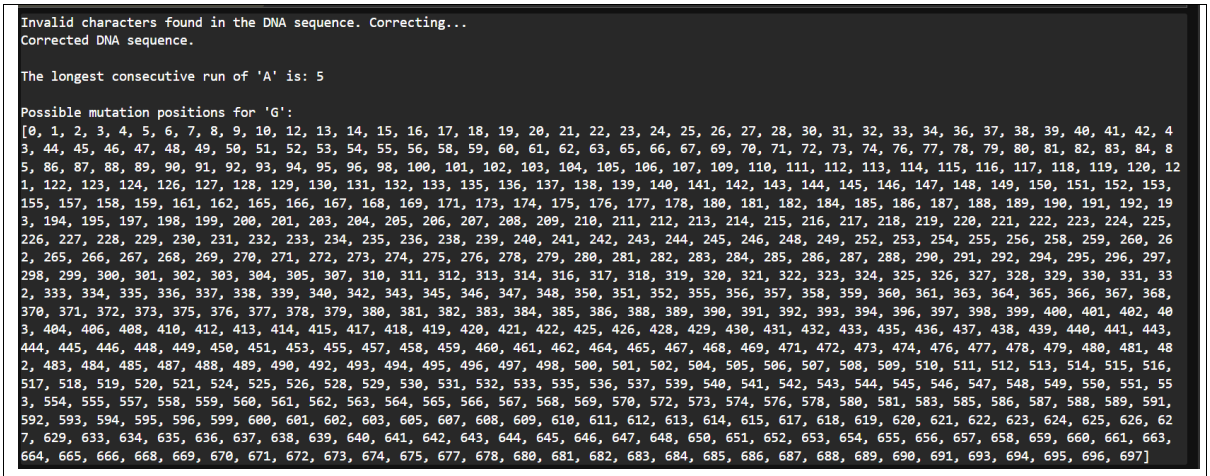

Figure 1.3.4 showing the output of the program

1.3.4 Program test, debug and optimization

Program Testing, Debugging, and Optimisation

Testing

• The validation step successfully removed invalid characters from the input.

• The program accurately identified the longest consecutive run of 'A'.

• Mutation positions for 'G' were correctly located and displayed.

Debugging:

• Issues with invalid characters were fixed by adding a validation check.

• The while loop for counting nucleotide runs was debugged to ensure accurate results.

Optimisation:

• Validation was sped up by using a set for faster checks.

• Comments were added to improve the program's clarity and readability.

1.3.5 Task summary

This Python program validates a DNA sequence, counts the longest run of 'A', and identifies mutation positions for 'G'. It corrects invalid characters but only works for single-character mutations and specific nucleotides.

1.3.5.1 Program advantages

- Validates the DNA sequence and corrects errors.
- Efficiently counts the longest nucleotide run using a while loop.
- Identifies potential mutation positions for a specified target nucleotide.

1.3.5.2 Program limitations

- Only checks for single-character mutations.
- Limited to finding positions for a specific target nucleotide.

.

TASK 1.4 DEBUGGING

1.4.1 Task Overview

The task involves debugging a Python script intended to analyse a DNA sequence. The script calculates the frequency of each nucleotide (A, T, C, G) and identifies the most common triplet (set of three nucleotides) in the sequence. The python script had issues, including incorrect nucleotide counts and errors in identifying the most common triplet. The objectives of this task were to fix these issues, optimize the code, and ensure it works accurately with any valid DNA sequence.

1.4.2 Program design

The program is designed to perform two main functions:

- Takes a DNA string as input, calculates and returns the frequency of each nucleotide (A, T, C, G).

- Identifies the most common triplet in the DNA sequence and supports both overlapping and non-overlapping triplet analysis.

```
# Check if the input is a string
IF dna is not a string THEN
    Raise TypeError with message "Input DNA sequence must be a string."
END IF
# Initialize nucleotide count dictionary
Create dictionary freq with keys 'A', 'T', 'C', 'G' all set to 0
Initialize invalid_chars as 0
# Loop through each character in dna sequence
FOR each char in dna
    IF char is in freq THEN
        Increment freq[char] by 1
    ELSE
        Increment invalid_chars by 1
    END IF
END FOR
# Handle invalid characters
IF invalid_chars equals length of dna THEN
    Raise ValueError with message "The DNA sequence contains no valid nucleotides."
ELSE IF invalid_chars is greater than 0 THEN
    Print "Warning: X invalid characters ignored in the DNA sequence."
END IF
# Display the nucleotide frequencies
Display freq
```

Figure 1.4.1 pseudocode for Nucleotide frequencies

```
IF dna is not a string THEN
    Raise TypeError with message "Input DNA sequence must be a string."
END IF
IF length of dna is less than 3 THEN
    Raise ValueError with message "DNA sequence must have at least 3 nucleotides."
END IF
# Initialize dictionary for triplet counts
Create dictionary triplet_count as empty
# Set step size based on overlapping flag
IF overlapping is True THEN
    Set step to 1
ELSE
    Set step to 3
END IF
FOR i from 0 to length of dna - 2 with step size
    Set triplet to dna[i:i+3]
    IF triplet is in triplet_count THEN
        Increment triplet_count[triplet] by 1
    ELSE
        Set triplet_count[triplet] to 1
    END IF
END FOR
Set max_triplet to the triplet with the highest count in triplet_count
Set count to triplet_count[max_triplet]

# Display the most common triplet and its count
Display max_triplet, count
```

Figure 1.4.2 pseudocode for most common triplet

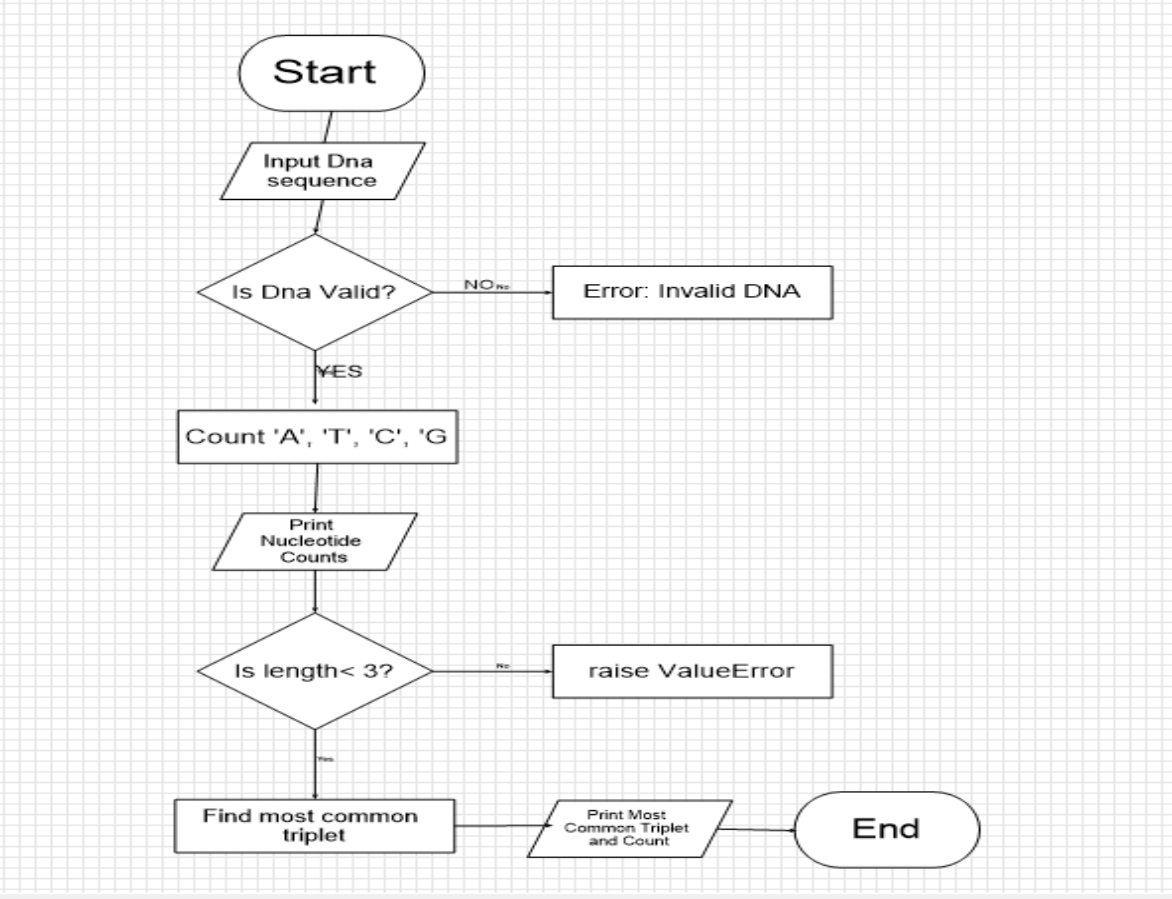

Figure 1.4.3 flowchart of the program

1.4.3 Program execution demonstration

```python
def nucleotide_frequency(dna):
    """
    Calculate how often each nucleotide (A, T, C, G) appears in the DNA sequence.

    Args:
        dna (str): A string representing the DNA sequence.

    Returns:
        dict: A dictionary containing the counts of each nucleotide (A, T, C, G).

    Raises:
        TypeError: If the input is not a string.
        ValueError: If the string contains only invalid characters.
    """
    # Check that the input is a string, otherwise raise an error.
    if not isinstance(dna, str):
        raise TypeError("Input DNA sequence must be a string.")

    # Create a dictionary to store the counts for each nucleotide.
    freq = {'A': 0, 'T': 0, 'C': 0, 'G': 0}
    invalid_chars = 0  # Counter for characters that aren't valid nucleotides.

    # Go through each character in the DNA sequence.
    for char in dna:
        if char in freq:
```

```python
    # Go through each character in the DNA sequence.
    for char in dna:
        if char in freq:
            # If the character is A, T, C, or G, increase its count.
            freq[char] += 1
        else:
            # If it's not a valid nucleotide, increase the invalid character count.
            invalid_chars += 1

    # If there are invalid characters, raise a warning or exception if needed.
    if invalid_chars == len(dna):
        raise ValueError("The DNA sequence contains no valid nucleotides (A, T, C, G).")
    elif invalid_chars > 0:
        print(f"Warning: {invalid_chars} invalid characters ignored in the DNA sequence.")

    # Return the frequency of each nucleotide.
    return freq


def most_common_triplet(dna, overlapping=False):
    """
    Find the three-letter combination (triplet) that appears most often in the DNA sequence.

    Args:
        dna (str): A string representing the DNA sequence.
        overlapping (bool): If True, include overlapping triplets; if False, use non-overlapping triplets.
```

```python
    Returns:
        tuple: A tuple containing the most common triplet and its count.

    Raises:
        TypeError: If the input is not a string.
        ValueError: If the string is shorter than 3 characters.
    """
    # Check that the input is a string and has at least 3 characters.
    if not isinstance(dna, str):
        raise TypeError("Input DNA sequence must be a string.")
    if len(dna) < 3:
        raise ValueError("DNA sequence must have at least 3 nucleotides.")

    # Create a dictionary to count how many times each triplet appears.
    triplet_count = {}

    # Look at groups of three characters in the DNA sequence.
    step = 1 if overlapping else 3
    for i in range(0, len(dna) - 2, step):
        triplet = dna[i:i + 3]  # Get the current triplet (3 characters).
        # Increase the count for this triplet, or set it to 1 if it's the first time.
        triplet_count[triplet] = triplet_count.get(triplet, 0) + 1
```

```
    # Find the triplet that has the highest count.
    max_triplet = max(triplet_count, key=triplet_count.get)
    # Return the most common triplet and its count.
    return max_triplet, triplet_count[max_triplet]


# Example Usage
# Define a DNA sequence to test the functions.
dna_sequence = (
    "AAAATACAATGTACATTCTTCGTATTATAGACATTTAGCTTAAAAATATACCAAAGATTTAATTTTATATTAATACATCAATATTTAATCTGCTGTATATGCGTTTATTTATAGGGGATGGAAATAT"
    "ACTTAGTCATTATAAAGAAAATTTTGCTTGATATAATTATGACATATAAGTCTGTACAAATCACTTGATTTATTTGACACAATGCCAGCTCAAGTAATGACTATTTGGTACATCTTAAGAAAGAGTA"
    "AGCCGGACTTTGAGGACAGAAAAATACATTCATATATCATTCGATTTATTTTTCTTATTTACCTTTTTTAGGACCTTATTTATAATTAAAATCAAAAAAAAAATCTTTCAAAGTTACAAAAAGGACA"
    "ACTTTATACTATAAAGTATACCTTTTTTAATTAAATAAGTTATACAAGAATTATACCATTAACTCACAAACAGCTTATAAATACTTGATTAATTGACTAGATTCATTTT"
)

# Print how many times each nucleotide appears.
print("Nucleotide Frequencies:", nucleotide_frequency(dna_sequence))

# Find and print the most common triplet for non-overlapping triplets.
triplet, count = most_common_triplet(dna_sequence, overlapping=False)
print(f"Most Common Non-Overlapping Triplet: {triplet} (Count: {count})")

# Find and print the most common triplet for overlapping triplets.
triplet, count = most_common_triplet(dna_sequence, overlapping=True)
print(f"Most Common Overlapping Triplet: {triplet} (Count: {count})")
```

```
# Additional Test Case
# Use a smaller DNA sequence for testing.
test_sequence = "AAAATACAATGTACATTCTTCGTATTATAGACATTTAGCTTAAAAATATACCAAAGATTTAATTTTATATTAATACATCA"


print("\nAdditional Test:")
print("Nucleotide Frequencies for Test Sequence:", nucleotide_frequency(test_sequence))

# Find and print the most common non-overlapping triplet for the smaller sequence.
triplet, count = most_common_triplet(test_sequence, overlapping=False)
print(f"Most Common Non-Overlapping Triplet in Test Sequence: {triplet} (Count: {count})")

# Find and print the most common overlapping triplet for the smaller sequence.
triplet, count = most_common_triplet(test_sequence, overlapping=True)
print(f"Most Common Overlapping Triplet in Test Sequence: {triplet} (Count: {count})")
```

Figure 1.4.4 correct input of the debugged code

```
Nucleotide Frequencies: {'A': 192, 'T': 182, 'C': 63, 'G': 53}
Most Common Non-Overlapping Triplet: AAA (Count: 13)
Most Common Overlapping Triplet: AAA (Count: 33)

Additional Test:
Nucleotide Frequencies for Test Sequence: {'A': 35, 'T': 30, 'C': 10, 'G': 5}
Most Common Non-Overlapping Triplet in Test Sequence: ATA (Count: 4)
Most Common Overlapping Triplet in Test Sequence: AAA (Count: 6)
```

Figure 1.4.5 showing the corrected output of the code

The corrected script was tested with the provided DNA sequence and an additional test sequence chosen to include a variety of nucleotide combinations and triplet patterns. This was done to validate the script's robustness in handling different sequence complexities and to ensure consistent performance across various DNA inputs.

1.4. 4 Program test, debug and optimization

Bugs Identified:

- **Incorrect frequency calculation:** The original script did not handle invalid characters, this led to incorrect nucleotide counts.
- **Most common triplet calculation error**: The function did not differentiate between overlapping and non-overlapping triplets, resulting in incorrect triplet counts.

Debugging Process:

- Added input validation to ensure the DNA sequence is a valid string.
- Handled invalid characters by counting and warning the user.
- Modified the triplet function to handle both overlapping and non-overlapping triplets.

Optimization:

- Added error handling and input validation.
- Optimized the triplet function to differentiate between overlapping and non-overlapping triplets.
- Added docstrings to Improved code readability with comments and docstrings.

1.4.5 Task summary

This Python programme calculates nucleotide frequencies and identifies the most common triplet in a DNA sequence. It handles input validation, supports overlapping and non-overlapping triplets, but does not filter invalid characters or handle file input**.**

1.4.5.1 Program advantages

- The corrected script accurately calculates nucleotide frequencies and identifies the most common triplet.
- The program handles invalid input accurately with appropriate error messages.
- Supports both overlapping and non-overlapping triplet calculation

1.4.5.2 Program limitations

- The program assumes the input is a single continuous DNA sequence without spaces or special characters. it may require further modification to handle sequences from files.
- The current implementation prints warnings for invalid characters but does not filter them out.

Task 2.0 Advance of Python

2.1.1 Task Overview

The objective of this task is to develop a Python program that reads DNA sequences from a FASTA file, transcribes DNA to RNA, translates RNA to protein sequences, calculates amino acid frequencies, performs sequence alignments, and constructs a phylogenetic tree to study evolutionary relationships.

The task requires using Biopython to handle sequence data and perform bioinformatics analyses efficiently. The final deliverables include a Python script, input file, and the output showcasing DNA, RNA, protein sequences, alignments, and a phylogenetic tree.

2.1.2 Program design

The program follows a structured approach using Biopython to analyse DNA sequences from a given FASTA file (sequence). The steps include:

- **Input Handling:** Load sequences from the provided FASTA file, Validate and parse sequences using Biopython's SeqIO module.
- **DNA Analysis:** Transcription of DNA to RNA using the **transcribe()** method, Translation of RNA to protein using the **translate()** method. And the Calculation of amino acid frequencies using the **Counter class** from the collections module.
- **Pairwise Alignment:** Perform global alignment between the first two sequences using the **pairwise2 module** from Biopython.
- **Multiple Sequence Alignment (MSA):**Align sequences using Muscle 5 through the command line. Parse the alignment file using **Biopython's AlignIO** module.
- **Phylogenetic Tree Construction:** Calculate a distance matrix from the MSA using DistanceCalculator. Construct a phylogenetic tree using the **Neighbor-Joining (NJ)** method from the DistanceTreeConstructor class.
- Display the phylogenetic tree in ASCII format and visualize it graphically using Matplotlib.

```
Define the path to the FASTA file
Read sequences from the FASTA file
FOR each sequence in the FASTA file
    Get the DNA sequence
    Transcribe DNA to RNA
    Translate RNA to Protein
    Count amino acid frequencies
    Print the sequence ID, DNA, RNA, and Protein sequences
    Print the amino acid frequencies
END FOR
IF there are at least 2 sequences THEN
    Get the first two sequences
    Perform pairwise alignment between them
    Print the aligned sequences
END IF
Load the Multiple Sequence Alignment (MSA) file
Print the alignment
Calculate the distance matrix for the sequences
Print the distance matrix
Construct a phylogenetic tree using the NJ method
Save the tree in Newick format
Print the tree as ASCII
Visualize the phylogenetic tree graphically using matplotlib
```

Figure 2.0 pseudocode for Description: DNA Analysis, Alignment, and Phylogenetic Tree Construction

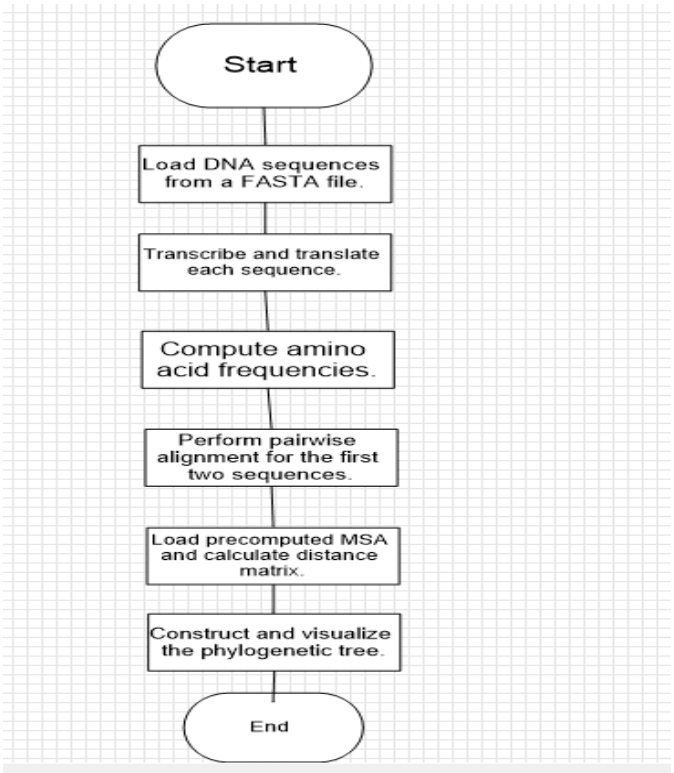

Figure 2.1 flowchart for the program.

2.1.3 Program execution demonstration

```
from Bio import SeqIO
from Bio.Seq import Seq
from Bio.Align import MultipleSeqAlignment
from Bio.SeqRecord import SeqRecord
from Bio.Phylo.TreeConstruction import DistanceTreeConstructor, DistanceCalculator
from Bio import AlignIO
from Bio.Phylo import draw_ascii
from Bio.pairwise2 import align, format_alignment
from collections import Counter
from Bio import Phylo
```

Figure 2.2 Biopython library to handle sequence parsing, alignments, and phylogenetic tree construction.

```
# Path to the FASTA file
path_file = r"C:\Users\44798\Downloads\sequences 2.1 (2).fasta"

# Read the sequences from the FASTA file
sequences = list(SeqIO.parse(path_file, "fasta"))

# Step 1: Analyze each sequence - Transcribe, translate, and calculate amino acid frequency
print("Step 1: DNA Analysis:\n")
for record in sequences:
    # DNA sequence
    dna_seq = record.seq

    # Transcribe DNA to RNA
    rna_seq = dna_seq.transcribe()

    # Translate RNA to Protein
    protein_seq = rna_seq.translate()

    # Calculate amino acid frequency
    amino_acid_freq = Counter(str(protein_seq))

    # Print the results for each sequence
    print(f"Sequence ID: {record.id}")
    print(f"DNA: {dna_seq}")
    print(f"RNA: {rna_seq}")
    print(f"Protein: {protein_seq}")
   # Print amino acid frequencies in a vertical format
   print("Amino Acid Frequency:")
   for amino_acid, frequency in amino_acid_freq.items():
       print(f"  {amino_acid}: {frequency}")
```

Figure 2.3 input of DNA sequencing

```
Step 1: DNA Analysis:

Sequence ID: seq1
DNA: AAAATACAATGTACATTCTTCGTATTATAGACATTTAGCTTAAAAATATACCAAAGATTTAATTTTATATTAATACATCAATATTTAATCTGCTGTATATGCGTTTATTTATAGGGGATGGAAATATACTTAGTCATTATAAAGAAA
ATTTTGCTTGATATAATTATGACATATAAGTCTGTACAAATCACTTGATTTATTTGACACAATGCCAGCTCAAGTAATGACTATTTGGTACATCTTAAGAAAGAGTAAGCCGGACTTTGAGGACAGAAAAATACATTCATATATCATTCGAT
TTATTTTTCTTATTTACCTTTTTTAGGACCTTATTTATAATTAAAATCAAAAAAAAAATCTTTCAAAGTTACAAAAAGGACAACTTTATACTATAAAGTATACCTTTTTTAATTAAATAAGTTATACAAGAATTATACCATTAACTCACAAA
CAGCTTATAAATACTTGATTAATTGACTAGATTCATTTTGCTTTAAGTT
RNA: AAAAUACAAUGUACAUUCUUCGUAUUAUAGACAUUUAGCUUAAAAAUAUACCAAAGAUUUAAUUUUAUAUUAAUACAUCAAUAUUUAAUCUGCUGUAUAUGCGUUUAUUUAUAGGGGAUGGAAAUAUACUUAGUCAUUAUAAAGAAA
AUUUUGCUUGAUAUAAUUAUGACAUAUAAGUCUGUACAAAUCACUUGAUUUAUUUGACACAAUGCCAGCUCAAGUAAUGACUAUUUGGUACAUCUUAAGAAAGAGUAAGCCGGACUUUGAGGACAGAAAAAUACAUUCAUAUAUCAUUCGAU
UUAUUUUUCUUAUUUACCUUUUUUAGGACCUUAUUUAUAAUUAAAAUCAAAAAAAAAAUCUUUCAAAGUUACAAAAAGGACAACUUUAUACUAUAAAGUAUACCUUUUUUAAUUAAAUAAGUUAUACAAGAAUUAUACCAUUAACUCACAAA
CAGCUUAUAAAUACUUGAUUAAUUGACUAGAUUCAUUUUGCUUUAAGUU
Protein: KIQCTFFVL*TFSLKIYQRFNFILIHQYLICCICVYL*GMEIYLVIIKKILLDIIMTYKSVQIT*FI*HNASSSNDYLVHLKKE*AGL*GQKNTFIYHSIYFSYLPFLGPYL*LKSKKKSFKVTKRTTLYYKVYLF*LNKLYK
NYTINSQTAYKYLIN*LDSFCFK
Amino Acid Frequency:
  K: 19
  I: 18
  Q: 6
  C: 5
C: 5
T: 10
F: 13
V: 7
L: 21
*: 9
S: 11
Y: 17
R: 2
N: 8
H: 4
G: 4
M: 2
E: 2
D: 3
A: 3
P: 2
Sequence ID: seq2
DNA: GCCCCCGAGCGTCAGGGCCGCCGTCATTCGCGCTGGGGGTGACGGTCAGACCCGCGCCGTGCCTCTCGACGGCGGTCTGGAGGCCCCCAGTGGTCCCAGCCCGGGGCGCTCGGTCCCGGCACGCCCGGGAGCTACCGCGCGCTGC
CCCGTCGCCTAGGGCTCACGTTTGATCGGACACCTCCAAGGTAGACGAGGTGTCGGCGGCGTCGCAGAGTCTCCACACGCACCACGGCGGTTGTGTAAACACCCTGCCGGACGGCGCCGTCTACACGTTCGAGTCTGACGTCCCTTTATGC
GAAGTGCTCCCCCGAGCCCACACAATGAAGTTTCCAGCGCACTTTTGGGCGAGGCTCGCTGCTGTCAGGCGGGCTCATGCTACTCTTTGAAAGCCGGGCCACAGCTCCGAGCCGGCTCGTCGACGAAGGACTTTGGGCTTGAGCGCCTCGC
CCGCCGTGAGCGCCCGGACTCCGTGGGCCCACGGCGCACCTACTAAGCA
RNA: GCCCCCGAGCGUCAGGGCCGCCGUCAUUCGCGCUGGGGGUGACGGUCAGACCCGCGCCGUGCCUCUCGACGGCGGUCUGGAGGCCCCCAGUGGUCCCAGCCCGGGGCGCUCGGUCCCGGCACGCCCGGGAGCUACCGCGCGCUGCC
CCCGUCGCCUAGGGCUCACGUUUGAUCGGACACCUCCAAGGUAGACGAGGUGUCGGCGGCGUCGCAGAGUCUCCACACGCACCACGGCGGUUGUGUAAACACCCUGCCGGACGGCGCCGUCUACACGUUCGAGUCUGACGUCCCUUUAUGC
GAAGUGCUCCCCCGAGCCCACACAAUGAAGUUUCCAGCGCACUUUUGGGCGAGGCUCGCUGCUGUCAGGCGGGCUCAUGCUACUCUUUGAAAGCCGGGCCACAGCUCCGAGCCGGCUCGUCGACGAAGGACUUUGGGCUUGAGCGCCUCGC
CCGCCGUGAGCGCCCGGACUCCGUGGGCCCACGGCGCACCUACUAAGCA
Protein: APERQGRRHSRWG*RSDPRRASRRRSGGPQWSQPGALGPGTPGSYRALPPVA*GSRLIGHLQGRRGVGGVAESPHAPRRLCKHPAGRRRLHVRV*RPFMVKCSPEPTQ*SFQRTFGRGSLLSGGLMLLFESRATAPSRLVDE
LWA*APRPAVSARTPWAHGAPTK
Amino Acid Frequency:
  A: 16
  P: 19
  E: 5
  R: 27
  Q: 6
  G: 21
  H: 6
  S: 15
  W: 4
  *: 5
  D: 2
  L: 13
  T: 6
  Y: 1
  V: 8
  I: 1
  C: 2
  C: 2
  K: 3
  F: 4
  M: 2
Sequence ID: seq3
DNA: ACTAACATCCAGTTAATAATTTATACATAGTTTAAGAATATACTACATTAAAAATACGAACTAAGACTGTATTACGCTCCCATCTAATCTATATTTTGTGTTCTATAAGATCCTTTATTAATATAGTATACACTGTAATTATTACAA
CCAATCTGTGTAAGTAGGTTAATGCGATATTTGTTTGTTCTAAATCAATGATAACGTATAATTTCTATTATTCTAAATTCTCTCCTATCTGTCTATTTCAGCAGGATAAAGATAGAGGTATTCTTCCAATGTTTAATAATATGATAGTATCA
ATTATAAGTGGAATTTGTAATTAATTACCTATCAATTTCTACGATGATATGTCAGGACCCCTCGTAAACCGCTTCGTAATTTTAAAACGTACCATTGTTAATATCCGTGAGCGTTATGAAACTAGGTCGCTACCTGAATTATTGTGAGATGT
TTTGCGTGAAAATCGTAGTTCGAATTGTCCGATACTTAATGAACCTTAGA
RNA: ACUAACAUCCAGUUAAUAAUUUAUACAUAGUUUAAGAAUAUACUACAUUAAAAAUACGAACUAAGACUGUAUUACGCUCCCAUCUAAUCUAUAUUUUGUGUUCUAUAAGAUCCUUUAUUAAUAUAGUAUACACUGUAAUUAUUACAA
CCAAUCUGUGUAAGUAGGUUAAUGCGAUAUUUGUUUGUUCUAAAUCAAUGAUAACGUAUAAUUUCUAUUAUUCUAAAUUCUCUCCUAUCUGUCUAUUUCAGCAGGAUAAAGAUAGAGGUAUUCUUCCAAUGUUUAAUAAUAUGAUAGUAUCA
AUUAUAAGUGGAAUUUGUAAUUAAUUACCUAUCAAUUUCUACGAUGAUAUGUCAGGACCCCUCGUAAACCGCUUCGUAAUUUUAAAACGUACCAUUGUUAAUAUCCGUGAGCGUUAUGAAACUAGGUCGCUACCUGAAUUAUUGUGAGAUGU
UUUGCGUGAAAAUCGUAGUUCGAAUUGUCCGAUACUUAAUGAACCUUAGA
Protein: TNIQLIIYT*FKNILH*KYELRLYYAPI*SIFCVL*DPLLI*YTL*LLQPICVSRLMRYLFVLNQ**RIISIILNSLLSVYFSRIKIEVFFQCLII**YQL*VEFVINYLSISTMICQDPS*TAS*F*NVPLLISVSVMKLGR
YLNYCEMFCVKIVVRIVRYLMNLR
Amino Acid Frequency:
  T: 5
  N: 8
```

```
I: 22
Q: 6
L: 25
Y: 12
*: 14
F: 9
K: 5
H: 1
E: 4
R: 9
A: 2
P: 5
S: 12
C: 6
V: 14
D: 2
M: 5
G: 1
Sequence ID: seq4
DNA: TGGCGCCCCCGACAGCCATGCGTACGGCAGGCCACAAGCCGTGGCGCGACCGACCACAGCGTGTCGGCGCTTTACGCGCGGCCGGACGGGACCTGGCAGGGCGAGGTTGTGTGGGCTGAGCCGCCGGCTGCCCGGGAGCGGGCGGCA
GCGGGGGGAACGGCGGGGATGAGCGGCCGCGCCGCCCGGCCCCCCCGGTACGCAGCAAGCCCGCCCTCGTCACGCGGGGTCCGGGGCAGGGTGGGGCAGGCGCTCCCGTCGCCCGGCACGCTTGTGTCGGCGCTCGTTGGCGCCGGACCAGG
CGGGTCAGCGGAAGGGCTGACCCGTGTGCCCGCCGGTTCCCGCAAGTGGCGGGCATCGCCCTGTCCGCCTGGGTCAGGGCACACTATGCGGGGGAGTAGCTCGGAGTCTACCATTCGTTCGAGCGTCGCGCAGGAACATCGTCGAGCGCCCC
CTCCCCTGAGCGCTCGCGGCCTTGGGCCCGAGCGGGGCAGGCGCGCCACGC
RNA: UGGCGCCCCCGACAGCCAUGCGUACGGCAGGCCACAAGCCGUGGCGCGACCGACCACAGCGUGUCGGCGCUUUACGCGCGGCCGGACGGGACCUGGCAGGGCGAGGUUGUGUGGGCUGAGCCGCCGGCUGCCCGGGAGCGGGCGGCA
GCGGGGGGAACGGCGGGGAUGAGCGGCCGCGCCGCCCGGCCCCCCGGUACGCAGCAAGCCCGCCCUCGUCACGCGGGGUCCGGGGCAGGGUGGGGCAGGCGCUCCCGUCGCCCGGCACGCUUGUGUCGGCGCUCGUUGGCGCCGGACCAGG
CGGGUCAGCGGAAGGGCUGACCCGUGUGCCCGCCGGUUCCCGCAAGUGGCGGGCAUCGCCCUGUCCGCCUGGGUCAGGGCACACUAUGCGGGGGAGUAGCUCGGAGUCUACCAUUCGUUCGAGCGUCGCGCAGGAACAUCGUCGAGCGCCCC
CUCCCCUGAGCGCUCGCGGCCUUGGGCCCGAGCGGGGCAGGCGCGCCACGC
Protein: WRPRQPCVRQATSRGATDHSVSALYARPDGTWQGEVVWAEPPAARERAAAGGTAGMSGRAARPPRYAASPPSSRGVRGRVGQALPSPGTLVSALVGAGPGGSAEGLTRVPAGSRKWRASPCPPGSGHTMRGSSSESTIRSSVA
QEHRRAPPPLSARGLGPERGRRAT
Amino Acid Frequency:
  W: 4
  R: 24
  P: 20
  Q: 5
  C: 2
  V: 10
  A: 25
  T: 9
  S: 20
  G: 23
  D: 2
  H: 3
  L: 7
  Y: 2
  E: 7
  M: 2
  K: 1
  I: 1

tep 2: Pairwise Alignment:
```

Figure 2.4 output for DNA sequencing

```python
# Step 2: Pairwise Alignment
print("\nStep 2: Pairwise Alignment:\n")
if len(sequences) >= 2:
    seq1 = str(sequences[0].seq)
    seq2 = str(sequences[1].seq)
    alignments = align.globalxx(seq1, seq2)
    print(format_alignment(*alignments[0]))
```

Figure 2.5 input for pairwise sequencing

```
Step 2: Pairwise Alignment:

AAAATA-CA-----ATG--TACATT---CTTCG---T-ATTATA-GACATTTAGCT-----TAAAA-A---T-AT-ACCAAA-GATTTAATTTTATATTAATACAT-CAATA--T---T-TA--ATCTG-CTG-TA-T--ATGCGTTT----
-ATT-TA--T---AG---GGGATG-GAAATATACTTA-G-TCATTATAAA--GAAA--ATTTT-GCTT--GATATAATTATG-A-CATA--------TA---A---GTCTG--TA---CAAATCACTTGATTTATTTGA-C--ACAATGCCA
GCTC-AAG-TAATGACT-AT--T-T-GGTACAT--CT-TAA-GAA-AGAGT----A-A-GC-C---GGACTTTGAG--GACA-GAAA-AATACAT---T-C---A--------TA-TATCATTCGATTT--ATTTT-TCTT-ATT--TACCT
TT-TTTA-GGACCTTATTTAT-AAT-TAAAA--TCAAAAA-----A----A-A-AATCTTTCA-AAGTTA-CAAAA-AG-GACAACTTTAT-----A--CTA---TAAAG-TA-T-A--C---CTTTTTTA-AT--TAAA-TAAG-TTATAC
AAGAATTATA-CCATT-----A-A-CTCACAA-A-CA-G-CTTA--T--A--AAT--ACTT-GA---TTA-AT-----T-GA-C------T-AG-------A-T---T---C--ATTTT-G-CT-----T--TAAGTT--
```

Figure 2.6 output for pairwise sequencing

```python
# Step 3: Multiple Sequence Alignment
input_file = r"C:\Users\44798\Downloads\sequence_analysis\sequence.clw"
alignment = AlignIO.read(input_file, "fasta")  # Use the appropriate format if needed

# Display the alignment
print("\nStep 3: Multiple Sequence Alignment:\n")
print(alignment)

# Step 4: Phylogenetic Tree Construction
print("\nStep 4: Phylogenetic Tree Construction:\n")

# Calculate the distance matrix
calculator = DistanceCalculator("identity")
dm = calculator.get_distance(alignment)
print("Distance Matrix:\n", dm)

# Construct the phylogenetic tree
constructor = DistanceTreeConstructor(calculator, method="nj")
tree = constructor.build_tree(alignment)

# Construct the phylogenetic tree
constructor = DistanceTreeConstructor(calculator, method="nj")
tree = constructor.build_tree(alignment)

# Save the tree in Newick format
Phylo.write(tree, "phylogenetic_tree.nwk", "newick")

# Visualize the tree in ASCII
print("\nPhylogenetic Tree (ASCII Representation):\n")
draw_ascii(tree)

# Visualize the tree graphically
print("\nDisplaying the Phylogenetic Tree Graphically...")
Phylo.draw(tree)
plt.show()
```

Figure 2.7 input of MSA and phylogenetic tree construction

```
Step 3: Multiple Sequence Alignment:

Alignment with 4 rows and 635 columns
AAAATACAATGTACATTCTTCGTATTATAGACAT---TTAGCTT...GTT seq1
ACTAACATCC--------AGTTAATAATTTATAC---ATAGTTT...AGA seq3
GCCCCCGAGCGTCAGGGCCGCCGTCATTCGCGCT---GGGGGTG...GCA seq2
TGGCGCCCCCGACAGCCATGCGTACGGCAGGCCACAAGCCGTGG...CGC seq4

Step 4: Phylogenetic Tree Construction:

Distance Matrix:
 seq1    0.000000
seq3    0.535433    0.000000
seq2    0.729134    0.629921    0.000000
seq4    0.833071    0.754331    0.486614    0.000000
    seq1    seq3    seq2    seq4
```

```
Phylogenetic Tree (ASCII Representation):

                              _____________________________________ seq1
  ___________________________|
 |                           |_____________________________ seq3
_|
 |_________________________ seq2
 |
 |_____________________________________ seq4


Displaying the Phylogenetic Tree Graphically...
```

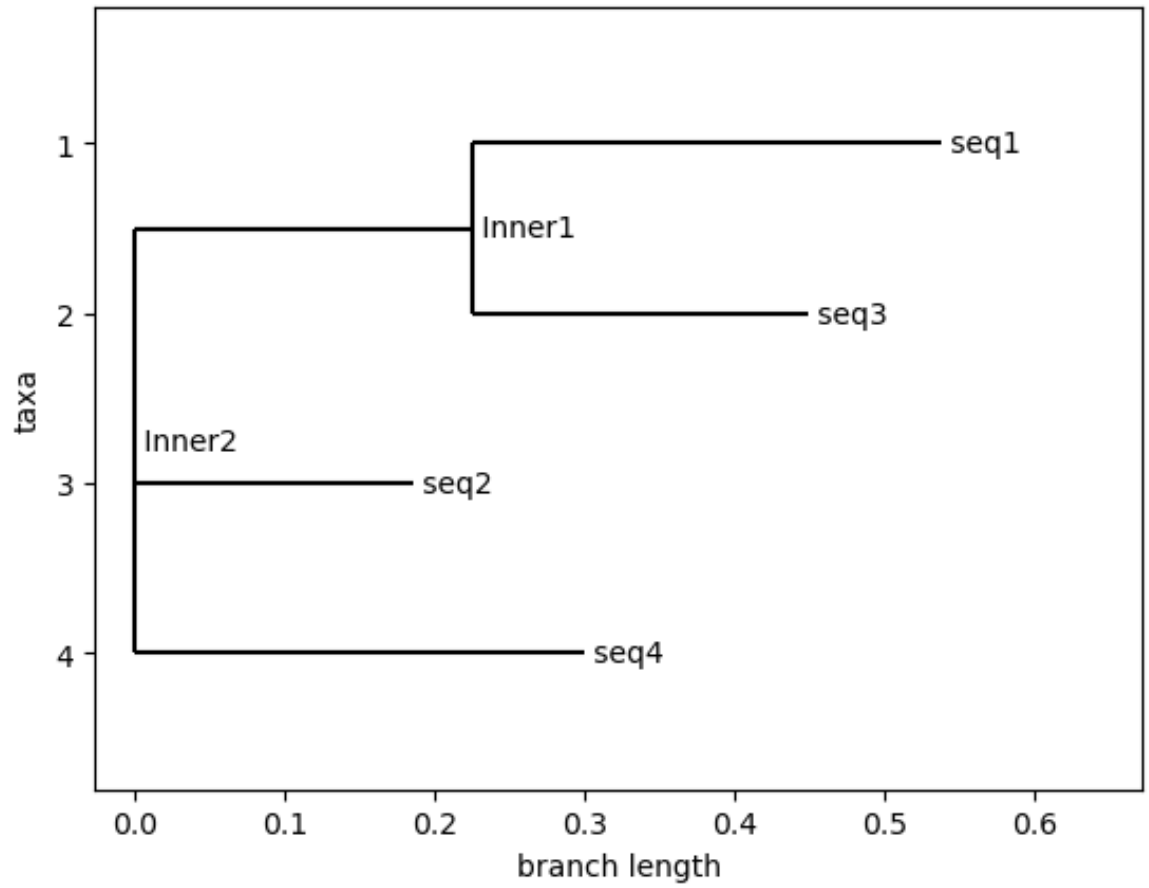

Figure 2.8 output of MSA and phylogenetic tree construction

2.1.3 Program test, debug and optimization

The code was tested using sample sequences from the provided FASTA file. Errors encountered during execution were primarily due to file path issues and format mismatches in MSA. These were resolved by:

- Correcting file paths.
- Ensuring compatibility between Muscle-generated alignment files and Biopython's AlignIO module.
- Optimizing the tree visualization step to handle graphical output.

Debugging Tools Used:

- Python's built-in print() function for debugging.
- Muscle 5 for MSA through the command line.

Optimization Techniques:

- Modularizing the code for reusability.
- Using efficient Biopython methods to handle large sequences.

2.1.5 Task summary

This Python program analyses DNA sequences from a FASTA file using Biopython. It transcribes DNA, translates RNA, calculates amino acid frequencies, and performs sequence alignments. It also constructs and visualises a phylogenetic tree. The program handles sequence data efficiently but requires specific input formats and may not suit all datasets.

2.1.5.1 Program advantages

- Efficient handling of sequence data using Biopython.
- Modular design for easy maintenance and future enhancements.
- Comprehensive output covering all aspects of the task.

2.1.5.2 Program limitations

- The program requires the input file to be in a specific format (FASTA).
- The pairwise alignment is limited to the first two sequences.
- The graphical visualization requires Matplotlib, which may not be available in all environments.

Task 2.2: Exploring Cancer-Linked Muscle Wasting Through 1H-NMR Metabolomics and Machine Learning

2.2.1 Task Overview

The objective of this task is to analyse a cancer-associated skeletal muscle wasting dataset using Python. The dataset contains 1H-NMR urine metabolite data, and the goal is to explore, visualize, and predict skeletal muscle loss associated with cancer. This involves data preparation, visualization, dimensionality reduction using PCA, and building a simple machine learning mode

2.2.2 Program design

The Python program is structured to follow a step-by-step approach:

- Data Preparation: Load the dataset, clean it, check its structure, and generate descriptive statistics.
- Data Visualization: Visualize the distribution of metabolites and the target variable using histograms, boxplots, and heatmaps.

- Dimensionality Reduction: Apply PCA to reduce the dimensionality of the dataset and plot the first two principal components, making data visualization easier
- Machine Learning Model: Build a Random Forest Classifier to predict skeletal muscle wasting and evaluate its performance.

The dataset was loaded into a Pandas DataFrame to handle tabular data efficiently, separating metabolites and labels for analysis. A Random Forest Classifier was chosen for predictive modelling due to its ability to manage complex data and highlight important features. Missing values were addressed using forward fill to maintain temporal order and minimise data loss. Heatmaps and scatter plots, helped explore correlations and distributions.

```
START
    Define the file path to the Excel file
    LOAD the Excel file into a DataFrame
    DISPLAY basic information about the dataset:
         Dataset info
         Missing values count
         Basic descriptive statistics
    HANDLE missing values by filling them using forward fill method
    DISPLAY the distribution of the target variable
    # Data Visualization
    Remove columns unrelated to metabolites
    PLOT histogram for metabolites
    PLOT boxplot for metabolites
    PLOT distribution of the target variable
    PLOT a correlation heatmap for metabolites
    # Dimensionality Reduction (PCA)
    SCALE the data
    Apply PCA to reduce dimensions to 2
    PLOT the PCA scatter plot with color coding for the target variable
    # Build a Machine Learning Model
     Separate features (X) and target (y)
     SPLIT data into training and test sets
    TRAIN a Random Forest model
     PREDICT using the trained model
    # Build a Machine Learning Model
     Separate features (X) and target (y)
     SPLIT data into training and test sets
    TRAIN a Random Forest model
     PREDICT using the trained model
    DISPLAY classification report for model performance
    # Evaluate Model with ROC Curve and AUC
     Recode the target variable to binary if needed
     PLOT ROC curve
     CALCULATE AUC score and display it on the ROC curve
    # Save Results
     Add model predictions to the dataset
     SAVE the results to a new Excel file
    PRINT "Analysis complete. Results saved."

END
```

Figure 2.2.1 pseudocode for the programme.

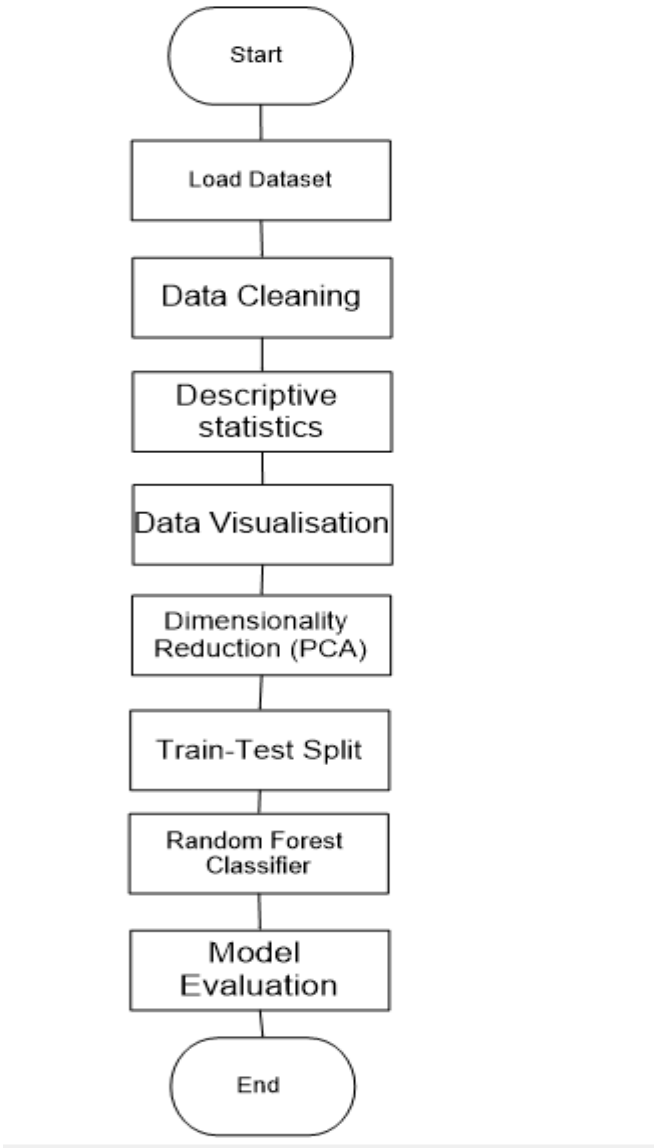

Figure 2.2.2 flowchart of the program

2.2.3 Program execution demonstration

Data Preparation.

- The dataset is loaded from an Excel file.
- Missing values are handled using forward fill.
- Descriptive statistics and basic data structure are displayed**.**

```
import pandas as pd
import seaborn as sns
import matplotlib.pyplot as plt
from sklearn.model_selection import train_test_split
from sklearn.ensemble import RandomForestClassifier
from sklearn.metrics import classification_report, roc_curve, auc
from sklearn.decomposition import PCA
from sklearn.preprocessing import StandardScaler
import numpy as np

# Step 1: Data Preparation
file_path = r"C:\Users\44798\Downloads\metabolite 2.2 (2).xlsx"   # file path

# Load the Excel file
data = pd.read_excel(file_path)

# Display basic information about the dataset
print("Dataset Info:")
print(data.info())
print("\nMissing Values:\n", data.isnull().sum())
print("\nBasic Descriptive Statistics:")
print(data.describe())

# Handle missing values
data.fillna(method='ffill', inplace=True)  # Forward fill as missing data handling strategy
```

Figure 2.2.3 Data preparation steps

Data Visualization

- **Distribution of Metabolites**: Histograms and boxplots are used to visualize the metabolite concentrations.
- **Target Variable Distribution**: A count plot shows the distribution of the target variable.
- **Correlation Heatmap**: A heatmap is generated to visualize correlations between metabolites.

```
# Confirm target variable distribution
target_variable = 'Label.1'  # Set the target variable as Label.1
print("\nTarget variable distribution:")
print(data[target_variable].value_counts())

# Step 2: Data Visualization

# Distribution of metabolites
metabolites = data.drop(columns=['Label', 'Label.1', 'ID'])  # Assuming other columns are metabolites

# Histogram for metabolites
metabolites.hist(bins=20, figsize=(15, 10))
plt.tight_layout()
plt.show()
```

```
# Boxplot for metabolites
plt.figure(figsize=(15, 5))
sns.boxplot(data=metabolites)
plt.xticks(rotation=90)
plt.title("Boxplot of Metabolite Concentrations")
plt.show()

# Distribution of the target variable
sns.countplot(x=data['Label.1'])
plt.title("Target Variable Distribution")
plt.xlabel("Label.1 (0 = Non-cachexic, 1 = Cachexic)")
plt.ylabel("Count")
plt.show()

# Correlation heatmap
plt.figure(figsize=(12, 8))
sns.heatmap(metabolites.corr(), cmap="coolwarm", annot=False)
plt.title("Correlation Heatmap of Metabolites")
plt.show()
```

Figure 2.2.4 Data visualization steps

Dimensionality Reduction with PCA

- PCA is applied to the scaled dataset to reduce its dimensions and plot the first two principal components**.**

```
# Step 3: Dimensionality Reduction with PCA
scaler = StandardScaler()
X_scaled = scaler.fit_transform(metabolites)

pca = PCA(n_components=2)
X_pca = pca.fit_transform(X_scaled)

plt.scatter(X_pca[:, 0], X_pca[:, 1], c=data['Label.1'], cmap='viridis', edgecolor='k', alpha=0.7)
plt.title("PCA Scatter Plot")
plt.xlabel("Principal Component 1")
plt.ylabel("Principal Component 2")
plt.colorbar(label="Target (Label.1)")
plt.show()
```

Figure 2.2.5 Dimensionality reduction steps with PCA

Machine Learning Model

- A Random Forest Classifier is used to predict skeletal muscle loss.
- The dataset is split into training and test sets.

```python
# Step 4: Build a Machine Learning Model

# Separate features and target
X = X_scaled  # Features after scaling
y = data['Label.1']  # Target variable

# Split data into training and test sets
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.3, random_state=42)

# Train a Random Forest Classifier
model = RandomForestClassifier(random_state=42)
model.fit(X_train, y_train)

# Predict on test set
y_pred = model.predict(X_test)
y_pred_proba = model.predict_proba(X_test)[:, 1]

# Classification report
print("\nClassification Report:")
print(classification_report(y_test, y_pred))

# Step 5: Evaluate the Model using ROC Curve and AUC

# Recode the target variable to binary (if needed)
y_test_binary = y_test.replace({1: 0, 2: 1})  # Assuming original labels are {1, 2}
```

```python
# ROC curve and AUC
fpr, tpr, _ = roc_curve(y_test_binary, y_pred_proba)
roc_auc = auc(fpr, tpr)

plt.figure(figsize=(8, 6))
plt.plot(fpr, tpr, label=f"AUC = {roc_auc:.2f}")
plt.plot([0, 1], [0, 1], 'r--')
plt.xlabel("False Positive Rate")
plt.ylabel("True Positive Rate")
plt.title("ROC Curve")
plt.legend(loc="lower right")
plt.show()

# Step 6: Save the model predictions and analysis results
data['Predictions'] = model.predict(X_scaled)  # Add predictions to the original data
data.to_excel("metabolite_analysis_results.xlsx", index=False)

print("Analysis complete. Results saved to 'metabolite_analysis_results.xlsx'.")
```

Figure 2.2.6 machine learning model steps

```
Dataset Info:
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 75 entries, 0 to 74
Data columns (total 66 columns):
 #   Column                      Non-Null Count  Dtype
---  ------                      --------------  -----
 0   ID                          75 non-null     object
 1   Label                       75 non-null     object
 2   Label.1                     75 non-null     int64
 3   1,6-Anhydro-beta-D-glucose  75 non-null     float64
 4   1-Methylnicotinamide        75 non-null     float64
 5   2-Aminobutyrate             75 non-null     float64
 6   2-Hydroxyisobutyrate        75 non-null     float64
 7   2-Oxoglutarate              75 non-null     float64
 8   3-Aminoisobutyrate          75 non-null     float64
 9   3-Hydroxybutyrate           75 non-null     float64
 10  3-Hydroxyisovalerate        75 non-null     float64
 11  3-Indoxylsulfate            75 non-null     float64
 12  4-Hydroxyphenylacetate      75 non-null     float64
 13  Acetate                     75 non-null     float64
 14  Acetone                     75 non-null     float64
 15  Adipate                     75 non-null     float64
 16  Alanine                     75 non-null     float64
 17  Asparagine                  75 non-null     float64
 18  Betaine                     75 non-null     float64
 19  Carnitine                   75 non-null     float64
 20  Citrate                     75 non-null     float64
 21  Creatine                    75 non-null     float64
 23  Dimethylamine               75 non-null     float64
 24  Ethanolamine                75 non-null     float64
 25  Formate                     75 non-null     float64
 26  Fucose                      75 non-null     float64
 27  Fumarate                    75 non-null     float64
 28  Glucose                     75 non-null     float64
 29  Glutamine                   75 non-null     float64
 30  Glycine                     75 non-null     float64
 31  Glycolate                   75 non-null     float64
 32  Guanidoacetate              75 non-null     float64
 33  Hippurate                   75 non-null     float64
 34  Histidine                   75 non-null     float64
 35  Hypoxanthine                75 non-null     float64
 36  Isoleucine                  75 non-null     float64
 37  Lactate                     75 non-null     float64
 38  Leucine                     75 non-null     float64
 39  Lysine                      75 non-null     float64
 40  Methylamine                 75 non-null     float64
 41  Methylguanidine             75 non-null     float64
 42  N,N-Dimethylglycine         75 non-null     float64
 43  O-Acetylcarnitine           75 non-null     float64
 44  Pantothenate                75 non-null     float64
 45  Pyroglutamate               75 non-null     float64
 46  Pyruvate                    75 non-null     float64
 47  Quinolinate                 75 non-null     float64
 48  Serine                      75 non-null     float64
 49  Succinate                   75 non-null     float64
 50  Sucrose                     75 non-null     float64
 51  Tartrate                    75 non-null     float64
 52  Taurine                     75 non-null     float64
 53  Threonine                   75 non-null     float64
 54  Trigonelline                75 non-null     float64
 55  Trimethylamine N-oxide      75 non-null     float64
 56  Tryptophan                  75 non-null     float64
 57  Tyrosine                    75 non-null     float64
 58  Uracil                      75 non-null     float64
 59  Valine                      75 non-null     float64
 60  Xylose                      75 non-null     float64
 61  cis-Aconitate               75 non-null     float64
 62  myo-Inositol                75 non-null     float64
 63  trans-Aconitate             75 non-null     float64
 64  pi-Methylhistidine          75 non-null     float64
 65  tau-Methylhistidine         75 non-null     float64
dtypes: float64(63), int64(1), object(2)
memory usage: 38.8+ KB
None
```

```
Missing Values:
 ID                          0
Label                        0
Label.1                      0
1,6-Anhydro-beta-D-glucose   0
1-Methylnicotinamide         0
                            ..
cis-Aconitate                0
myo-Inositol                 0
trans-Aconitate              0
pi-Methylhistidine           0
tau-Methylhistidine          0
Length: 66, dtype: int64

Basic Descriptive Statistics:
         Label.1  1,6-Anhydro-beta-D-glucose  1-Methylnicotinamide  \
count  75.000000                   75.000000             75.000000
mean    1.400000                  107.073467             68.072533
std     0.493197                  131.450597            131.265193
min     1.000000                    4.710000              6.420000
25%     1.000000                   28.365000             14.905000
50%     1.000000                   45.600000             36.230000
75%     2.000000                  144.055000             71.905000
max     2.000000                  685.400000           1032.770000

       2-Aminobutyrate  2-Hydroxyisobutyrate  2-Oxoglutarate  \
count        75.000000             75.000000       75.000000
mean         18.070400             37.340933      147.104400
```

```
       2-Aminobutyrate  2-Hydroxyisobutyrate  2-Oxoglutarate  \
count        75.000000             75.000000       75.000000
mean         18.070400             37.340933      147.104400
std          27.975909             24.237811      346.891029
min           1.280000              4.850000        5.530000
25%           5.235000             15.490000       22.310000
50%          10.380000             32.460000       47.940000
75%          18.925000             56.875000       95.135000
max         172.430000             93.690000     2465.130000

       3-Aminoisobutyrate  3-Hydroxybutyrate  3-Hydroxyisovalerate  \
count           75.000000          75.000000             75.000000
mean            57.509200          20.954133             21.026267
std            102.248447          26.097521             24.313831
min              2.610000           1.700000              0.920000
25%             11.475000           5.900000              5.260000
50%             22.420000          11.590000             12.550000
75%             53.785000          27.960000             29.385000
max            561.160000         175.910000            164.020000

       3-Indoxylsulfate  ...  Tryptophan    Tyrosine      Uracil      Valine  \
count         75.000000  ...   75.000000   75.000000   75.000000   75.000000
mean         212.242533  ...   63.437333   77.839200   34.398667   33.998933
std          194.476185  ...   52.354696   80.105531   34.324536   28.168729
min           27.660000  ...    8.670000    4.220000    3.100000    4.100000
25%           82.270000  ...   21.120000   23.570000   11.940000   11.385000
```

```
count         75.000000  ...   75.000000   75.000000   75.000000   75.000000
mean         212.242533  ...   63.437333   77.839200   34.398667   33.998933
std          194.476185  ...   52.354696   80.105531   34.324536   28.168729
min           27.660000  ...    8.670000    4.220000    3.100000    4.100000
25%           82.270000  ...   21.120000   23.570000   11.940000   11.385000
50%          138.380000  ...   46.060000   58.560000   25.790000   31.820000
75%          298.005000  ...   96.540000  106.780000   40.875000   47.465000
max         1043.150000  ...  239.850000  539.150000  179.470000  160.770000

            Xylose  cis-Aconitate  myo-Inositol  trans-Aconitate  \
count    75.000000      75.000000     75.000000        75.000000
mean    100.095333     202.051200    132.184267        38.127733
std     253.328308     281.438952    170.236517        34.341131
min      10.070000      12.940000     11.590000         4.900000
25%      29.815000      35.350000     29.970000        12.430000
50%      49.900000     103.540000     72.970000        24.780000
75%      88.240000     254.680000    164.055000        56.830000
max    2164.620000    1863.110000    854.060000       181.270000

       pi-Methylhistidine  tau-Methylhistidine
count           75.00000            75.000000
mean           373.95520            88.197333
std            537.18276            77.680573
min             11.36000             8.000000
25%             67.36000            26.850000
50%            162.39000            64.720000
75%            393.51000           126.475000
max           2697.28000           317.350000
```

```
[8 rows x 64 columns]

Target variable distribution:
Label.1
1    45
2    30
Name: count, dtype: int64
```

Figure 2.2.7 Output of the program

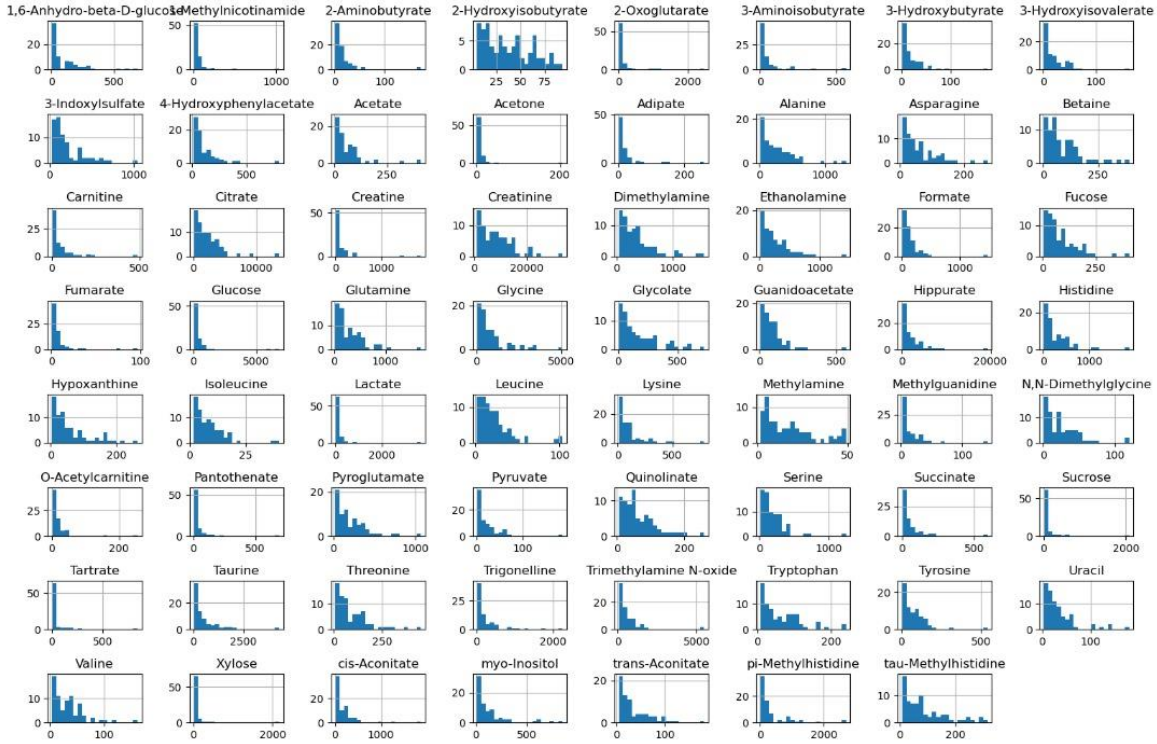

Figure 2.2.8 multiple histograms representing the distribution of various metabolites.

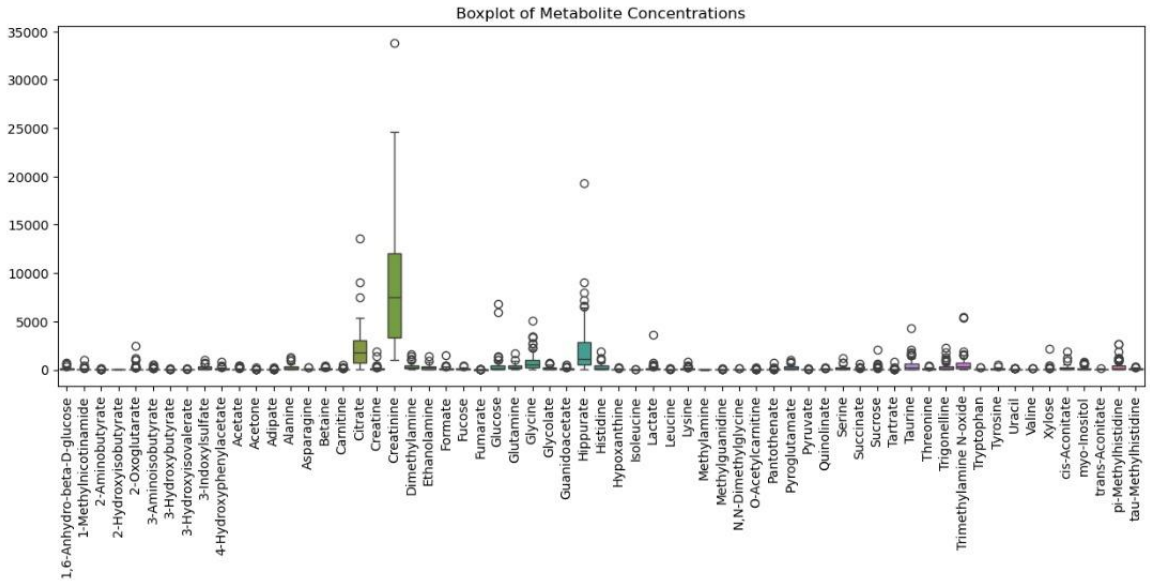

Figure 2.2.9 box plot of metabolite concentration

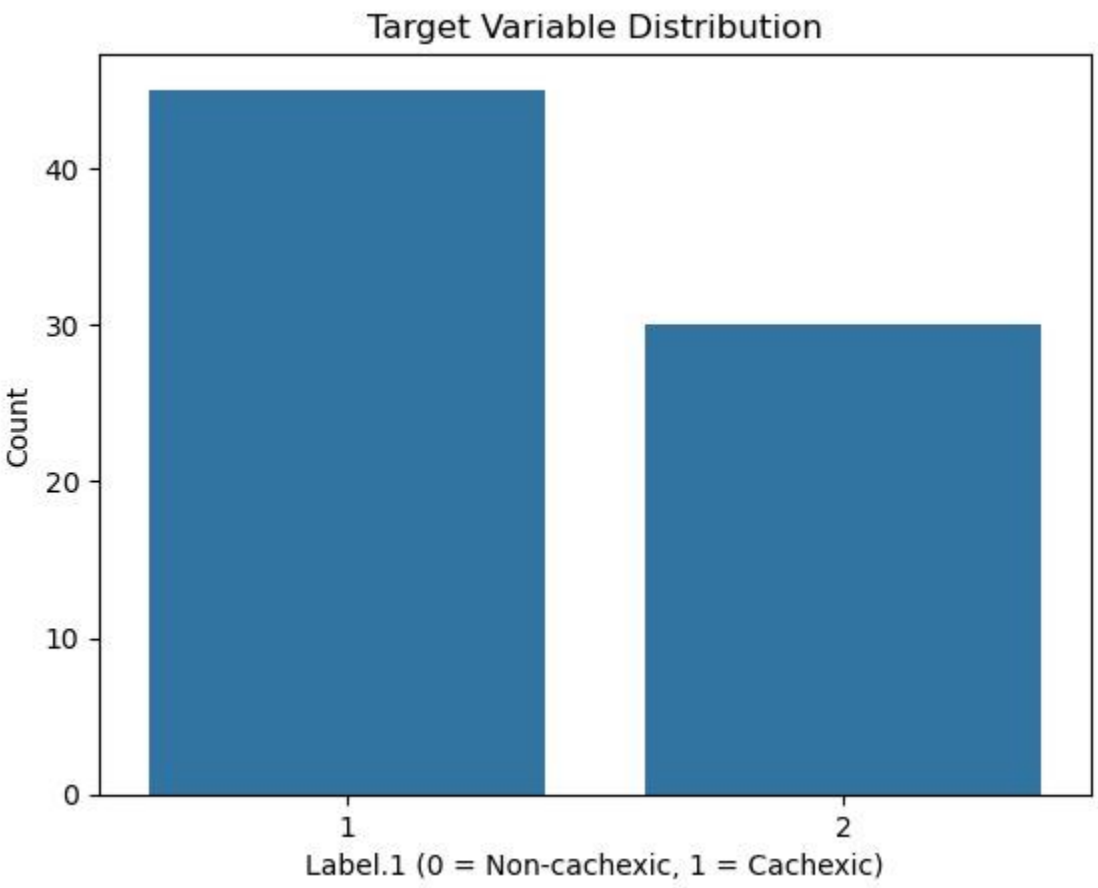

Figure 2.2.10 bar plot showing the distribution of target variable in the data set.

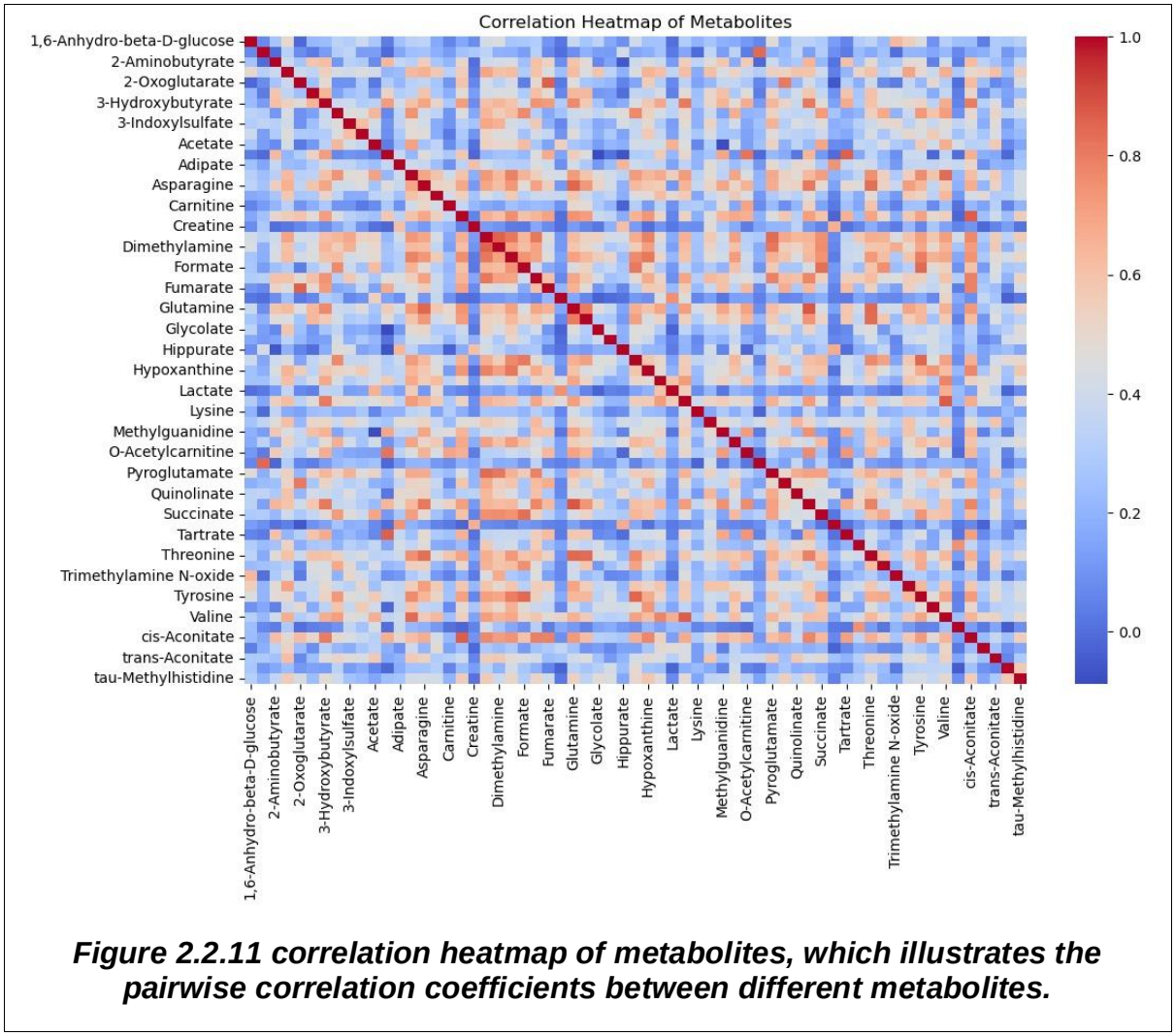

Figure 2.2.11 correlation heatmap of metabolites, which illustrates the pairwise correlation coefficients between different metabolites.

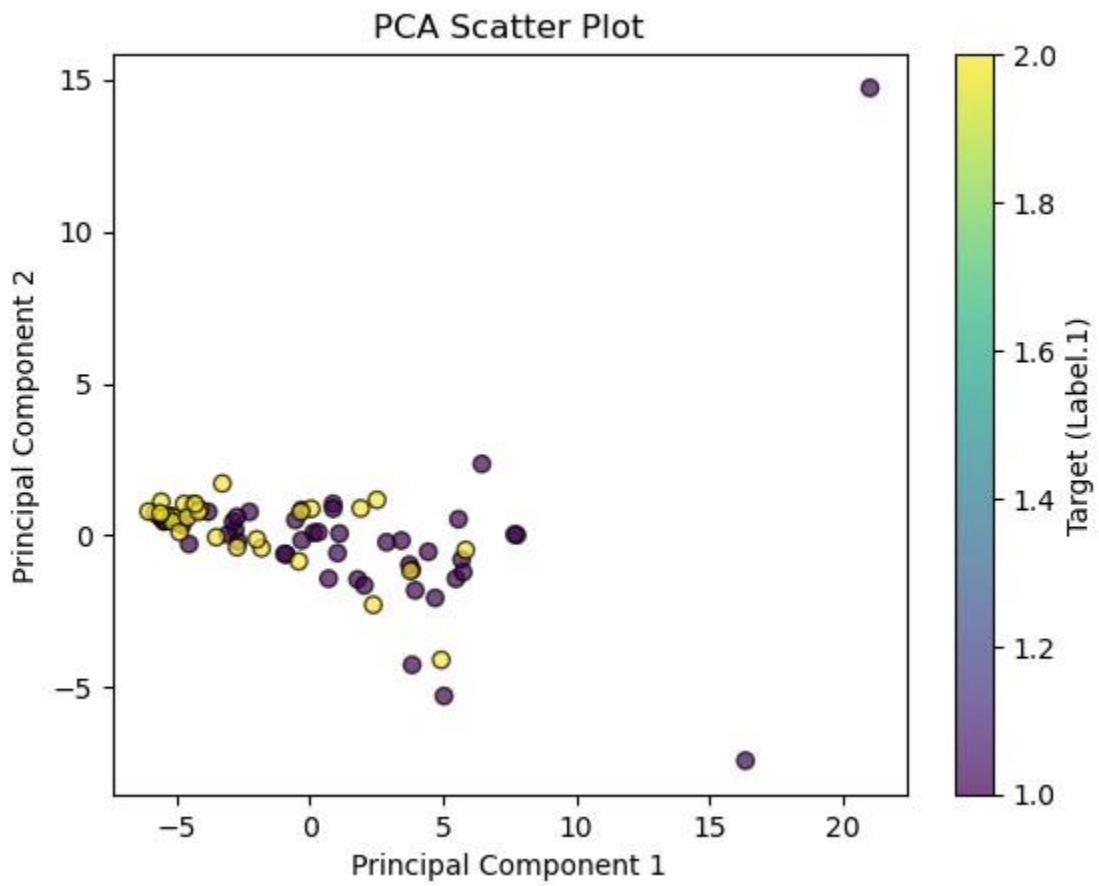

Figure 2.2.12 PCA (Principal Component Analysis) scatter plot

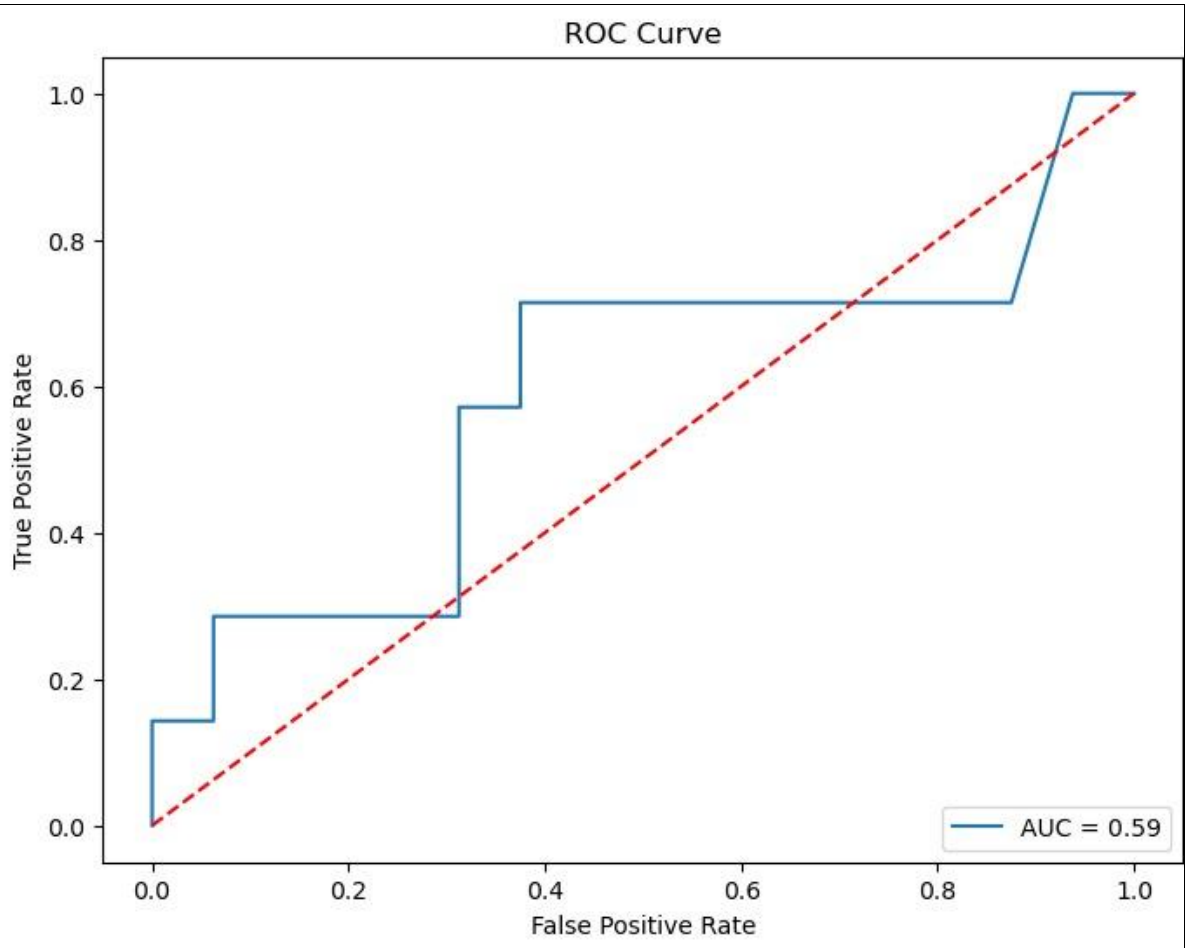

Figure 2.2.13ROC (Receiver Operating Characteristic) Curve, for evaluating the diagnostic ability of a binary classification model.

2.2.4 Program test, debug and optimization

Debugging

- The data loading process was debugged by ensuring the file path was correctly specified.
- Missing values were handled using forward fill to ensure no empty cells affected the analysis.

Optimization

- Data scaling using StandardScaler improved the PCA and model performance.
- Forward fill was chosen as an efficient strategy to handle missing values in the dataset.

- The Random Forest model was optimized with default parameters to balance accuracy and simplicity.

2.2.5 Task summary

This Python program analyses a cancer-associated skeletal muscle wasting dataset using 1H-NMR metabolite data. It involves data preparation, visualization, PCA for dimensionality reduction, and a Random Forest Classifier for predicting muscle loss. The program provides comprehensive analysis but could benefit from further model optimization and data transformation**.**

2.2.5.1 Program advantages

- Comprehensive Data Analysis: The script covers essential steps from data cleaning to predictive modelling.
- Visualization: Provides clear visual insights into data distribution and correlations.
- Machine Learning: Demonstrates a practical application of machine learning using Random Forest**.**

2.2.5.2 Program limitations

- Model Accuracy: The Random Forest model could be further fine-tuned to improve accuracy.
- Limited Feature Engineering: No additional feature engineering or hyperparameter tuning was performed.
- Data Handling: The forward fill method for missing values may not be the best strategy for all datasets**.**

REFERENCE

Cock PA, Antao T, Chang JT, Chapman BA, Cox CJ, Dalke A, Friedberg I, Hamelryck T, Kauff F, Wilczynski B and de Hoon MJL (2009) Biopython: freely available Python tools for computational molecular biology and bioinformatics. *Bioinformatics*, 25, 1422-1423

Cock PJ, Fields CJ, Goto N, Heuer ML and Rice PM (2009) The Sanger FASTQ file format for sequences with quality scores, and the Solexa/Illumina FASTQ variants. *Nucleic Acids Res.*, 38, 1767-1771

Eisner, R., Stretch, C., Eastman, T.B., Xia, J., Hau, D.D., Damaraju, S., Greiner, R., Wishart, D.S. and Baracos, V.E., 2011. Learning to predict cancer-associated skeletal muscle wasting from 1H-NMR profiles of urinary metabolites. *Metabolomics*, 7(1), pp.25–34.

Talevich E, Invergo BM, Cock PJ and Chapman BA (2012) Bio.Phylo: a unified toolkit for processing, analyzing and visualizing phylogenetic trees in Biopython. *BMC Bioinformatics*, 13, 209

Appendix: Declaration of Generative AI Use

If you DID use Generative AI technology in your Assignment, please provide detailed responses in the Appendix of your assignment to the following items (1-3) and include statement (4):

- Specify the reasons and purposes for using Generative AI technology in this assignment. Clearly explain how the Generative AI technology assisted in the development, writing, or editing processes.

- **While using AI I did not alter or used it in writing my work. Rather I used it to ensure that my work was accurately formatted in terms of language structuring and clarification of my work.**

  I used AI to gain better understanding on cases that was related to the task given and as well as the steps needed for certain task that I found difficult to handle

.

- By including this statement in my assessment submission, I attest that the information provided in this Originality and use of AI Declaration Statement is accurate and complete to the best of my knowledge. I understand that providing

false information is a violation of the University's Academic Misconduct Regulations and may result in academic and/or disciplinary consequences.

- Please note declarations, when incorporated into the Title Page and, if required, inclusion of the items (1-4) in the Appendix do not contribute to the assessment word count.