Universidad Distrital Francisco José de Caldas
Sebastian Pardo Tirado - 20221020121
Análisis de Sistemas

Workshop I - Systems Analysis: Entropy

1. Create a *dummy database* of genetic sequences composed of nucleotide bases (*A*, *C*, *G*, *T*), where each sequence must have between 10 and 20 bases. Your database must be composed for 50.000 genetic sequences.

1.1. A function is created to generate a random genetic sequence between 10 and 20 characters.

```python
import random


def create_sequence() -> str:
    """
    This function is used to generate a random genetic sequence.

    Returns:
    - str: random genetic sequence
    """
    nucleotid_bases = ["A", "C", "G", "T"]
    size_sequence = random.randint(10, 20)
    new_sequence = [nucleotid_bases[random.randint(0, 3)] for i in range(size_sequence)]
    return "".join(new_sequence)

print(create_sequence())
```

```
TAGTGATGTC
```

1.2. A data set composed of the previously defined genetic sequences is created.

```python
[4] def create_dataset(dataset_size: int) -> list:
        """
        This function is used to create a dataset composed by a set of genetic sequences.

        Parameters:
        - dataset_size (int): size of the dummy dataset to be generated.

        Returns:
        - list: a list of genetic sequences
        """
        dataset = [create_sequence() for i in range(dataset_size)]
        return dataset

    print(create_dataset(50000))
```

```
['ACTCCAATAGACGTCAAC', 'AAGCGGAATACAT', 'GTCTACCGAACCCATG', 'GTACTGTCATTTTTTT', 'CAGTACATATCCTTC',
```

2. Get the motifs (must be repeated sequence) of size 6 and 8.

2.1. A set of combinations is generated from a list of nucleotide bases. To facilitate the process, this function is defined as a recursion.

```python
def get_combinations(n: int, sequences: list, bases: list) -> list:
    """
    This method is used to generate a set of combinations based on a list of nucleotid bases.
    To make easy the process, this function is defined as a recurssion.

    Parameters:
    - n (int): amount of elements of each combination
    - sequences (list): list of recursive sequences obtained
    - bases (list): list of nucleotid bases to be used

    Returns:
    - list: list of combinations
    """
    if n == 1:
        return [sequence + base for sequence in sequences for base in bases]
    else:
        sequence_ = [sequence + base for sequence in sequences for base in bases]
        return get_combinations(n - 1, sequence_, bases)
print(get_combinations(6, [""], ["A", "C", "G", "T"]))
print(get_combinations(8, [""], ["A", "C", "G", "T"]))
```

```
['AAAAAA', 'AAAAAC', 'AAAAAG', 'AAAAAT', 'AAAACA', 'AAAACC', 'AAAACG', 'AAAACT', 'AAAAGA', 'AAAAGC',
['AAAAAAAA', 'AAAAAAAC', 'AAAAAAAG', 'AAAAAAAT', 'AAAAAACA', 'AAAAAACC', 'AAAAAACG', 'AAAAAACT', 'AA
```

2.2. The function is created to count the number of times a motif appears in a set of genetic sequences.

```python
def count_motif(motif: str, sequences_dataset: list) -> int:
    """
    This function is used to count the number of times a motif appears in a set of genetic sequences.

    Parameters:
    - motif (str): genetic motif to be searched.
    - sequences_dataset (list): list of genetic sequences.

    Returns:
    - int: number of times the motif appears in the dataset.
    """
    count = 0
    for sequence in sequences_dataset:
        count += sequence.count(motif)
    return count
```

2.3. The function used to obtain the motif with the highest count in a set of genetic sequences is created.

```
[ ]  from typing import Union
     def get_motif(motif_size: int, sequences_dataset: list) -> Union[str, int]:
         """
         This function is used to get the motif with the highest count in a set of genetic sequences.

         Parameters:
         - motif_size (int): size of the motif to be searched.
         - sequences_db (list): list of genetic sequences.

         Returns:
         - (str, int): motif with the highest count and the number of times it appears in the dataset.
         """
         nucleotid_bases = ["A", "C", "G", "T"]
         combinations = get_combinations(motif_size, [""], nucleotid_bases)
         # get motif with the highest count
         max_counter = 0
         motif_winner = ""
         for motif_candidate in combinations:
             temp_counter = count_motif(motif_candidate, sequences_dataset)
             if temp_counter > max_counter:
                 max_counter = temp_counter
                 motif_winner = motif_candidate

         return motif_winner, max_counter

     for size in [6, 8]:
         print(f"\nMotifs of size: {size}")
         for i in range(3):
             print(get_motif(size, create_dataset(50000)))
```

```
Motifs of size: 6
('TTAACC', 162)
('AATTGG', 161)
('GTCTGT', 177)

Motifs of size: 8
('ACCCGATC', 18)
('ACCCGCTC', 19)
('GCAAGAGT', 20)
```

3. Use the **Shannon Entropy** measurement to filter sequences with not a good variance level.

3.1. The function to calculate the Shannon Entropy of a genetic sequence is created.

```
[23] import math


     def calculate_shannon_entrophy(sequence: str) -> float:
         """
         This function is used to calculate the Shannon Entropy of a genetic sequence.

         Parameters:
         - sequence (str): genetic sequence.

         Returns:
         - float: Shannon Entropy of the sequence.
         """
         counts = {}
         total = 0
         for base in sequence:
             counts[base] = counts.get(base, 0) + 1
             total += 1

         entrophy = 0
         for count in counts.values():
             probability = count / total
             entrophy -= probability * math.log2(probability)

         return entrophy
```

3.2. The function used to filter genetic sequences according to their Shannon Entropy is created.

```
def filter_shannon(sequence: str, threshold: float) -> bool:
    """
    This function is used to filter genetic sequences based on their Shannon Entropy.

    Parameters:
    - sequence (str): genetic sequence.
    - threshold (float): the threshold value for entropy filtering.

    Returns:
    - bool: True if the sequence passes the filter, False otherwise.
    """
    entrophy = calculate_shannon_entrophy(sequence)
    return entrophy >= threshold
```

4. Get again the *motifs* of size 6 and 8, with threshold of 0.5.

```
entropy_threshold = 0.5
for size in [6, 8]:
    print(f"\nAfter filter, motifs of size: {size}")
    for i in range(10):
        dataset = create_dataset(50000)
        dataset = [seq for seq in dataset if filter_shannon(seq, entropy_threshold)]
        print(f"Dataset size: {len(dataset)}, Motif: {get_motif(size, dataset)}")
```

```
After filter, motifs of size: 6
Dataset size: 50000, Motif: ('GCCCCA', 164)
Dataset size: 50000, Motif: ('GTTGGC', 162)
Dataset size: 49998, Motif: ('CTGGTT', 161)
Dataset size: 49999, Motif: ('AGCGGG', 163)
Dataset size: 49998, Motif: ('CACTGG', 166)
Dataset size: 50000, Motif: ('CCAGAT', 169)
Dataset size: 49998, Motif: ('GCGTAA', 159)
Dataset size: 50000, Motif: ('GGTAGT', 160)
Dataset size: 49999, Motif: ('ACAGAT', 168)
Dataset size: 49999, Motif: ('GCTAGA', 174)

After filter, motifs of size: 8
Dataset size: 49998, Motif: ('ACTGAAAG', 22)
Dataset size: 49998, Motif: ('GGGCTGAG', 21)
Dataset size: 49999, Motif: ('GAATGCCA', 20)
Dataset size: 50000, Motif: ('GGTAAGAA', 20)
Dataset size: 49999, Motif: ('TTCGCACC', 20)
Dataset size: 50000, Motif: ('GCTTGGGC', 19)
Dataset size: 49999, Motif: ('GTCCGCGA', 21)
Dataset size: 49999, Motif: ('CATCAGGT', 19)
Dataset size: 49999, Motif: ('TAGTCAAT', 20)
Dataset size: 50000, Motif: ('ACTTTGTT', 21)
```

With an entropy threshold of 0.5, we can draw the following conclusion based on the analysis:

Sequences that pass the entropy filter are likely to contain more structured or conserved genomic regions. By setting a relatively low entropy threshold, there's a risk of filtering out sequences that contain variable or less-conserved regions. These regions may still hold valuable information, especially in the context of genetic diversity, population studies, or evolutionary analyses. Setting a lower entropy threshold increases the sensitivity of motif detection, as it allows more sequences to pass through the filter.

Conclusions.

In this project, we embarked on an analysis of genomic data with the objective of identifying motifs, or recurring patterns, within genetic sequences. The project was divided into several key steps:

1. **Database Creation:** We generated a dummy database of genetic sequences composed of nucleotide bases (A, C, G, T), with each sequence ranging from 10 to 20 bases in length.

The database consisted of 50,000 genetic sequences, providing a substantial dataset for analysis.

2. **Motif Discovery:** Using algorithms based on combinatorial approaches, we identified motifs of sizes 6 and 8 within the genetic sequences. These motifs represent recurring patterns that may hold biological significance, such as regulatory elements or protein-binding sites.
3. **Entropy-Based Filtering:** To enhance the specificity of motif discovery, we introduced a filtering step based on Shannon Entropy measurement. Sequences with low entropy, indicative of low variance or randomness, were filtered out to focus the analysis on more structured or conserved genomic regions.
4. **Re-evaluation of Motifs:** Following entropy-based filtering, we revisited motif discovery to identify motifs of sizes 6 and 8 from the filtered sequences. This step allowed us to refine our analysis and capture motifs that are more likely to be biologically relevant within the dataset.

Key Findings and Insights:

- Motifs obtained after entropy-based filtering are likely to represent highly conserved patterns within the dataset, offering valuable insights into genomic organization and function.
- The choice of entropy threshold (in this case, 0.5) influences the balance between sensitivity and specificity in motif discovery. Fine-tuning this threshold is essential to strike the right balance and minimize the inclusion of noise or false positives in the analysis.
- Technical considerations, such as computational efficiency and algorithm optimization, play a crucial role in scaling the analysis to larger genomic datasets and ensuring timely delivery of results.

Technical Challenges and Considerations:

- Validation of the chosen entropy threshold against known biological motifs or functional annotations is necessary to ensure the reliability and biological relevance of the results.
- Careful interpretation of the obtained motifs is required, taking into account biological context, evolutionary conservation, and potential functional significance.
- Implementing efficient algorithms and data structures is essential for handling large genomic datasets and optimizing computational resources.

In summary, this project represents a comprehensive analysis of genomic data, integrating motif discovery with entropy-based filtering to identify biologically relevant patterns within genetic sequences. While providing valuable insights into genomic organization and function, the project also highlights the importance of iterative refinement, validation, and interpretation in genomic analysis.