Progetto
Elementi di bioinformatica

Nicolas Chines – 899536

Indice

Parametri e modulo ArgumentParser

Parametri dello script:

Lo script implementato presenta la possibilità di passare tramite linea di comando 3 argomenti opzionali :

- **-f** deve essere un filepath di une file con estensione .fastq , esso verra utilizzato come file di input , in caso di parametro assente allora dovrà essere salvato all'interno della cartella contenente lo script un file .fastq e verrà utilizzato il suddetto file (in presenza di piu file .fastq nella cartella, verrà mostrato all'utente una scelta fra i vari file presenti)
- **-fr** soglia di frequenza F, essa deve essere un numero compreso tra 0 e1, in caso di parametro assente viene utilizzato un valore di default fr = 0.00142
- **-k** lunghezza dei k-meri , deve essere un numero intero positivo, in caso di parametro assente viene utilizzato un valore di default = 7

Esempi di esecuzione dello script da linea di comando :

C:\Users\nicol\OneDrive\Desktop\progetto bioinfo> python MainFastq.py -k 19 -fr 0.00005

I valori K e F saranno rispettivamente 19 e 0.00005 mentre il file .fastq dovrà essere salvato nella cartella "progetto bioinfo"

C:\...\progetto bioinfo> python MainFastq.py -k 19 -fr 0.00005 -f "C:\Users\nicol\OneDrive\Desktop\progetto bioinfo\SRR18961685-5000.fastq"

Come prima ma viene specificato quale file usare tramite percorso assoluto

C:\Users\nicol\OneDrive\Desktop\progetto bioinfo > python MainFastq.py

In questo caso non è stato specificato nessun valore e verranno utilizzati tutti i valori di default K = 7, F = 0.00142 e il file di input dovrà essere salvato nella cartella "progetto bioinfo"

Gestione dei parametri tramite il modulo argument_parser:

Argument_parser viene utilizzato per contenere tutte le funzioni utili alla gestione del parsing degli argomenti, esso utilizza principalmente la libreria argparse per implementare il parser e le librerie os e pathlib utilizzate per delle funzionalità necessarie alla verifica del formato del file e per la sua ricerca nel caso di default value.

È stato implementato come modulo separato ai fini di una maggiore leggibilità e un debugging più efficiente

Le funzioni presenti sono:

- Parse_arguments(), funzione che crea il parser aggiungendo gli argomenti e definendo per ognuno i valori di default e il campo type, in questo caso utilizzato richiamando funzioni personalizzate per ogni argomento cosi da verificarne il formato corretto
- check_frequency(), check_positive_integer() e check_fastq() funzioni utilizzate per verificare il formato degli argomenti -fr, -k e -f
- find_fastq_in_directory() funzione utilizzata per restituire i file presenti nella cartella come valore di default all'argomento -f
- choose_fastq_files() questa funziona è utilizzata per discriminare il file .fastq nel caso in cui ne vengano salvati più di uno all'interno della directory, generando una lista dei file presenti e lasciando all'utente la scelta di quale file utilizzare;
  questa funzione è stata separata dalla logica di find_fastq_in_directory() in quanto se la scelta del file fosse stata implementata in quella funzione, allora essa sarebbe stata sempre richiamata al momento della creazione del parser, in quanto il parser viene creato ed eseguito in ogni caso esso avrebbe mandato a video la scelta del file fastq anche quando si avesse chiamato lo script con l'argomento -h , cosi da generare un comportamento non richiesto ovvero la visualizzazione della lista dei file .fastq presenti nella directory durante il messaggio di help, per tanto ne è stata separata la logica, e essa viene richiamata dallo script principale in un secondo momento solo dopo l'avvenuto parsing degli argomenti

Classe FastqAnalyzer e metodi principali

Class FastqAnalyzer:

la logica principale di tutto lo script è implementata tramite la classe FastqAnalyzer, essa presenta i seguenti attributi definiti nel costruttore:

```
def __init__(self, file_path: str, frequency: float, kmer_length: int) -> None:
    self.file_path = file_path
    self.frequency = frequency
    self.kmer_length = kmer_length
    self.fastq_records = []
    self.sequences = []
    self.quality_scores = []
    self.kmer_dict = {}
    self.kmer_pos_dict = {}
    self.kmer_max = ""
    self.kmer_max_pos = 0
```

file_path, *frequency* e *kmer_length* contengono rispettivamente i parametri passati da linea di comando;

fastq_records, *sequences* e *quality_scores* sono liste utilizzate per salvare rispettivamente i record del file fastq in input, le sequenze presenti nei record e i valori di qualita dei reads

kmer_pos_dict contiene per ogni k-mero una lista con un contatore per ogni posizione i-esima ad indicare quante volte il k-mero compare in quella posizione, e sarà l'attributo più utilizzato in quanto utile a trovare il k-mero che compare più volte in una posizione, a calcolare la frequenza di ogni k-mero e a fare il plot dei grafici.

kmer_dict è un dizionario di dizionari di dizionari, contenente per ogni k-mero, ogni sequenza in cui compare e per ogni sequenza in quali posizioni compare; è stato scelto di utilizzare una struttura siffatta in quanto utile per l'ultimo punto del progetto in cui per cercare tutte le sequenze in cui *kmer_max* compare in posizione *kmer_max_pos* una ricerca sulle sequenze sarebbe stata molto lenta, cosi invece si ha diretto accesso alle posizioni in cui un k-mero compare per ogni sequenza; a migliorare ancora la prestazione è il fatto che entrambi i dizionari vengano calcolati con l'uso degli stessi cicli for cosi da svolgere tutto il calcolo solo all'inizio e solamente una volta, questo a leggero discapito della memoria.

kmer_max e *kmer_max_pos* contengo il k-mero che compare più volte in una posizione e la posizione in questione

Logica principale del progetto, i metodi di FastqAnalyzer:

load_fastq():

```
def load_fastq(self) -> None:
    fastq_records = list(SeqIO.parse(self.file_path, 'fastq'))
    if not fastq_records:
        raise ValueError(f"Il file {self.file_path} non contiene sequenze
valide.")

    self.fastq_records = fastq_records
    self.sequences = [str(record.seq) for record in fastq_records]
    self.quality_scores = [record.letter_annotations['phred_quality'] for record
in fastq_records]
```

il metodo utilizza la libreria Bio.SeqIO per effettuare il parsing del file in input e salva i record nella lista *fastq_records*, successivamente salva in *sequences* tutte le sequenze dei record e in *quality_scores* i valori di qualità per ogni record;

in caso il file sia vuoto o non contenga record validi viene sollevato un errore.

calculate_kmers():

```
def calculate_kmers(self) -> None:
    read_length = len(self.sequences[0])
    for seq_index, seq in enumerate(self.sequences):
        for i in range(0, read_length - self.kmer_length + 1):
            kmer = seq[i:i + self.kmer_length]

            # Aggiorna dizionario delle posizioni per ogni sequenza
            seq_dict = self.kmer_dict.get(kmer, {})
            pos_found = seq_dict.get(seq_index, {})
            pos_found[i] = 1
            seq_dict[seq_index] = pos_found
            self.kmer_dict[kmer] = seq_dict

            # Aggiorna conteggio per posizione
            pos_arr = self.kmer_pos_dict.get(kmer, [0] * read_length)
            pos_arr[i] += 1
            self.kmer_pos_dict[kmer] = pos_arr
```

il metodo popola i due dizionari *kmer_dict* e *kmer_pos_dict* per ogni sequenza utilizza una finestra scorrevole [i:i+K] per calcolare il k-mero presente in posizione *i* nella sequenza con indice *seq_index* ,

per popolare *kmer_dict* vengono utilizzati *seq_dict* e *pos_found* , che sono a loro volta due dizionari ausiliari ,*seq_dict* è il dizionario delle sequenze e contiene per ogni sequenza in cui compare un kmero il suo indice *seq_index* e il dizionario delle posizioni associato, pos_found è il dizionario delle posizioni e contiene per ogni posizione *i* il valore 1 se il kmero compare in quella posizione ; essi vengono di volta in volta popolati cosi che alla fine *kmer_dict* contenga per ogni kmero tutte le sequenze in cui compare e per ogni sequenza le posizioni in cui compare

kmer_pos_dict viene popolato utilizzando una lista di contatori, la prima volta che viene incontrato un kmero viene restituita una lista con tutti i valori a 0 lunga quanto la lunghezza di un read, successivamente verrà incrementato il contattore in posizione *i* ogni volta che viene incontrato un kmero in tale posizione

filter_kmers_by_frequency()

```
def filter kmers by frequency(self) -> None:
    total_positions = (len(self.sequences[0]) - self.kmer_length + 1) *
len(self.sequences)
    filtered_kmer_dict = {}
    filtered_kmer_pos_dict = {}

    for kmer, pos_array in self.kmer_pos_dict.items():
        frequency = sum(pos_array) / total_positions
        if frequency >= self.frequency:
            filtered_kmer_dict[kmer] = self.kmer_dict[kmer]
            filtered_kmer_pos_dict[kmer] = pos_array

    if not filtered_kmer_dict:
        raise ValueError("Nessun k-mero supera la soglia di frequenza
specificata. Modifica la soglia e riprova.")

    self.kmer_dict = filtered_kmer_dict
    self.kmer_pos_dict = filtered_kmer_pos_dict
```

Aggiorna i dizionari *kmer_dict* e *kmer_pos_dict* inserendo solo le informazioni dei k-meri che compaiono con una frequenza maggiore di F in due dizionari di supporto per poi ricopiarli nei dizionari originali , qualora nessun k-mero superi tale frequenza solleva un errore

total_position = le posizioni possibili per un kmero in una sequenza * tutte le sequenze

find_most_frequent_kmer():

```
def find_most_frequent_kmer(self) -> None:
    max_count = 0
    for kmer, positions in self.kmer_pos_dict.items():
        for i, count in enumerate(positions):
            if count > max_count:
                max_count = count
                self.kmer_max = kmer
                self.kmer_max_pos = i
```

il metodo seleziona il k-mero che appare più volte in una posizione, valutando per ogni k-mero il suo array delle posizioni, salva il k-mero in *kmer_max* e la posizione in *kmer_max_pos*

plot_all_kmers():

```
def plot_all_kmers(self) -> None:
    kmer_list = list(self.kmer_pos_dict.keys())
    print("K-mer disponibili:")
    for i, kmer in enumerate(kmer_list):
        print(f"{i}: {kmer}")
    while True:
        choice = input("Inserisci l'indice del k-mer da visualizzare (o 'exit'
per uscire): ").strip()
        if choice.lower() == 'exit':
            break
        try:
            index = int(choice)
            if 0 <= index < len(kmer_list):
                self.plot_kmer_occurrences(kmer_list[index])
            else:
                print("Indice non valido.")
        except ValueError:
            print("Input non valido.")
```

al posto di un report testuale dell'uso dei k-meri per posizione, si è preferito fornire un grafico per ogni singolo k-mero, il seguente metodo implementa tale funzionalità mostrando a video tutti i k-meri, e facendo scegliere all'utente il k-mero di cui si vuole mostrare il grafico, mostrandolo tramite la funzione plot_kmer_occurences();

viene fatto reinserire il valore in caso si inserisca un numero non presente nella lista e nel caso si inserisca un valore diverso da un numero

la funzionalità .strip() rimuove gli spazi in testa e in coda ad una stringa cosi da rilassare la funzione di input anche ai casi in cui venga inserito un valore valido con spazi prima o dopo il valore

plot_kmer_occurrences()

```
def plot_kmer_occurrences(self, kmer: str) -> None:
    y_values = self.kmer_pos_dict.get(kmer)
    x_values = list(range(len(y_values)))
    plt.figure(figsize=(15, 6), dpi=100)
    plt.bar(x_values, y_values, color='blue')
    plt.xlabel('Posizioni')
    plt.ylabel('Occorrenze')
    plt.title(f'Occorrenze del k-mer: {kmer}')
    plt.xticks(list(range(0, len(x_values), 5)))
    plt.show()
```

il metodo è abbastanza autoesplicativo, fornisce un grafico a barre delle occorrenze di un k-mero per ogni posizione dei read;

sull' asse delle **x** sono segnate le posizioni , viene usato len(y_values) in quanto esso contiene l'array dei contatori per ogni posizione di un read, per tanto è lungo quanto tutte le posizioni di un read per definizione

sull'asse delle **y** sono segnate le occorrenze per ogni posizione

save_fasta_output():

```
def save_fasta_output(self, output_path: str) -> None:
    output_list = []
    for seq_index, positions in self.kmer_dict.get(self.kmer_max).items():
        if self.kmer_max_pos in positions:
            record = self.fastq_records[seq_index]
            record.description += f"
quality_mean={statistics.mean(self.quality_scores[seq_index]):.2f}"
            output_list.append(record)

    SeqIO.write(output_list, output_path, 'fasta')
```

il metodo crea un file di output in formato FASTA , inserendo tutte le sequenze in cui *kmer_max* compare esattamente alla posizione *kmer_max_pos*

al contempo aggiunge nella descrizione di ogni sequenza la qualità media delle basi arrotondata alla seconda cifra decimale per poterla riportare poi nel header FASTA

per trovare tutte le sequenza viene effettuato un for sulle coppie *seq_index* , *positions*; contenenti rispettivamente l'indice delle sequenze in cui *kmer_max* compare e il dizionario delle posizioni in cui compare in quella determinata sequenza;

verificando poi se *kmer_max_pos* è presente come chiave in tale dizionario allora possiamo dire che la sequenza *seq_index* contiene *kmer_max* proprio in posizione *kmer_max_pos*

__main__, thread e animazioni

La funzione __main__:

```
if __name__ == "__main__":

    args = vars(ArgumentParser.parse_arguments())
    analyzer = FastqAnalyzer(
        file_path=ArgumentParser.chose_fastq_files(args['file']),
        frequency=args['frequency'],
        kmer_length=args['kmer']
    )

    def process_workload():
        analyzer.load_fastq()
        analyzer.calculate_kmers()

    calculate_kmer_process = threading.Thread(name='process',
target=process_workload)
    calculate_kmer_process.start()
    while calculate_kmer_process.is_alive():
        Animations.dna_animation("Calcolo k-meri in corso")

    analyzer.filter_kmers_by_frequency()
    analyzer.plot_all_kmers()
    analyzer.find_most_frequent_kmer()
    print(f"Il k-mero più frequente è '{analyzer.kmer_max}' nella posizione
{analyzer.kmer_max_pos}.")
    os.system("pause")
    analyzer.plot_kmer_occurrences(analyzer.kmer_max)

    output_file = input("Inserisci il nome del file di output (senza estensione,
verra salvato in automatico un file FASTA): ")
    analyzer.save_fasta_output(output_file + '.fasta')
    print(f"File salvato con successo!")
```

Rispetto alla logica principale vi è poco da dire, viene richiamato il parser, istanziato l'oggetto analyzer e vengono richiamate le funzionalità di FastqAnalyzer già presentate, cosi da mostrare il report di ogni k-mero, mostrare il grafico del k-mero che appare più volte in una posizione e successivamente salvare il file di output.

In aggiunta alla logica che ci aspettavamo è stata integrata una piccola animazione che viene eseguita in parallelo al caricamento del file di input ed al calcolo dei dizionari , in quanto queste due operazioni con valori di K specifici e con file in input molto grandi, potrebbero impiegare del tempo non indifferente,

per tanto le due funzioni sono state racchiuse in un thread, annidandole in un'unica funzione, ed è stata aggiunta un animazione tramite il modulo separato *Animations,* cosi che durante tali operazioni viene visualizzata una animazione molto carina di un elica di DNA .

Librerie e moduli

Nello svolgimento del progetto sono state utilizzate le seguenti librerie:

- OS
- Statistics
- Threading
- Matplotlib
- Biopython
- Pathlib
- Argparse
- Time

E sono stati implementati i seguenti moduli:

- Animations
- ArgumentParser

Le librerie fanno riferimento sia a quelle utilizzate nello script principale sia a quelle utilizzate dai moduli Animations e ArgumentParser

La libreria OS è stata utilizzata per le funzionalità Listdir, cosi da ricavare i file presenti sulla cartella corrente e System cosi da eseguire comandi sulla shell come ‘pause’ e ‘cls’, lo script per tanto è pensato per essere runnato su un sistema operativo windows, è comunque di facile estensione agli altri sistemi operativi…

La libreria Statistics viene utilizzata per il calcolo della media della qualità delle basi

La libreria Threading viene utilizzata per la creazione del thread presente nel main

La libreria Matplotlib viene utilizzata per la generazione dei diagrammi a barre

La libreria Biopython viene utilizzata per la lettura dei file in input, la scrittura dei file in output, e la gestione dei dati come record, sequenze e qualità dei read

La libreria Pathlib viene utilizzata per la verifica del formato dell’argomento -f

La libreria Argparse viene utilizzata per il parsing degli argomenti da linea di comando

La libreria Time viene utilizzata per implementare delle pause durante l’esecuzione dell’animazione cosi da visualizzare correttamente i frame

Il modulo ArgumentParser come gia presentato implementa la funzionalità per il parsing degli argomenti.

Il modulo Animations implementa un'unica funzione , dna_animations() che manda a video un animazione su terminale di un elica di DNA che gira su se stessa e di un messaggio di caricamento personalizzato.

Entrambi sono stati implementati come moduli separati cosi da garantire una migliore leggibilità del codice, un debugging più efficiente e una separazione più distinta della logica del programma dal resto delle funzionalità.

*in tutto il codice si è scelto di tipizzare i parametri ed i valori di ritorno delle funzioni, per documentazione delle stesse, e per funzionalità interne all’IDE utilizzato(*PyCharm*) che fornisce degli warning in caso di assegnamenti scorretti.
