CS482 PS1 report

Arvin Asgharian Rezaee (a29asgha)

January 2025

1 Preliminary

The assignment was done using the following setup for python. //

```
> python --version
Python 3.12.8
```

all of the given files don't use any additional libraries other than those provided by base Python. Keep in mind some files have "–verbose" as an option to showcase more details about the computation for the purpose of debugging. while using this option you should have **Numpy** installed.

2 edit_distance.py

To solve this problem, we would need to find the total number of gaps and mismatches present in the best alignment match between the sequences. we'll need to implement a global alignment matching algorithm for the lectures to find the best alignment. The pseudo-code provided from the lectures was used to recreate an accurate Python script which is present in **find_best_alignment** function.
Furthermore **create_best_alignment** uses these results to show construct the best alignment. Then **calculate_edit_distance** will use the matrix **B** to calculate the distance. below you can find the runtime for all of them

```
// assuming n >= m
len(seq1) <- n
len(seq2) <- m

runtime(find_best_alignment) = O(mn)
runtime(create_best_alignment) = O(n)
runtime(calculate_edit_distance) = O(n)
```

3 fit_alignment.py

This solution borrows a lot of implementation from the first question. The main difference is that we use the Ford-Waterman algorithm to find a local best alignment. we modify the algorithm from the previous section, to use a zero in the matrix when all the other options are negative values. All of these are implemented in **fit_best_alignment** function. The rest of the functions are used to recreate the optimal alignments similar to the ones in the previous section and have similar runtime efficiency.

4 search_varient.py

In this question, the extending step approach uses a match score of +1 and a mismatch penalty -1. The extending algorithm will extend until 3 misses are seen, which were found to have the best results by try and error, with dummy values for candidates and database entries. furthermore, a HashMap was used to create an inverted index for each "mer" inside of the query. Then the extending algorithm tries every match inside of the query, to find the best match. below are the run times for each function:

```
// assuming n >= m
len(seq1) <- n
len(seq2) <- m
k <- k

runtime(search_best_match) = O(mn / k)
runtime(calculate_score) = O(m * n)
runtime(build_k_mers) = O(n)
```