Yihang Fu
yf329
yihang.fu@yale.edu

Assignment 3
BIS 634 Computational Methods for Informatics

2024/12/07

1 Exercise 1

```python
def align(seq1, seq2, match=1, gap_penalty=1, mismatch_penalty=1):
    m, n = len(seq1), len(seq2)
    # Initialize DP matrix and traceback indicators
    dp = [[0] * (n + 1) for _ in range(m + 1)]
    traceback = [[None] * (n + 1) for _ in range(m + 1)]

    max_score = 0
    max_pos = None

    # DP matrix
    for i in range(1, m + 1):
        for j in range(1, n + 1):
            if seq1[i - 1] == seq2[j - 1]:
                match_score = dp[i - 1][j - 1] + match
            else:
                match_score = dp[i - 1][j - 1] - mismatch_penalty

            gap_seq1 = dp[i - 1][j] - gap_penalty
            gap_seq2 = dp[i][j - 1] - gap_penalty

            dp[i][j] = max(0, match_score, gap_seq1, gap_seq2)

            # Store traceback direction
            if dp[i][j] == match_score:
                traceback[i][j] = "DIAG"
            elif dp[i][j] == gap_seq1:
                traceback[i][j] = "UP"
            elif dp[i][j] == gap_seq2:
                traceback[i][j] = "LEFT"

            # Update max score position
            if dp[i][j] > max_score:
                max_score = dp[i][j]
                max_pos = (i, j)

    # Traceback to count the optimal alignment
    aligned_seq1 = []
    aligned_seq2 = []
    i, j = max_pos

    while dp[i][j] != 0:
        if traceback[i][j] == "DIAG":
            aligned_seq1.append(seq1[i - 1])
            aligned_seq2.append(seq2[j - 1])
            i -= 1
            j -= 1
        elif traceback[i][j] == "UP":
            aligned_seq1.append(seq1[i - 1])
            aligned_seq2.append("-")
            i -= 1
        elif traceback[i][j] == "LEFT":
            aligned_seq1.append("-")
            aligned_seq2.append(seq2[j - 1])
            j -= 1

    # Reverse the sequences (since we traceback from the end)
    aligned_seq1 = ''.join(reversed(aligned_seq1))
    aligned_seq2 = ''.join(reversed(aligned_seq2))

    return aligned_seq1, aligned_seq2, max_score
```

Listing 1: Dynamic Programming Smith-Waterman

Answers and Explanations

Implementation

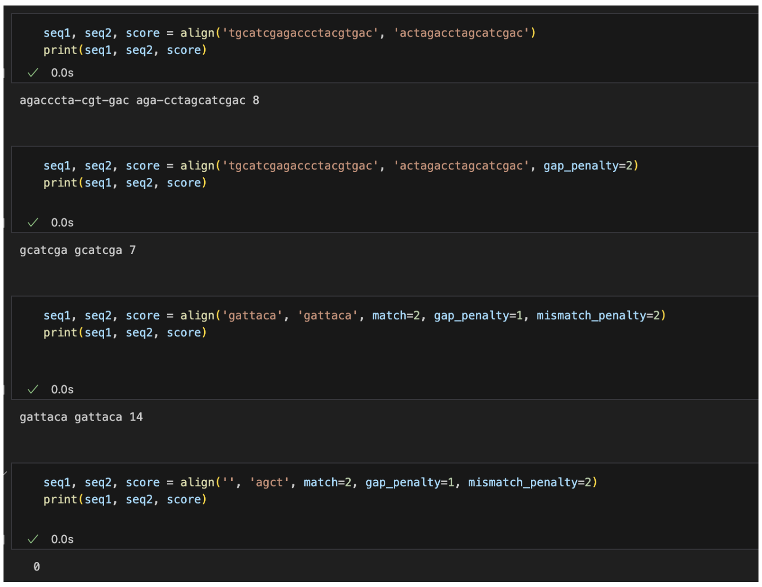
The implementation of the Smith-Waterman algorithm correctly computes the local alignment between two sequences, taking into account matches, mismatches, and gap penalties. The function uses a dynamic programming (DP) approach to compute the optimal local alignment and score. It fills the DP matrix using the recurrence relation:

$$\text{dp}[i][j] = \max(0, \text{dp}[i-1][j-1] + \text{match/mismatch\_score}, \text{dp}[i-1][j] - \text{gap\_penalty}, \text{dp}[i][j-1] - \text{gap\_penalty}),$$

where: - Matches add the `match` score. - Mismatches subtract the `mismatch_penalty`. - Gaps subtract the `gap_penalty`.
Traceback begins from the cell with the maximum score and continues until reaching a score of 0, ensuring correct local alignment reconstruction.

Tests and Results

The function was tested on various cases to validate its correctness and flexibility. For default parameters, the alignment and score matched expected results, such as aligning `tgcatcgagaccctacgtgac` with `actagacctagcatcgac` yielding a score of 8. Adjusting the gap penalty (e.g., `gap_penalty=2`) produced shorter alignments, such as `gcatcga` with a score of 7. For identical sequences like `gattaca`, with custom penalties (`match=2, gap_penalty=1, mismatch_penalty=2`), the function correctly aligned the sequences with a score of 14. Furthermore, inspired by CHATGPT, we also test edge cases, including empty sequences, which returned an empty alignment and a score of 0, ensuring robust handling of various inputs. These tests demonstrate that the function performs as expected across a wide range of scenarios.



```
seq1, seq2, score = align('tgcatcgagaccctacgtgac', 'actagacctagcatcgac')
print(seq1, seq2, score)
✓ 0.0s
agacccta-cgt-gac aga-cctagcatcgac 8

seq1, seq2, score = align('tgcatcgagaccctacgtgac', 'actagacctagcatcgac', gap_penalty=2)
print(seq1, seq2, score)
✓ 0.0s
gcatcga gcatcga 7

seq1, seq2, score = align('gattaca', 'gattaca', match=2, gap_penalty=1, mismatch_penalty=2)
print(seq1, seq2, score)
✓ 0.0s
gattaca gattaca 14

seq1, seq2, score = align('', 'agct', match=2, gap_penalty=1, mismatch_penalty=2)
print(seq1, seq2, score)
✓ 0.0s
  0
```

Figure 1: Test Result

Testing Explanation

The test cases demonstrate that the function handles default and customized scoring schemes correctly:

- With default parameters, the function identifies the optimal alignment and calculates the correct score.
- By increasing the gap penalty, the alignment length decreases, and the score adjusts accordingly, reflecting the expected behavior.

The outputs align with theoretical expectations for the Smith-Waterman algorithm and validate the function's correctness.

2 Exercise 2

This `QuadTree` implementation [1] organizes 2D points into a hierarchical tree structure to enable efficient k-nearest neighbor searches. The tree divides a rectangular region (defined by `x_bounds` and `y_bounds`) into four quadrants if the number of points exceeds `max_points`. Points are recursively assigned to child nodes, following the hint to "split into children if enough points exist." Then, the `_within_distance` method ensures efficient searching by checking if a point lies within a distance d (Traditional Euclidean Distance) from the node's bounds, as suggested by the hint to use distance-based containment checks. Another section is the `query` method recursively retrieves k-nearest neighbors by exploring only relevant child nodes. It avoids unnecessary descent into regions that cannot contain neighbors. Finally, after collecting candidates, they are sorted by Euclidean distance (use squared distance here to avoid the overlapping point calculation error), and the closest k points are returned.

```python
# QuadTree Implementation
class QuadTree:
    def __init__(self, points, x_bounds, y_bounds, max_points=4):
        self.points = points
        self.x_bounds = x_bounds
        self.y_bounds = y_bounds
        self.children = []
        self.max_points = max_points
        self._split()

    def _split(self):
        if len(self.points) <= self.max_points:
            return
        # Calculate midpoints and define four quadrants for splitting each region
        mid_x = (self.x_bounds[0] + self.x_bounds[1]) / 2
        mid_y = (self.y_bounds[0] + self.y_bounds[1]) / 2
        quadrants = [
            ([], (self.x_bounds[0], mid_x), (self.y_bounds[0], mid_y)),
            ([], (mid_x, self.x_bounds[1]), (self.y_bounds[0], mid_y)),
            ([], (self.x_bounds[0], mid_x), (mid_y, self.y_bounds[1])),
            ([], (mid_x, self.x_bounds[1]), (mid_y, self.y_bounds[1])),
        ]
        # Distribute points into appropriate quadrants and recursively create child QuadTree nodes for non-empty quadrants
        for point in self.points:
            x, y, _ = point
            for quadrant in quadrants:
                if quadrant[1][0] <= x <= quadrant[1][1] and quadrant[2][0] <= y <= quadrant[2][1]: ##check 'contains'
                    quadrant[0].append(point)
                    break
        self.points = []
        for q_points, q_x_bounds, q_y_bounds in quadrants:
            if q_points:
                self.children.append(QuadTree(q_points, q_x_bounds, q_y_bounds, self.max_points))

    def _within_distance(self, x, y, d):
        x_min, x_max = self.x_bounds
        y_min, y_max = self.y_bounds
        closest_x = min(max(x, x_min), x_max)
        closest_y = min(max(y, y_min), y_max)
        return (closest_x - x) ** 2 + (closest_y - y) ** 2 <= d ** 2

    def query(self, x, y, k):
        candidates = []
        def search(node):
            if not node.children:
                candidates.extend(node.points)
            else:
                for child in node.children:
                    if child._within_distance(x, y, float('inf')):
                        search(child)
        search(self)
        candidates.sort(key=lambda p: (p[0] - x) ** 2 + (p[1] - y) ** 2)
        return candidates[:k]
```

Listing 2: QuadTree implementation

[1] I wrote the body part of the QuadTree implementation following the Youtube Video https://www.youtube.com/watch?v=iG1o0qOyZlA and our slides. Some minor problems are debugged by CHATGPT (I fell into the Infinite Splitting and sometimes got stuck when implementing the 'check-contain' process)

Then we take advantage of QuadTree data structure to do Nearest Neighbor Searching. The `knn_predict` function queries the `QuadTree` to find the k-nearest neighbors for each test point. It predicts the most common class among these neighbors using a majority 'vote'.

```python
# Build the QuadTree
quadtree = QuadTree(train_data, (x_min, x_max), (y_min, y_max))

# KNN prediction function
def knn_predict(quadtree, test_points, k):
    predictions = []
    for x, y, _ in test_points:
        neighbors = quadtree.query(x, y, k)
        classes = [neighbor[2] for neighbor in neighbors]
        predictions.append(Counter(classes).most_common(1)[0][0])
    return predictions
```

Listing 3: KNN implementation

Before implementing KNN on our rice data, we do basic preprocessing. Normalize the seven quantitative columns to a mean of 0 and standard deviation 1. Reduce the data to two dimensions using PCA.

```python
# Load the dataset
file_path = '/Users/yihang/Desktop/BIS_634/HW5/Rice_Dataset_Commeo_and_Osmancik/
    Rice_Cammeo_Osmancik.xlsx'
data = pd.read_excel(file_path)

# Normalize the features
features = ['Area', 'Perimeter', 'Major_Axis_Length', 'Minor_Axis_Length',
            'Eccentricity', 'Convex_Area', 'Extent']
scaler = StandardScaler()
data_normalized = scaler.fit_transform(data[features])

# Apply PCA to reduce dimensions to 2 PCs
pca = PCA(n_components=2)
data_reduced = pca.fit_transform(data_normalized)
data['PC0'], data['PC1'] = data_reduced[:, 0], data_reduced[:, 1]
#I added two extra columns to the original data
```

Listing 4: Data preprocessing

The scatterplot after PCA shows that the two rice types (Cammeo and Osmancik) are reasonably well-separated in the reduced 2D space, with clusters corresponding to each type. However, there is some overlap between the clusters. The overlapping indicates that some points near the boundary will likely be misclassified, especially when the true class distributions are close in the feature space. Do a train-test split with test size = 20%.

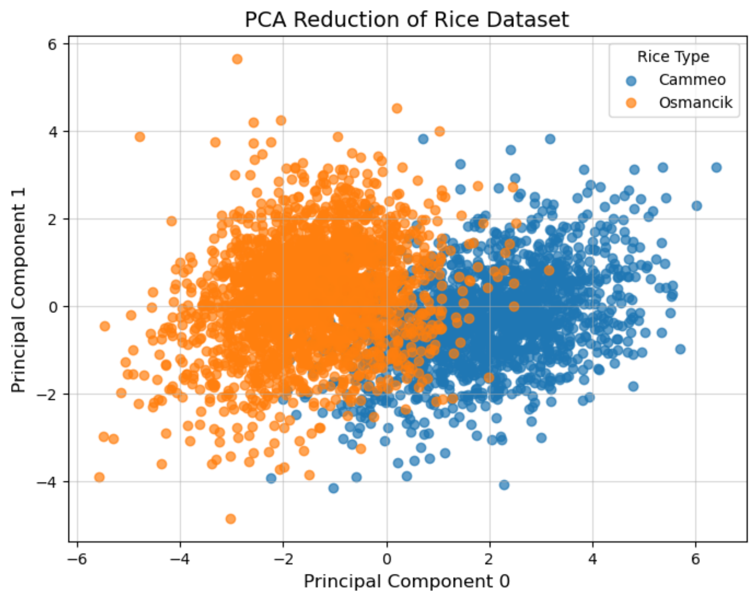


Figure 2: PCA visualization

```python
train_data, test_data = train_test_split(data[['PC0', 'PC1', 'Class']].to_numpy(), test_size
    =0.2, random_state=42)
x_min, x_max = train_data[:, 0].min(), train_data[:, 0].max()
y_min, y_max = train_data[:, 1].min(), train_data[:, 1].max()
```

```

# Build the QuadTree
quadtree = QuadTree(train_data, (x_min, x_max), (y_min, y_max))

# Test KNN with k=1 and k=5
test_labels = test_data[:, 2]
predictions_k1 = knn_predict(quadtree, test_data, k=1)
predictions_k5 = knn_predict(quadtree, test_data, k=5)

# Confusion Matrices, this and the visualization below are assisted by CHATGPT
confusion_k1 = pd.crosstab(pd.Series(test_labels, name='Actual'), pd.Series(predictions_k1,
    name='Predicted'))
confusion_k5 = pd.crosstab(pd.Series(test_labels, name='Actual'), pd.Series(predictions_k5,
    name='Predicted'))


plt.figure(figsize=(8, 6))
for rice_type, group in data.groupby('Class'):
    plt.scatter(group['PC0'], group['PC1'], label=rice_type, alpha=0.7)

plt.title('PCA Reduction of Rice Dataset', fontsize=14)
plt.xlabel('Principal Component 0', fontsize=12)
plt.ylabel('Principal Component 1', fontsize=12)
plt.legend(title='Rice Type')
plt.grid(True, alpha=0.5)
plt.show()
# Heatmaps for confusion matrices
plt.figure(figsize=(12, 5))
plt.subplot(1, 2, 1)
sns.heatmap(confusion_k1, annot=True, fmt='d', cmap='Blues', cbar=False)
plt.title('Confusion Matrix (k=1)', fontsize=14)
plt.xlabel('Predicted', fontsize=12)
plt.ylabel('Actual', fontsize=12)
plt.subplot(1, 2, 2)
sns.heatmap(confusion_k5, annot=True, fmt='d', cmap='Blues', cbar=False)
plt.title('Confusion Matrix (k=5)', fontsize=14)
plt.xlabel('Predicted', fontsize=12)
plt.ylabel('Actual', fontsize=12)
plt.tight_layout()
plt.show()
```

Listing 5: data preparation

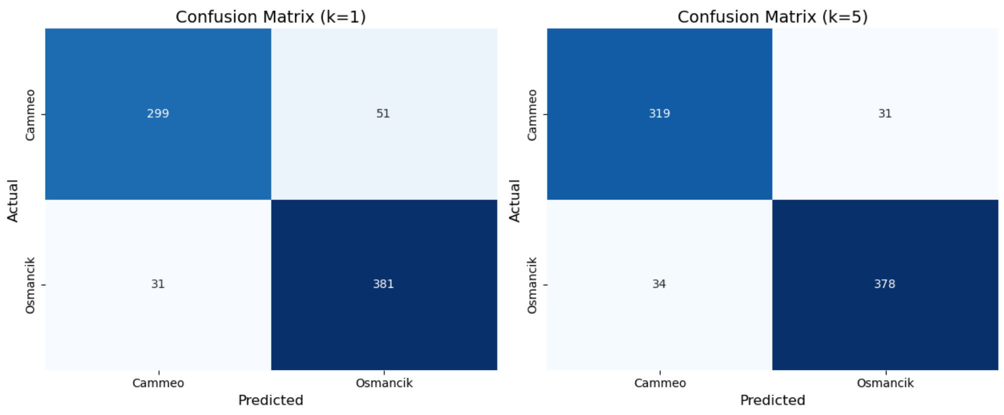


Figure 3: Confusion Matrix

As shown in figure 3, for $k = 1$, the model correctly classifies 299 instances of Cammeo and 381 instances of Osmancik, but misclassifies 51 Cammeo instances as Osmancik and 31 Osmancik instances as Cammeo. This indicates good performance but highlights the sensitivity of $k = 1$ to noise and boundary cases, leading to higher misclassification rates. For $k = 5$, the model correctly classifies 319 Cammeo and 378 Osmancik instances, with 31 and 34 misclassifications, respectively. This shows that increasing k in this case improves robustness by reducing the influence of noise

3 Exercise 3

```
from mpi4py import MPI
```

```python
import numpy as np
import matplotlib.pyplot as plt
import time

# Define bounds and parameters
xlo, ylo = -2.5, -1.5
xhi, yhi = 0.75, 1.5
nx, ny = 2048, 1536
dx, dy = (xhi - xlo) / nx, (yhi - ylo) / ny
iter_limit = 200
set_threshold = 2

def mandelbrot_test(x, y):
    z = 0
    c = x + y * 1j
    for i in range(iter_limit):
        z = z ** 2 + c
        if abs(z) > set_threshold:
            return i
    return i

def calculate_chunk(start, end):
    """
    Calculates a chunk of the Mandelbrot set rows from 'start' to 'end'.
    """
    local_result = np.zeros([end - start, nx])
    for i, row in enumerate(range(start, end)):
        y = row * dy + ylo
        for j in range(nx):
            x = j * dx + xlo
            local_result[i, j] = mandelbrot_test(x, y)
    return local_result

def calculate_serial():
    """
    Serial implementation of Mandelbrot set calculation for comparison.
    """
    result = np.zeros([ny, nx])
    for i in range(ny):
        y = i * dy + ylo
        for j in range(nx):
            x = j * dx + xlo
            result[i, j] = mandelbrot_test(x, y)
    return result

if __name__ == "__main__":
    # Initialize MPI
    comm = MPI.COMM_WORLD
    rank = comm.Get_rank()
    size = comm.Get_size()

    # Determine workload per process
    rows_per_process = ny // size
    start_row = rank * rows_per_process
    end_row = (rank + 1) * rows_per_process if rank != size - 1 else ny
    start_time = time.perf_counter()

    # Each process calculates its chunk
    local_result = calculate_chunk(start_row, end_row)

    # Gather all chunks to the root process
    if rank == 0:
        mandelbrot_set = np.zeros([ny, nx])
    else:
        mandelbrot_set = None

    comm.Gather(local_result, mandelbrot_set, root=0)
    stop_time = time.perf_counter()

    # Perform correctness and performance comparison
    if rank == 0:
        print(f"Parallel calculation took {stop_time - start_time:.2f} seconds with {size}
    processes")

        # Serial calculation for comparison
        serial_start_time = time.perf_counter()
        serial_result = calculate_serial()
        serial_stop_time = time.perf_counter()
        print(f"Serial calculation took {serial_stop_time - serial_start_time:.2f} seconds")

        # Check exact correctness,debugged by CHATGPT
        if np.array_equal(mandelbrot_set, serial_result):
            print("Parallel and serial results match")
        else:
            print("Mismatch between parallel and serial results.")

        # Plot the results
        plt.imshow(mandelbrot_seat, interpolation="nearest", cmap="Greys")
        plt.gca().set_aspect("equal")
        plt.axis("off")
        plt.show()
```

Listing 6: code implementation of exercise 3

The parallel implementation of the Mandelbrot set calculation was verified by comparing the results of the parallel and serial executions. For all test cases ($n = 1, 2, 4$), the parallel and serial outputs matched exactly, demonstrating the correctness of the parallel approach (cannot use '==' directly, otherwise will return a list of boolean values).
To run the code, the command will look like this

```
    mpiexec -n 4 python /Users/yihang/Desktop/BIS_634/HW5/question3.py
```

The execution time of the parallel implementation was observed to improve as the number of processes increased. For $n = 1$, the parallel computation took approximately 6.76 seconds. When $n = 2$, the computation time decreased to 3.52 seconds, and for $n = 4$, it further reduced to 3.25 seconds. And the corresponding serial calculation took 7.10, 7.56, 7.42 seconds separately. These results highlight the performance benefits of distributing the workload across multiple processes.
The parallel implementation divides the rows of the Mandelbrot set among the available processes using MPI. Each process computes a subset of rows independently, and the results are gathered at the root process. This approach minimizes computation time by leveraging multiple processes and ensures correctness through careful synchronization using `MPI.Gather`.