ATIAM - Fundamentals project - Quick Report

KANG Jiale

Sequence alignment

Part 1 - Exploration

Quick Sort Algorithm

Quick Sort works by selecting a "pivot" element from the array and then partitioning the array into two parts. Elements less than the pivot are moved to its left, and elements greater than the pivot are moved to its right. This partitioning step is recursively applied to each sub-array until the entire array is sorted.

Code

```
def my_sort(array):
    sorted_array = array.copy()
    left = 0
    right = len(sorted_array) - 1
    if left >= right:
        return sorted_array
    stack = []
    while stack or left < right:
        if left < right:
            pivot = left
            i, j = left, right
            while i < j:
                while sorted_array[pivot] <= sorted_array[j] and i < j:
                    j -= 1
                while sorted_array[pivot] >= sorted_array[i] and i < j:
                    i += 1
                if i < j:
                    sorted_array[i], sorted_array[j] = sorted_array[j], sorted_array[i]
            sorted_array[pivot], sorted_array[i] = sorted_array[i], sorted_array[pivot]
            stack.append((left, i, right))
            right = i - 1
        else:
            left, mid, right = stack.pop()
            left = mid + 1
    return sorted_array
```

Explanation of the Code

1. Initialization: First, creates a copy of the input array to avoid modifying

the original data. `left` and `right` pointers are initialized to the start and end of the array, respectively. The algorithm immediately returns if the array has one or no elements (i.e., when `left >= right`).

2. Loop Control: A `stack` is used to manage the subarrays that need to be sorted, simulating the recursive calls in a traditional quick sort. This allows the code to be written iteratively without actual recursion.
3. Partitioning: The main idea of partitioning is to rearrange elements in such a way that, given a `pivot`, elements smaller than or equal to the pivot move to the left, while elements greater move to the right. Here:
   - `pivot`, `i`, and `j` are pointers used to control the partitioning.
   - `pivot` is initially set to left.
   - `i` and `j` are used to scan from both ends toward the pivot.
   - The inner `while` loops move `j` to the left if the element at `j` is greater than the pivot and move `i` to the right if the element at `i` is less than the pivot.
4. Swapping Elements: When `i` and `j` meet, the pivot element is placed in its correct position within the array. Elements that are on the incorrect side of the pivot (smaller elements on the right or larger on the left) are swapped.
5. Stack Management: After partitioning, the code pushes the current state (`left`, `i`, `right`) onto the stack and updates `right` to `i - 1` for the next iteration. When the `left` pointer is greater than `right`, the algorithm pops from the stack to continue sorting the remaining subarrays.
6. Final Output: The process repeats until the stack is empty, at which point the sorted array is returned.

Part 2 - Symbolic alignments

Needleman-Wunsch Algorithm

The Needleman-Wunsch algorithm finds the optimal alignment between two sequences by maximizing a similarity score based on matches, mismatches, and gaps (insertions or deletions). The method uses a scoring matrix to determine match and mismatch scores and gap penalties for opening and extending gaps.

Code

```
def my_needleman_simple(str1, str2, matrix='atiam-fpa_alpha.dist', gap_open=-5, gap_extend=-
    score = 0
    # initialization
    dist_matrix = pd.read_csv(matrix, sep='\s+', index_col=0)
    gap = gap_open
    n = len(str1)
    m = len(str2)
    score_matrix = np.zeros((n+1, m+1))
    # fill the first row and column
    for i in range(1, n+1):
```

```
    score_matrix[i][0] = score_matrix[i-1][0] + gap
for j in range(1, m+1):
    score_matrix[0][j] = score_matrix[0][j-1] + gap
# fill the matrix
for i in range(1, n+1):
    for j in range(1, m+1):
        match = score_matrix[i-1][j-1] + dist_matrix[str1[i-1]][str2[j-1]]
        delete = score_matrix[i-1][j] + gap
        insert = score_matrix[i][j-1] + gap
        score_matrix[i][j] = max(match, delete, insert)
# traceback
align1 = ""
align2 = ""
i, j = n, m
while i > 0 and j > 0:
    score = score_matrix[i][j]
    score_diag = score_matrix[i-1][j-1]
    score_up = score_matrix[i-1][j]
    score_left = score_matrix[i][j-1]
    if score == score_diag + dist_matrix[str1[i-1]][str2[j-1]]:
        align1 = str1[i-1] + align1
        align2 = str2[j-1] + align2
        i -= 1
        j -= 1
    elif score == score_up + gap:
        align1 = str1[i-1] + align1
        align2 = "-" + align2
        i -= 1
    elif score == score_left + gap:
        align1 = "-" + align1
        align2 = str2[j-1] + align2
        j -= 1
while i > 0:
    align1 = str1[i-1] + align1
    align2 = "-" + align2
    i -= 1
while j > 0:
    align1 = "-" + align1
    align2 = str2[j-1] + align2
    j -= 1
score = score_matrix[n][m]
return (align1, align2, score)
```

Explanation of the Code

1. Scoring Matrix Initialization: First, reads a scoring matrix from a file, where each entry defines the score for aligning each possible character pair. `score_matrix`: Initializes a 2D array of zeros to store the alignment scores for subproblems. It has dimensions (n+1, m+1), where n and m are the lengths of the sequences.
2. Filling the First Row and Column: The first row and first column represent alignments where one sequence is aligned entirely with gaps. Each cell in the first row and column is filled with cumulative gap penalties, assuming that a gap is opened at each step.
3. Dynamic Programming Matrix Filling: Each cell `(i, j)` in `score_matrix` represents the optimal alignment score for the subsequences `str1[0:i]` and `str2[0:j]`. For each cell `(i, j)`, three options are considered:
   - Match or Mismatch: The score from the diagonal cell `score_matrix[i-1][j-1]` plus the match/mismatch score from `dist_matrix`.
   - Deletion: The score from the cell directly above `score_matrix[i-1][j]` plus the gap penalty.
   - Insertion: The score from the cell directly to the left `score_matrix[i][j-1]` plus the gap penalty.

   The maximum of these three values is stored in `score_matrix[i][j]`.
4. Traceback for Alignment Construction: Starting from the bottom-right cell, the algorithm traces back to reconstruct the optimal alignment. At each step:
   - If the score at `(i, j)` matches the diagonal score plus the match/mismatch, a match/mismatch is added.
   - If it matches the score above plus a gap penalty, a deletion (gap in `str2`) is added.
   - If it matches the score to the left plus a gap penalty, an insertion (gap in `str1`) is added.

   This traceback process continues until reaching the top-left cell `(0, 0)`, with additional steps to add remaining gaps if one sequence is shorter.

Part 3 - Musically-informed track names alignment

RE based information extraction

The goal of this algorithm is to:

1. Extract structured information (e.g., title, key, instrument, opus) from two unstructured input strings describing musical compositions.
2. Compute a similarity score based on the presence and matching of these extracted attributes using custom weights.
3. Transform input strings to improve consistency in comparison, focusing on case and character normalization.

Code

```python
# define weight matrix
SYMBOL_WEIGHTS = {
    'title': 3,
    'key': 2,
    'instrument': 2,
    'opus': 1
}

# Preprocess and extract symbols
def extract_symbols(name):
    symbols = {
        'title': None,
        'key': None,
        'instrument': None,
        'opus': None
    }
    # Simple regexes for common patterns (expandable)
    title_match = re.search(r"(symphony|concerto|sonata|etude)", name)
    key_match = re.search(r"in ([a-gA-G](?: flat| sharp)?(?: major| minor))", name)
    instrument_match = re.search(r"for|pour (.+?)\b", name)
    opus_match = re.search(r"(\d+)", name)

    if title_match:
        symbols['title'] = title_match.group(1)
    if key_match:
        symbols['key'] = key_match.group(1)
    if instrument_match:
        symbols['instrument'] = instrument_match.group(1)
    if opus_match:
        symbols['opus'] = opus_match.group(1)

    return symbols

# Compute alignment score with custom weights
def weighted_score(symbols1, symbols2):
    score = 0
    for symbol, weight in SYMBOL_WEIGHTS.items():
        if symbols1[symbol] and symbols2[symbol]:  # Both have the symbol
            score += weight if symbols1[symbol] == symbols2[symbol] else -weight
        elif symbols1[symbol] or symbols2[symbol]:  # Only one has the symbol
            score -= weight  # Penalize for missing symbol
    return score

# Local alignment with affine gap penalties
def re_alignment(name1, name2):
    symbols1 = extract_symbols(name1)
```

```
    symbols2 = extract_symbols(name2)
    return weighted_score(symbols1, symbols2)

def transform_string(input_string):
    lowercased_string = input_string.lower()
    transformed_string = re.sub(r'[^a-z0-9\.]', ' ', lowercased_string)

    return transformed_string
```

Explanation of the Code

1. Weight Matrix Definition `SYMBOL_WEIGHTS`: This dictionary assigns weights to each attribute. The higher the weight, the more influence that attribute has on the final alignment score. For instance, the "title" has the highest weight, making it the most impactful on the score.
2. Extracting Symbols with Regular Expressions The function `extract_symbols` takes an input string (name) representing a musical piece and attempts to extract four attributes (title, key, instrument, opus) using regular expressions.
3. Computing the Weighted Alignment Score The function `weighted_score` computes a score based on how well the extracted symbols from two pieces match:
   - Match and Mismatch Scoring:
     - If both strings have the same attribute, the score increases by the attribute¡¯s weight.
     - If they have different values for the same attribute, the score decreases by the attribute¡¯s weight.
     - If only one of the two strings has a given attribute (indicating a missing attribute), a penalty of the attribute¡¯s weight is subtracted from the score.
   - The total score reflects the similarity of the two pieces, with higher scores indicating closer matches.

Extend NW algorithm (Gotoh Algorithm)

The Gotoh algorithm improves upon Needleman-Wunsch by introducing affine gap penalties, which have two components:

- Gap Opening Penalty (`gap_open`): The penalty for starting a new gap.
- Gap Extension Penalty (`gap_extend`): The penalty for extending an already opened gap.

Affine gap penalties allow gaps of varying lengths to be treated differently, more closely mirroring biological events where insertions or deletions (indels) of multiple bases or amino acids often occur together. The algorithm uses three matrices to keep track of alignment scores:

- `M` Matrix: Represents the main alignment, tracking matches/mismatches.

- `X` Matrix: Tracks scores for alignments ending in a gap in `str2`.
- `Y` Matrix: Tracks scores for alignments ending in a gap in `str1`.

Code

```
def my_needleman_affine(str1, str2, matrix='atiam-fpa_alpha.dist', gap_open=-5, gap_extend=-
    # Initialization
    dist_matrix = pd.read_csv(matrix, sep='\s+', index_col=0)
    n = len(str1)
    m = len(str2)

    M = np.zeros((n + 1, m + 1))
    X = np.full((n + 1, m + 1), -np.inf)
    Y = np.full((n + 1, m + 1), -np.inf)

    for i in range(1, n + 1):
        M[i, 0] = gap_open + (i - 1) * gap_extend
        Y[i, 0] = gap_open + (i - 1) * gap_extend
    for j in range(1, m + 1):
        M[0, j] = gap_open + (j - 1) * gap_extend
        X[0, j] = gap_open + (j - 1) * gap_extend

    for i in range(1, n + 1):
        for j in range(1, m + 1):
            match_score = dist_matrix[str1[i - 1]][str2[j - 1]]
            M[i, j] = max(M[i - 1, j - 1] + match_score, X[i - 1, j - 1] + match_score, Y[i
            X[i, j] = max(M[i, j - 1] + gap_open, X[i, j - 1] + gap_extend)
            Y[i, j] = max(M[i - 1, j] + gap_open, Y[i - 1, j] + gap_extend)

    score = max(M[n, m], X[n, m], Y[n, m])

    align1, align2 = "", ""
    i, j = n, m
    current_matrix = np.argmax([M[n, m], X[n, m], Y[n, m]])

    while i > 0 or j > 0:
        if current_matrix == 0:  # M matrix
            if i > 0 and j > 0 and M[i, j] == M[i - 1, j - 1] + dist_matrix[str1[i - 1]][str
                align1 = str1[i - 1] + align1
                align2 = str2[j - 1] + align2
                i -= 1
                j -= 1
            else:
                current_matrix = 1 if M[i, j] == X[i, j] else 2
        elif current_matrix == 1:  # X matrix
            align1 = "-" + align1
```

```
            align2 = str2[j - 1] + align2
            j -= 1
            if j > 0 and X[i, j] == X[i, j - 1] + gap_extend:
                current_matrix = 1
            else:
                current_matrix = 0
        elif current_matrix == 2:  # Y matrix
            align1 = str1[i - 1] + align1
            align2 = "-" + align2
            i -= 1
            if i > 0 and Y[i, j] == Y[i - 1, j] + gap_extend:
                current_matrix = 2
            else:
                current_matrix = 0

    return align1, align2, score
```

Part 4 - Musical alignments between MIDI files

Evaluation of MIDI files

```
def evaluate_midi_quality(file_path):
    print(f"Evaluating {file_path}")
    try:
        score, _ = importMIDI(file_path)
    except:
        return {"file": file_path, "quality_score": 0}

    # Pitch Range
    pitches = []
    for part in score.parts:
        for el in part.recurse(classFilter=('Note')):
            pitches.append(el.pitch.midi)
    if len(pitches) == 0:
        return {"file": file_path, "quality_score": 0}  # No notes, low quality
    pitch_range = max(pitches) - min(pitches)
    pitch_range_score = 1 if 12 <= pitch_range <= 48 else 0  # Optimal range: 1 octave to 4

    # Rhythmic Consistency
    durations = []
    for part in score.parts:
        for el in part.recurse(classFilter=('Note')):
            durations.append(el.duration.quarterLength)

    if len(durations) > 1:
        rhythm_consistency = statistics.stdev(durations)
```

```
else:
    rhythm_consistency = float('inf')
rhythm_score = 1 if rhythm_consistency < 2 else 0  # Low variance -> rhythmic consisten

# Harmonic Coherence
chords = []
for part in score.parts:
    for el in part.recurse(classFilter=('Chord')):
        chords.append(el)
harmonic_intervals = []
for c in chords:
    if len(c.pitches) > 1:
        harmonic_intervals.extend(interval.Interval(c.pitches[i], c.pitches[j]).semitone
                                  for i in range(len(c.pitches)) for j in range(i + 1, l
if len(harmonic_intervals) >= 1:
    harmonic_score = 1 if statistics.mean(harmonic_intervals) < 800 else 0
else:
    harmonic_score = 0

# Melodic Interval Consistency
melodic_intervals = []
previous_note = None
notes = []
for part in score.parts:
    for el in part.recurse(classFilter=('Note')):
        notes.append(el)
for n in notes:
    if previous_note:
        melodic_intervals.append(interval.Interval(previous_note, n).semitones)
    previous_note = n

if len(melodic_intervals) > 1:
    melodic_variance = statistics.stdev(melodic_intervals)
else:
    melodic_variance = float('inf')
melodic_score = 1 if melodic_variance < 12 else 0

# Note Density
duration_seconds = score.highestTime
note_count = len(score.flatten().notes)
note_density = note_count / duration_seconds if duration_seconds > 0 else 0
density_score = 1 if 0.5 < note_density < 10 else 0

# Total quality score
quality_score = pitch_range_score + rhythm_score + harmonic_score + melodic_score + dens
```

```
    return {
        "file": file_path,
        "pitch_range_score": pitch_range_score,
        "rhythm_score": rhythm_score,
        "harmonic_score": harmonic_score,
        "melodic_score": melodic_score,
        "density_score": density_score,
        "quality_score": quality_score
    }
```

The function `evaluate_midi_quality` takes a MIDI file path as input, attempts to import the file, and evaluates its musical quality based on specific criteria. The scoring approach is binary for each criterion (either 1 or 0), with a higher total score indicating higher quality. If any part of the evaluation fails, it assigns a score of 0, which suggests low quality.

Extend Gotoh Algorithm for MIDI files

```
def my_needleman_midi(piece1, piece2, gap_open=-5, gap_extend=-1):
    # use harmonic intervals for comparison
    intervals1 = get_harmonic_intervals(piece1)
    intervals2 = get_harmonic_intervals(piece2)

    # Initialization
    n = len(intervals1)
    m = len(intervals2)

    M = np.zeros((n + 1, m + 1))
    X = np.full((n + 1, m + 1), -np.inf)
    Y = np.full((n + 1, m + 1), -np.inf)

    for i in range(1, n + 1):
        M[i, 0] = gap_open + (i - 1) * gap_extend
        Y[i, 0] = gap_open + (i - 1) * gap_extend
    for j in range(1, m + 1):
        M[0, j] = gap_open + (j - 1) * gap_extend
        X[0, j] = gap_open + (j - 1) * gap_extend

    for i in range(1, n + 1):
        for j in range(1, m + 1):
            match_score = 5 if intervals1[i - 1] == intervals2[j - 1] else -3
            M[i, j] = max(M[i - 1, j - 1] + match_score, X[i - 1, j - 1] + match_score, Y[i
            X[i, j] = max(M[i, j - 1] + gap_open, X[i, j - 1] + gap_extend)
            Y[i, j] = max(M[i - 1, j] + gap_open, Y[i - 1, j] + gap_extend)

    score = max(M[n, m], X[n, m], Y[n, m])
```

```
    return score
```

The function `get_harmonic_intervals` is designed to extract harmonic intervals from a given musical piece by analyzing chords. And this algorithm aligns two pieces based on their harmonic intervals, assigning match scores for identical intervals and penalties for mismatches.