PA4

Env Setup

Use python 3.10.6 and poetry as dependency management

1. You may install poetry from this link
2. Run the following command under submitted directory.

```
poetry env use python
poetry install
poetry shell
```

3. You may run

```
python pa4.py
```

and see result.

- **Note: the `requirements.txt` file is auto generated by poetry through `poetry export -f requirements.txt -o requirements.txt --without-hashes`, you may try `pip install -r requirements.txt` but the environment can't be promised to be the same as mine.**

Source code logic

1. Load documents, preprocess and calculate tf-idf vector

Uses source code in previous homework to load documents, preprocess (including tokenize, lower, stemming, and stop word removal)

2. Starts hierarchical agglomerative clustering

- ***NOTE: The implementation below has used priority queue to speed up.***

```
class Queue:
    def __init__(self, q = None):
        self.queue = q or []

    def is_empty(self):
        return len(self.queue) == 0
    @property
    def front(self):
        return self.queue[0]

    def push(self, x):
        self.queue.append(x)

    def pop(self):
        item, *self.queue = self.queue
        return item

    def delete(self, x):
        self.queue.remove(x)



class PriorityQueue(Queue):
    def __init__(self, q = None):
        super().__init__(q)
        self.queue = sorted(self.queue, reverse=True)

    @property
    def key(self):
        return lambda x: -1 * x.similarity

    def push(self, x):
        pos = self._get_insert_position(self.key(x), key=self.key)
        self.queue.insert(pos, x)

    def _get_insert_position(self, x, key=None):
        q = self.queue
        l, r = 0, len(q)
        while l < r:
            mid = (l + r) // 2
            if x < key(q[mid]):
                r = mid
            else:
                l = mid + 1
        return l

    def _get_element_position(self, x):
        q = self.queue
        l, r = 0, len(self.queue)
        while l <= r:
            mid = (l + r) // 2
            if q[mid] > x:
                l = mid + 1
            elif q[mid] < x:
                r = mid - 1
            elif q[mid] == x:
                return mid
            elif q[l] == x:
                return l
            elif q[r] == x:
                return r
        raise Exception('not found')
    def delete(self, x):
        self.queue.remove(x)
```

- The data class to store index and similarity is defined as below.

```
@dataclass
class Item:
    index: int
    similarity: float

    def __gt__(self, x):
        if self.similarity > x.similarity:
            return True
        elif self.similarity == x.similarity:
            return self.index > x.index
        return False

    def __mul__(self, x):
        return Item(self.index, x * self.similarity)

    __rmul__ = __mul__
```

a. Declare variables and initialize them

```
def __init__(self, tfidfs):
        self.tfidfs = tfidfs
        doc_count = len(tfidfs)
        self.doc_count = doc_count
        self.C = []
        self.I = np.ones(doc_count)
        self.P = []
        self.A = []
        self.init()

    def init(self):
        for n, tfidf_n in enumerate(self.tfidfs):
            self.C.append([])
            for i, tfidf_i in enumerate(self.tfidfs):
                self.C[n].append(Item(i, cosine(tfidf_n, tfidf_i)))
            self.P.append(PriorityQueue(self.C[n]))
            self.P[n].delete(self.C[n][n])
```

a. Start clustering

```
def get_most_similar_documents(self):
        max_sim = -1
        document_pair = None
        for i, p in enumerate(self.P):
            if not self.I[i]:
                continue
            if p.front.similarity > max_sim:
                item = p.front
                max_sim = item.similarity
                document_pair = i, item.index
        return document_pair

    def classify(self):
        start = time.time()
        for k in range(self.doc_count - 1):
            k1, k2 = self.get_most_similar_documents()
            self.A.append((k1, k2))
            self.I[k2] = 0
            self.P[k1] = PriorityQueue()
            for i in range(len(self.I)):
                if self.I[i] == 1 and i != k1:
                    self.P[i].delete(self.C[i][k1])
                    self.P[i].delete(self.C[i][k2])
                    self.C[i][k1].similarity = min(self.C[i][k1].similarity, self.
C[i][k2].similarity)
                    self.P[i].push(self.C[i][k1])
                    self.C[k1][i].similarity = min(self.C[i][k1].similarity, self.
C[i][k2].similarity)
                    self.P[k1].push(self.C[k1][i])
        print(f"done after {time.time() - start}s")
```

b. Save cluster result

```
def save_result(self, cluster_counts):
        A = self.A
        merge_result = {i: [i] for i in range(len(A) + 1)}
        for i, (c1, c2) in enumerate(A):
            merge_result[c1].extend(merge_result[c2].copy())
            merge_result.pop(c2)
            if len(merge_result) in cluster_counts:
                with open(f"{len(merge_result)}.txt", "w") as file:
                    for cluster in merge_result.values():
                        for doc_id in sorted(cluster):
                            file.write(f"{doc_id + 1}\n")
                        file.write('\n')
```