PA2

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 pa2.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

Uses `os.listdir` to access all document files and sort them by filename through builtin `sorted` function

2. Preprocess documents

Uses `Preprocessor` written in `pa1` to preprocess the document, stopwords are `nltk.corpus.stopwords.words('english')`, stemmer is `nltk.stem.proter.ProterStemmer` and uses `[ \r\n,./\'"~!@#$%^&*()_`1234567890:;{}?\[\]+-]` as token delimiter.

3. Construct dictionary

Code logic is designed in `TFIDFVectorizer.get_terms` function, steps are as follow:

1. Init an empty dict through python's builtin data structure `collection.defaultdict`, the difference between `dict` and `defaultdict` is that `defaultdict` won't raise `KeyError` and it provides a default value for the key that doesn't exist.
2. Create a dataclass called `Term` to store information of each term.
3. Iterate every token in each document to create the dictionary.
4. Write the dictionary into `dictionary.txt`

```
@dataclass
class Term:
    index: int
    term: str
    frequency: int

def get_terms(self, filename=None) -> List[Term]:
    dictionary = defaultdict(int)
    for document in self.documents:
        for token in set(document):
            dictionary[token] += 1
    sorted_dict = sorted(dictionary.items(), key=lambda x: x[0])
    terms = [Term(index=i, term=term, frequency=frequency) for i, (term, frequency) in
enumerate(sorted_dict)]
    if filename:
        with open(DICTIONARY_FILENAME, 'w') as file:
            file.write('t_index\tterm\tdf\n')
            for term in terms:
                file.write(f'{term.index + 1}\t{term.term}\t{term.frequency}\n')
    return terms
```

4. Calculate tf-idf vectors

Code logic is designed in `TFIDFVectorizer.tfidf` property, and `get_index_dict` function is designed to let each term has its specific index, steps are as follow:

```
def get_index_dict(self) -> dict:
    """
    :return: {term_string: term_index}
    """
    terms = self.get_terms()
    return {term.term: term.index for term in terms}
```

1. Calculate tf
    a. Iterate tokens and calculate token frequency of each document, and save the frequency in `tf_vectors`, `tf_vectors[m][n]` represents the number of term n in document m.

```
@property
def TF(self):
    index_dict = self.get_index_dict()
    tf_vectors = []
    for document in self.documents:
        tf_vector = np.zeros(len(index_dict))
        for token in document:
            tf_vector[index_dict[token]] += 1
        tf_vectors.append(tf_vector)
    return tf_vectors
```

2. Calculate idf
    a. Get all terms through `get_terms` function.
    b. Create an array whose length is length of all terms.
    c. Iterate all terms and calculate each term's idf by $idf_t = log_{10}(\frac{N}{df_t})$

```
@property
def IDF(self):
    terms = self.get_terms()
    idf_vector = np.zeros(len(terms))
    for term in terms:
        idf_vector[term.index] = log(self.document_count / term.frequency, 10)
    return idf_vector
```

3. Calculate tf-idf
    a. Get `tf` and `idf` from above properties.
    b. Calculate tf-idf vectors with formula $tf\text{-}idf_{t,d} = tf_{t,d} \times idf_t$, and normalize the tf-idf vector to tf-idf unit vector.

```
@property
def TFIDF(self):
    tf_vectors = self.TF
    idf_vector = self.IDF
    tf_idf_vectors = [tf_vector * idf_vector for tf_vector in tf_vectors]
    tf_idf_vectors = [tf_idf_vector / sum(tf_idf_vector ** 2) ** 0.5 for tf_idf_vector
in tf_idf_vectors]
    return tf_idf_vectors
```

5. Save tf-idf vectors

1. Check if `output` directory exists.

2. Iterate each document's tf-idf vector, and save the term index and term's tfidf if term's tfidf is not 0.

```
if not os.path.exists(OUTPUT_DIR_NAME): # OUTPUT_DIR_NAME = 'output'
    os.mkdir(OUTPUT_DIR_NAME)

for i, tfidf_vec in enumerate(tfidf_vecs):
    with open(f"{OUTPUT_DIR_NAME}/doc{i + 1}.txt", 'w') as file:
        term_count = sum([tfidf > 0 for tfidf in tfidf_vec])
        file.write(f"{term_count}\n")
        file.write("t_index\ttf-idf\n")
        for t_index, tfidf in enumerate(tfidf_vec):
            if tfidf == 0:
                continue
            file.write(f"{t_index + 1}\t{tfidf}\n")
```

6. Define cosine similarity function

Note: The `next_line` function is used to parse next line's term index and value in the vector file, it will return next line's term index and term tf-idf value, if next line

1. Load two files' instance and use `next` function to skip first two rows.
2. Read a line from each vector file and parse it.
3. variable `num` is numerator of the cosine similarity, `denom_x` and `denom_y` represents the denominator of cosine similarity, and the initial value of them are set as `num = 0, denom_x = x_val^2, denom_y = y_val^2`
4. Start comparing x_index and y_index through the while loop
    a. `if not x_index and not y_index`: if x_index and y_index are both 0, both vector files are read thoroughly, we may stop the loop.
    b. `if x_index == y_index`, add `x_val * y_val` into variable `num`, read next line of each document, and add new `x_val ^ 2` into `denom_x`, `y_val ^2` into `denom_y`
    c. `elif not y_index or (x_index < y_index)`: if not y_index means that document y is iterated, and x_index < y_index means we have to read next line of document x. If **any of the condition** is satisfied, we have to read next line of document x and add `x_val ^2` into `denom_x`
    d. `else`: means x_index > y_index, so we have to read next line of document y and add `y_val ^ 2` into `denom_y`
5. After the loop is finished, calculate `num / (denom_x ^ 0.5 * denom_y ^ 0.5)` then we can get the cosine similarity of two documents.

```
def next_line(doc_instance):
      try:
          index, val = map(float, doc_instance.readline().split('\t'))
      except ValueError:
          index, val = 0, 0
      return index, val


def cosine_similarity(doc_x_path: str, doc_y_path: str) -> float:
    with open(doc_x_path, 'r') as doc_x, \
            open(doc_y_path, 'r') as doc_y:
        # skip first two rows
        next(doc_x), next(doc_y)
        next(doc_x), next(doc_y)

        x_index, x_val = next_line(doc_x)
        y_index, y_val = next_line(doc_y)
        num = 0
        denom_x, denom_y = x_val ** 2, y_val ** 2
        while True:
            if not x_index and not y_index:
                break
            if x_index == y_index:
                num += x_val * y_val
                x_index, x_val = next_line(doc_x)
                y_index, y_val = next_line(doc_y)
                denom_x, denom_y = denom_x + x_val ** 2, denom_y + y_val ** 2
            elif not y_index or (x_index and (x_index < y_index)):
                x_index, x_val = next_line(doc_x)
                denom_x += x_val ** 2
            else:
                y_index, y_val = next_line(doc_y)
                denom_y += y_val ** 2
        return num / (denom_x ** 0.5 * denom_y ** 0.5)
```

7. Calculate cosine similarity of doc1 and doc2

The similarity of doc1 and doc2 is `0.18826502396222747`