گزارش تمرین گروهی 1

شقایق شفیعی، عرفان آراسته، محمد هاشمی

هدف: یاده سازی یک سیستم باز یابی فضای بردار ی ساده

توضیحات کد:

```python
def preprocess_text(text):
    # Remove SGML tags
    text = re.sub(r"<[^>]+>", "", text)

    # Remove numbers and punctuation, normalize whitespace
    text = re.sub(r"\d+", "", text)  # Remove numbers
    text = re.sub(r"[^\w\s]", "", text)  # Remove punctuation
    text = re.sub(r"\s+", " ", text).strip()  # Normalize whitespace

    # Tokenize
    tokens = re.findall(r"\b\w+\b", text.lower())

    # Stem the tokens
    stemmed_tokens = [stemmer.stem(word) for word in tokens]

    # Remove stop-words after stemming
    filtered_tokens = [
        word for word in stemmed_tokens if word not in stop_words and len(word) > 2
    ]

    return filtered_tokens
```

1. **Remove SGML tags**: Removes any SGML tags (e.g. `<html>`, `<body>`) from the input text using a regular expression.
2. **Remove numbers and punctuation, normalize whitespace**: Removes numbers, punctuation, and normalizes whitespace characters (e.g. multiple spaces become a single space) using regular expressions.
3. **Tokenize**: Splits the text into individual words (tokens) using word boundaries (`\b`) and converts them to lowercase.
4. **Stem the tokens**: Applies a stemming algorithm (using the `stemmer` object) to reduce words to their base form (e.g. "running" becomes "run").
5. **Remove stop-words and short words**: Removes common stop words (e.g. "the", "and") and words with fewer than 3 characters from the stemmed tokens.

```
inverted_index = defaultdict(list)

for doc, text in processed_documents.items():
    for w in text:
        inverted_index[w].append(doc)
inverted_index["experiment"]
```

This code creates an inverted index data structure using Python's `defaultdict` class from the `collections` module. The inverted index is a mapping from words (tokens) to a list of documents that contain the word. Here's a step-by-step explanation of the code:

1. Create an empty `defaultdict` with a list as the default value. This is the inverted index data structure.
2. Iterate over the processed_documents dictionary, which contains documents as keys and lists of tokens as values.
3. For each document, iterate over the list of tokens.
4. Add the document to the list of documents associated with the token in the inverted index.
5. As an example, retrieve the list of documents associated with the word "experiment" from the inverted index.

The resulting inverted_index data structure can be used for efficient text search and retrieval. Given a query, the inverted index can be used to quickly find all documents that contain the query terms.

```
i_doc_freq = defaultdict(int)  # term -> number of docs
term_freq = {}  # term -> num of this term in each doc
N = len(processed_documents)
for term, p_list in inverted_index.items():
    i_doc_freq[term] = math.log(N / len(p_list), 10)
    for doc in p_list:
        doc_tf = term_freq.setdefault(term, [0 for _ in range(N)])
        doc_tf[int(doc) - 1] += 1
        term_freq[term] = doc_tf
term_freq["experiment"]
```

This code calculates the inverse document frequency (IDF) and term frequency (TF) for each term in a collection of documents. Here's a step-by-step explanation:

1. Create two dictionaries: `i_doc_freq` to store the inverse document frequency (IDF) for each term, and `term_freq` to store the term frequency (TF) for each term in each document.
2. Get the total number of documents (`N`) from the `processed_documents` collection.
3. Iterate over each term and its corresponding list of documents in the `inverted_index`.
4. Calculate the inverse document frequency (IDF) for the current term using the formula `log(N / len(p_list), 10)`, where `N` is the total number of documents and `len(p_list)` is the number of documents that contain the term. Store the IDF in the `i_doc_freq` dictionary.
5. Iterate over each document that contains the current term.
6. Get the term frequency (TF) list for the current term from the `term_freq` dictionary. If the term is not present, create a new list with `N` zeros.
7. Increment the TF count for the current document by 1.
8. Update the `term_freq` dictionary with the updated TF list.
9. Finally, retrieve the TF list for the term "experiment" from the `term_freq` dictionary.

```
for t, f in term_freq.items():
    term_freq[t] = [1 + math.log10(t) if t != 0 else 0 for t in f]

term_freq["experiment"]
```

This code is calculating the term frequency (TF) for each term in a document, using a specific weighting scheme. Here's a breakdown of the code:

1. `term_freq` is a dictionary where the keys are terms (words) and the values are lists of frequencies for each document.
2. The code iterates over each term `t` and its corresponding frequency list `f` in the `term_freq` dictionary.
3. For each term, the code calculates a new frequency list using a list comprehension. The new frequency is calculated as `1 + math.log10(t)` if the original frequency `t` is not zero, otherwise it's set to zero.
4. Finally, the code retrieves the weighted frequency list for the term "experiment" from the updated `term_freq` dictionary.

```
doc_tf = doc_tf / np.linalg.norm(doc_tf, ord=2, axis=0)
doc_tf, doc_tf.shape
```

This code calculates the L2-norm (Euclidean norm) of a document vector `doc_tf` and then normalizes the vector by dividing each element by the L2-norm.

```
def preprocess_query(query):
    # Preprocess the query similar to preprocess_text function
    return preprocess_text(query)
```

This code defines a function `preprocess_query` that takes a `query` as input and returns the preprocessed query. The preprocessing is done by calling another function `preprocess_text` and passing the `query` to it.

```
def calculate_cosine_similarity(doc_vector, query_vector):
    # Calculate cosine similarity between document and query vectors
    cosine_similarities = cosine_similarity(
        doc_vector.reshape(1, -1), query_vector.T
    ).flatten()
    return cosine_similarities
```

This code defines a function `calculate_cosine_similarity` that takes two input vectors, `doc_vector` and `query_vector`, and returns the cosine similarity between them.

Here's a step-by-step explanation:

1. The function takes two input vectors:
    - `doc_vector`: a vector representing a document (e.g., a TF-IDF vector)
    - `query_vector`: a vector representing a query (e.g., a TF-IDF vector)
2. The function uses the `cosine_similarity` function from a library (likely scikit-learn) to calculate the cosine similarity between the two input vectors.

The `cosine_similarity` function calculates the cosine similarity between the two input vectors, which is a measure of the cosine of the angle between them. The result is a scalar value between 0 and 1, where 1 means the vectors are identical, and 0 means they are orthogonal (i.e., perpendicular).

3. The resulting cosine similarity value is stored in the `cosine_similarities` variable.
4. The `flatten()` method is called on the `cosine_similarities` variable to ensure that the result is a single scalar value, rather than a 2D array with a single element.
5. The final cosine similarity value is returned by the `calculate_cosine_similarity` function.

```python
def process_queries(queries_file_path):
    processed_queries = {}
    with open(queries_file_path, "r") as file:
        for line in file:
            query_id, query_text = line.strip().split("\t")
            processed_queries[query_id] = preprocess_text(query_text)
    return processed_queries
```

This code defines a function `process_queries` that takes a file path `queries_file_path` as input and returns a dictionary `processed_queries` containing preprocessed query texts.

Here's a step-by-step explanation:

1. The function initializes an empty dictionary `processed_queries` that will store the preprocessed query texts.
2. The function opens the file at the specified `queries_file_path` in read mode (`"r"`).
3. The `with` statement ensures that the file is properly closed when the function is finished, regardless of whether an exception is thrown or not.
4. The function iterates over each line in the file using a `for` loop.
5. For each line, the `strip()` method is called to remove any leading or trailing whitespace characters.
6. The `split()` method is called with a separator of `\t` (tab character) to split the line into two parts:
    - `query_id`: the first part of the line, which is assumed to be a unique identifier for the query.
    - `query_text`: the second part of the line, which is the actual text of the query.
7. The `preprocess_text()` function is called with the `query_text` as input, and the resulting preprocessed text is stored as the value in the `processed_queries` dictionary with the `query_id` as the key.
8. The function returns the `processed_queries` dictionary, which now contains the preprocessed query texts, indexed by their corresponding query IDs.

```
queries_file_path = "queries.txt"
with open(queries_file_path, "r") as file:
    queries = [line.strip() for line in file]

# Preprocess queries
preprocessed_queries = [preprocess_query(query) for query in queries]
```

This code reads a file containing queries, preprocesses each query, and stores the preprocessed queries in a list.

Here's a step-by-step explanation:

1. The variable `queries_file_path` is assigned the string value `"queries.txt"`, which is the file path to the queries file.
2. The `with` statement is used to open the file at the specified `queries_file_path` in read mode (`"r"`).
3. The `with` statement ensures that the file is properly closed when the code block is exited, regardless of whether an exception is thrown or not.
4. The `queries` variable is assigned the result of a list comprehension that reads each line in the file, strips any leading or trailing whitespace characters, and stores the resulting string in a list.
5. The `preprocess_query()` function is called with each query in the `queries` list, and the resulting preprocessed query is stored in the `preprocessed_queries` list.

```
tf_idf_query = []
N = len(processed_documents)
for text in preprocessed_queries:
    tf_q = {k: 0 for k in term_freq.keys()}  # term -> num of this term in each doc
    for w in text:
        if w in term_freq:
            tf_q[w] += 1
    norm = math.sqrt(sum(t**2 for t in tf_q.values()))
    for w in tf_q:
        tf_q[w] = (
            (1 + math.log10(tf_q[w]) if tf_q[w] != 0 else 0) * i_doc_freq[w] / norm
        )
    tf_idf_query.append(np.array(list(tf_q.values())))
tf_idf_query = np.array(tf_idf_query).T
tf_idf_query, tf_idf_query.shape
```

This code calculates the TF-IDF (Term Frequency-Inverse Document Frequency) vector for each query in the `preprocessed_queries` list.

Here's a step-by-step explanation:

1. The `tf_idf_query` variable is initialized as an empty list.
2. The variable `N` is assigned the length of the `processed_documents` list.
3. The code iterates over each query in the `preprocessed_queries` list.
4. For each query, a dictionary `tf_q` is created to store the term frequency of each term in the query. The keys of the dictionary are the terms in the `term_freq` dictionary, and the initial values are all set to 0.
5. The code iterates over each term in the query, and if the term is in the `term_freq` dictionary, the corresponding value in the `tf_q` dictionary is incremented by 1.
6. The code calculates the L2 norm of the `tf_q` vector using the `math.sqrt()` function and the `sum()` function.
7. The code iterates over each term in the `tf_q` dictionary, and calculates the TF-IDF score for the term using the formula:

```python
# Rank documents for each query in reverse order based on cosine similarity
query_doc = {}
for i, query in enumerate(queries):
    sorted_indices = np.argsort(cosine_similarities[:, i])[
        ::-1
    ]  # Sort indices in descending order
    sorted_documents = [
        (list(processed_documents.keys())[idx], cosine_similarities[idx, i])
        for idx in sorted_indices
    ]
    query_doc[str(i)] = [
        list(processed_documents.keys())[idx] for idx in sorted_indices
    ]
    print(f"Query: {query}")
    for doc_id, similarity in sorted_documents[:20]:
        print(f"Document ID: {doc_id}, Similarity: {similarity}")
    print("\n")
print(query_doc)
```

This code ranks the documents for each query in reverse order based on cosine similarity.

Here's a step-by-step explanation:

1. The `query_doc` dictionary is initialized as an empty dictionary.
2. The code iterates over each query in the `queries` list.
3. For each query, the indices of the cosine similarities are sorted in descending order using the `np.argsort()` function.
4. The sorted indices are used to create a list of tuples, where each tuple contains the document ID and the corresponding cosine similarity score.
5. The sorted list of tuples is stored in the `query_doc` dictionary with the query index as the key.
6. The code prints the query and the top 20 ranked documents with their corresponding cosine similarity scores.
7. The resulting `query_doc` dictionary contains the ranked documents for each query, where the keys are the query indices and the values are lists of document IDs.

```
# load relevance.txt in a dict
relevance_file_path = "relevance.txt"
relations = {}
with open(relevance_file_path, "r") as file:
    for line in file:
        parts = line.strip().split()
        key = parts[0]
        value = parts[1]
        if key in relations:
            relations[key].append(value)
        else:
            relations[key] = [value]

print(relations)
```

This code loads a relevance score file into a dictionary called `relations`. The relevance score file is assumed to be a text file with one relevance score per line, where each line contains a document ID and a relevance score separated by a space.

Here's a step-by-step explanation:

1. The `relevance_file_path` variable is set to the file path of the relevance score file.
2. The `relations` dictionary is initialized as an empty dictionary.
3. The code opens the relevance score file using a `with` statement and a `for` loop to iterate over each line in the file.
4. For each line, the `strip()` method is used to remove any leading or trailing whitespace, and the `split()` method is used to split the line into a list of strings using a space as the delimiter.
5. The first element of the list is assigned to the `key` variable, and the second element is assigned to the `value` variable.
6. The `key` and `value` are added to the `relations` dictionary. If the `key` already exists in the dictionary, the `value` is appended to the list of values for that key. Otherwise, a new list containing the `value` is created for that key.
7. The resulting `relations` dictionary contains the relevance scores for each document, where the keys are the document IDs and the values are lists of relevance scores.

```python
def calculate_precision_recall(actual_relevant_docs, retrieved_docs, query_num):
    relevant_docs = actual_relevant_docs.get(query_num, [])
    retrieved_docs_for_query = retrieved_docs.get(query_num, [])
    precisions = []
    recalls = []

    for i in [10, 50, 100, 500]:
        true_positives = len(set(relevant_docs) & set(retrieved_docs_for_query[:i]))
        false_positives = i - true_positives
        false_negatives = len(relevant_docs) - true_positives
        precision = true_positives / i if i > 0 else 0
        recall = true_positives / len(relevant_docs) if len(relevant_docs) > 0 else 0
        precisions.append(precision)
        recalls.append(recall)
    return precisions, recalls


# Example data
actual_relevant_docs = relations
retrieved_docs = query_doc  # Retrieved documents for query 3

query_precisions = []
query_recalls = []

for query_num in retrieved_docs.keys():
    precisions, recalls = calculate_precision_recall(
        actual_relevant_docs, retrieved_docs, query_num
    )
    query_precisions.append(precisions)
    query_recalls.append(recalls)
    print(f"Query {query_num} - Precisions: {precisions}, Recalls: {recalls}")

avg_precisions = [
    sum(precisions) / len(precisions) for precisions in zip(*query_precisions)
```

This code calculates the precision and recall of a search engine for multiple queries. Here's a summary:

Function `calculate_precision_recall`

- Takes three inputs:
  - `actual_relevant_docs`: a dictionary of actual relevant documents for each query
  - `retrieved_docs`: a dictionary of retrieved documents for each query
  - `query_num`: the query number for which to calculate precision and recall
- Returns two lists: `precisions` and `recalls`
- Calculates precision and recall at different cutoffs (10, 50, 100, 500) using the following formulas:
  - Precision: `true_positives / i` (where `i` is the cutoff)
  - Recall: `true_positives / len(relevant_docs)`

- `true_positives` is the number of relevant documents in the top `i` retrieved documents
- `false_positives` is the number of non-relevant documents in the top `i` retrieved documents
- `false_negatives` is the number of relevant documents not in the top `i` retrieved documents

خروجی بخش 2:

```
Query: what investigations have been made of the wave system created by a static pressure distribution over a liquid surface .
Document ID: 0958, Similarity: 0.1647744229365691
Document ID: 0175, Similarity: 0.14822142491255424
Document ID: 0407, Similarity: 0.1453663937056803
Document ID: 1269, Similarity: 0.14380603385024116
Document ID: 1220, Similarity: 0.12777200375339134
Document ID: 0968, Similarity: 0.12493977073409215
Document ID: 0693, Similarity: 0.12247641728033412
Document ID: 1206, Similarity: 0.1179516761801831
Document ID: 1225, Similarity: 0.1155105605575951
Document ID: 1032, Similarity: 0.11373583672331081
Document ID: 0466, Similarity: 0.11214148295065608
Document ID: 0974, Similarity: 0.1110484404985977
Document ID: 0330, Similarity: 0.11057502014649223
Document ID: 0905, Similarity: 0.10272304108585953
Document ID: 1156, Similarity: 0.10250512753949455
Document ID: 0039, Similarity: 0.10248524969056638
Document ID: 1288, Similarity: 0.10086458658435375
Document ID: 0117, Similarity: 0.09905987061554127
Document ID: 1319, Similarity: 0.09749243272814782
Document ID: 0904, Similarity: 0.09379338485586747
```

همین طور که مشاهده می کنیم، برای کوئری خواسته شده بست داکتیومنت برتر اعلام شده اند.

```
Query 0 - Precisions: [0.0, 0.0, 0.0, 0.0], Recalls: [0, 0, 0, 0]
Query 1 - Precisions: [0.0, 0.0, 0.0, 0.0], Recalls: [0.0, 0.0, 0.0, 0.0]
Query 2 - Precisions: [0.0, 0.1, 0.06, 0.016], Recalls: [0.0, 0.3333333333333333, 0.4, 0.5333333333333333]
Query 3 - Precisions: [0.0, 0.0, 0.0, 0.006], Recalls: [0.0, 0.0, 0.0, 0.2]
Query 4 - Precisions: [0.0, 0.0, 0.02, 0.008], Recalls: [0.0, 0.0, 0.1111111111111111, 0.2222222222222222]
Query 5 - Precisions: [0.1, 0.06, 0.03, 0.016], Recalls: [0.05263157894736842, 0.1578947368421052, 0.15789473684210525, 0.42105263157894735]
Query 6 - Precisions: [0.0, 0.0, 0.0, 0.004], Recalls: [0.0, 0.0, 0.0, 0.1111111111111111]
Query 7 - Precisions: [0.3, 0.12, 0.06, 0.012], Recalls: [0.3333333333333333, 0.6666666666666666, 0.6666666666666666, 0.6666666666666666]
Query 8 - Precisions: [0.0, 0.0, 0.0, 0.002], Recalls: [0.0, 0.0, 0.0, 0.25]
Query 9 - Precisions: [0.0, 0.02, 0.01, 0.004], Recalls: [0.0, 0.125, 0.125, 0.25]
Average Precision: [0.04, 0.03, 0.018, 0.006800000000000005]
Average Recall: [0.03859649122807017, 0.12828947368421054, 0.1460672514619883, 0.2654385964912281]
```

مشاهده می کنیم که precision و recall برای هر کدام از داکیومنت ها محاسبه شده اند. مقدار هردو کم است و این نشان میدهد که بازیابی ما دقت پایینی دارد و یا از موارد کمی برای بررسی صحت عملکرد مدل استفاده شده است.

Low precision and recall values in information retrieval indicate that the search system is not performing well.

Low Precision: A low precision value indicates that the search system is retrieving many non-relevant documents, resulting in a high number of false positives. This means that the user has to sift through many irrelevant documents to find the relevant ones, which can be time-consuming and frustrating.

In other words, low precision means that the system is not good at filtering out irrelevant documents, resulting in a high overhead for the user to review non-relevant items.

Low Recall: A low recall value indicates that the search system is missing many relevant documents, resulting in a high number of false negatives. This means that the user may not be seeing all the relevant documents that they need, which can lead to incomplete or inaccurate results.