Project 2: Spark

Eric Grachev, Anderson Saw, Christina Lee

December 14, 2021

Abstract

When looking through large sets of data, the ability to efficiently compare and query through data is important. Scalability for large sets of data to be queried and looked through within a reasonable amount of time can be difficulty to achieve. Term-term relevance is a case where traditional methods would require too long of a run-time. The usage of term-term relevance is to find most related documents or from a search given a term. We attempt to create a program utilizing term frequency - inverse document frequency (TF-IDF) to efficiently parse through many documents to compute term-term relevance values.

1 Introduction

In order to compute TF-IDF, we require understanding of 2 other sub functions. The complete function of the TF-IDF is comprised of a term-frequency function along with an inverse document frequency. This is done to account for relevant terms that are found within documents but also with the terms that hold less weight. By creating a ratio we can more accurately identify the relevance value for each value.

1.1 Term Frequency

In order to find term frequency we need to account for all the words inside of the document, the size, along with the frequency of the given term. The value for this can be found with the formula below.

$$\mathrm{tf}(t,d) = \frac{f_{t,d}}{\sum_{t' \in d} f_{t',d}},$$

The formula is essentially the raw count of the terms in a document divided by the number of words within the document. The frequency is recorded with a summation of 0 and 1s, 1 is when the term occurred and 0 for when it is not.

1.2 Inverse document frequency

Now, there also cases where terms may not hold a lot of weight in a search. For example, "The Hen is an animal" is a term but within the term, we can easily state that there are words that are fairly common in the English language. Terms such as "The", "and", and "is" are highly common across all documents. Whereas "Hen" and "Animal" are less common meaning logically, they would hold more weight. In order to account for this we can measure an IDF.

$$\mathrm{idf}(t, D) = \log \frac{N}{|\{d \in D : t \in d\}|}$$

The formula above is used to calculate the importance of the word by looking if it is a common or none common word amongst given documents or web pages depending on what it is used in. The formula takes the log of total number of documents divided by, the number of documents that contain the term.

1.3 Term Frequency-Inverse Document Frequency

Now when we multiply the two ratios together, we get an accurate value to determine the relevance of a term across documents. The formula is shown below. The ratios are multiplied to account for the frequency in the documents along with the weight of of the term.

$$\mathrm{tfidf}(t, d, D) = \mathrm{tf}(t, d) \cdot \mathrm{idf}(t, D)$$

2 Coding

To calculate the TF-IDF and the term term relevance, we needed to use a two dimensional hash table to store each term, and for each term we store all the documents that contain the term along with the TF-IDF of each term/doc pair. We use a hash table to take advantage of its access speed at the cost of memory usage. TF-IDF is separated into two stages: TF is calculated during the partitioning stage of the code where we are reading data from the file into the hash table. IDF is calculated after the partitioning stage and immediately afterwards we take our IDF and multiply the existing TF from the hash table values with IDF to get TF-IDF. Term-term relevance is calculated for all terms based on the query we input and at our request: it is not immediately calculated to improve performance but when we query for TT we basically take the amount of same chars of the query term and each term in the hash table, from left to right, and divide it by the length of the query term times the length of each term in the hash table.

2.1 Source Code

```
from pyspark.sql import SparkSession
import math

spark = SparkSession.builder.getOrCreate()

file = spark.read.text("project2_data.txt")

doc_count = file.count()

docs_terms = {}

# How many lines at most to be processed per iteration; to prevent OOM errors
lines_per_partition = 1500

print(doc_count, "document(s) to be processed with up to", lines_per_partition, "line(s) per pa
```

```
print(math.ceil(doc_count / lines_per_partition), "partition(s) total")
print()

p_count = 0
while file.count() > 0:

    file_partition = file.limit(lines_per_partition)
    file_partition_p = file_partition.toPandas()

    # Term frequency (tf) for each term and its group of documents
    for row in file_partition_p.value:
        row_arr = row.strip().replace("  ", " ").split(" ")
        row_len = len(row_arr)
        row_docname = row_arr[0]
        row_tf_frac = 1 / (row_len - 1)
        for i in range(1, row_len):
            row_term = row_arr[i]
            if len(row_term) > 0:
                if docs_terms.get(row_term) == None:
                    docs_terms[row_term] = {}
                    docs_terms[row_term][row_docname] = row_tf_frac
                elif docs_terms[row_term].get(row_docname) == None:
                    docs_terms[row_term][row_docname] = row_tf_frac
                else:
                    docs_terms[row_term][row_docname] += row_tf_frac

    file = file.subtract(file_partition)

    p_count += 1
    print("Partition", p_count, "processed")

# tf * Inverse document frequency for each term and its group of documents
# doc_count is the size of the documents
# len(docs_terms[terms]) is the number of documents associated with each term
for terms in docs_terms:
    idf = math.log10(doc_count / len(docs_terms[terms]))
    for docnames in docs_terms[terms]:
        docs_terms[terms][docnames] *= idf

print()

while True:

    query = input("Enter a word: ")
    query_type = input("Enter 0 if querying for tf-idf; anything else if querying for term-term

    if docs_terms.get(query) == None:
        print("Not Found")
```

```
    else:
        if query_type == "0":
            for doc in docs_terms[query]:
                print(doc, docs_terms[query][doc])
        else:
            # Term-term frequency for each query / term pair, sorted in descending order
            query_len = len(query)

            term_term = []

            for terms in docs_terms:
                if terms != query:
                    terms_len = len(terms)
                    compare_size = query_len if query_len < terms_len else terms_len
                    same = 0
                    for i in range(compare_size):
                        if terms[i] == query[i]:
                            same += 1
                    term_term.append((terms, math.acos(same / (query_len * terms_len))))

            term_term.sort(key = lambda t : t[1], reverse = True)

            for terms in term_term:
                print(terms[0], terms[1])
```

2.2 Code Sample Output

Our output shows the corresponding TF-IDF value with its document id value.