Programming assignment for PDS 2

This module attempts to create a `Tf-Idf` oriented document search engine. This concept bases itself on creating word representations in relation to documents, using the frequency of those words as determinant factors in the norm of the vectors used in the representation.

Tf-Idf stands for Term Frequncy - Inverse Document Frequency. Each of these is a number, that represents some sense of a relation between the document and word in question. The term frequency can be written as

$$Tf(w, d) = \sum_{\substack{p \in d \\ w = p}} 1$$

The inverse document frequency is a little more complex, it relies on the full document data, and obtaining the information from that document. It can be written as

$$Idf(w) = \log(\frac{N}{\sum_{\substack{i \in D \\ w \in i}} 1})$$

The idea is to make a representation based on the values of the Tf-Idf rating on each word in relation to each document. We do this in this way:

$$W(d, w) = Tf(d, w) * Idf(w)$$

Where $W(d, w)$ is the value of the document d in the w dimension.

From there, the consine similarity formula is applied to the vectors of the documents and used to order the documents from a similarity base.

$$sim(d_j, q) = \cos(\theta) = \frac{\sum_{i=1}^{t} (W(d_j, P_i) \times W(q, P_i))}{\sqrt{\sum_i W(d_j, P_i)^2} \times \sqrt{\sum_i W(q, P_i)^2}}$$

Class Layout

The whole implementation was based on a single class, called `tf_idf_ranking`. This class takes the input from the user, a search query, processes it and compares it with the document database.

File input

The file input was simple, since the program was in `python`. It takes a directory and searches recursevely for all files in that directory, processing each file into an inverted index structure, based on a `python` dictionary.

For each file, it iterates through it's rows, and processes the row, removing punctuation and setting the case to lower, this way the vocabulary gets smaller and the tokenization is simpler.

The processing for removing the punctuation was done through regex, using the expression `[\W]+`, that takes all non wordlike characters and replacing those patterns with `''`, an empty character.

Indexation

The indexation of the documents and words was made taking advantage of `python` dictionary structure, which is basically a hashmap, which has an insersion and retrieval complexity of $\mathcal{O}(1)$, reducing the latent complexity of further operations.

The indexation was made in the constructor for the class, and was made with a `try-catch` structure, meaning, whenever there was not an index for the current word or document, the `catch` clause ran, creating a new index for each word or document that was not stored.

This algorithm is $\mathcal{O}(n)$, which is the minimum for this kind of operation.

The indexation was doubled, so that we could access easily each document from each word and each word frequency from each document.

Tf

The term frequency was calculated during the indexation, so that whenever a term occured in the document, it's frequency was automatically updated in the index.

The term frequency was stored in each document's index, so that you can get the Tf for any given word in any given document in $\mathcal{O}(1)$, therefore the general complexity of of the `tf` function is $\mathcal{O}(1)$.

Idf

The `idf` function is a lazy loading memoization based function, that stores the result of the function at each iteration and retrieves that value for any call of that function with the same input query, therefore amortizing the complexity to $\mathcal{O}(1)$, but for the first iteration, the function's complexity is the complexity of `python`'s log function.

This archtecture was used so that the query could take the minimum of time and operations to compute the value of all functions, meaning the program can take a bigger load of data and compute it in a manageable ammount of time.

W

Since the W function was just the multiplication of the Tf function and the Idf function, the implementation was simple, just returning the value of the function.

Cosine similarity

The implementation for the cosine similarity was just a plain and simple translation from the mathematical formulae stated previously to the corresponding code, applying the functions from `python`'s standart library set.

Page ranking

The page ranking simply applies the cosine similarity function to all documents comparing the W embedding of that document to the embedding of the query, and sorting the values obtained from all documents.

The sorting was made through `python`'s sort function, and the K-top selection was made through slice manipulation.

Testing

All the testing was unit testing through `python`'s `pytest` package. The test functions cover 100% of all codebase.

Conclusion

Although many optimizations were made in development, the program did not have a very good performance when running in a very large database. This was probably caused by the file input latency and `python`'s latent lack of performance, due to it being an interpreted language.

Usage

To run the module, first type

```
make init
```

then run

```
python -m main.py --folder [path to folder containing data] --query [query to run]
```

You can also run

```
python -m main.py --help
```

to get information on all possible commands.

To test the module, run

```
make test
```

after running

```
make init
```