Burrows Wheeler Transform Python Server

"Archimede" High School - Treviglio

Author:

Zanotti Giovanni

Abstract

It is often needed to compress data in order to efficiently send it over the network. To do this we first need a special algorithm to transform the string to compress in order for it to be ready for compression. This is where the Burrows Wheeler Transform algorithm comes into play. In our solution, we develop our own Burrows Wheeler Transform python script. The script is then used by the server solutions we developed to make the algorithm available for the services which are then going to compress the data.

Summary

1.0 Objective

All the data we elaborate through computers, in every sector, is sent as **strings**. Obviously, string means different letters in different positions, which can become complex as we need to scroll, search through them and then elaborate the found data. To make string manipulation more efficient, we can compress the data the string contains by using some special algorithms. Before using these algorithms, we have to apply a transformation, which makes it possible to bring similar characters together and make it easier to compress the data. The service we created will make it possible to elaborate and output data ready to be compressed, thanks to the Burrows Wheeler algorithm.
The objective of this project is to develop a Python service that offers the chance to calculate the Burrows Wheeler Transform after taking the string to elaborate as input.

2.0 Tools and Technologies

To develop this project, different technologies have been used.

2.1 Visual Studio Code

To develop this project, it has been decided to use the Visual Studio Code text editor, which excels for portability and ease of use, including the possibility to install extensions which increase its versatility in multi-language development.

2.2 Python

Python is a programming language, which excels for its syntax simplicity and available libraries, making it the perfect choice for data analysis, machine learning or to apply algorithms without having to focus on the language complexity itself.

2.2.1 Socket

Socket is a Python library which makes it possible for processes to communicate through the operating system and/or through the network. To actually use it, we create a socket object inside the server and give it the ip and source information, and then we make it listen for incoming requests. A client just has to create the socket and connect to the server. When the connection is established, the two machines can start communicating and exchanging messages; in our case, the client would send the string to encode, and then the server would reply with the transformed.

2.2.2 Threading

Threading is a Python library which allows you to make multithreading programs, which means having more than one independent flow of instructions.
The socket library makes it possible to realize communications, but the problem is that it would create a *sequential server*, which means that only one instruction at a time can be executed, which also translates to being able to only manage one client at a time. To solve this problem we decided to use the threading library. The solution is simple: instead of replying to only one client at a time, we create a thread for every client, which is going to independently handle the client request and reply.

2.2.3 FastAPI

FastAPI is a Python library which allows developers to develop APIs with simplicity and automatically generated documentation.
FastAPI is an excellent choice for developing this project as it allows you to easily and quickly develop APIs without losing execution speed later. At the same time, it offers the chance to access automatic documentation generated by the library and accessible to the */docs* endpoint.

2.3 Docker

Docker is an open source software which makes it possible to realize *containers*, for both Linux and Windows. Containers are "virtual boxes" where we can save the source code and all of its dependencies, making it possible to execute the program on a second machine without having to install all the necessary dependencies.
In this project, Docker has been used with FastAPI to make the API server easily runnable everywhere at any time.

4.0 Client - Server Architecture

4.1 Socket

In this solution, we find two simple implementations of the client and server architecture.

4.1.2 Server

The server script creates a socket and listens to incoming requests. When a request is made, the client handling is given to an independent thread, making it possible to manage multiple clients at once.

Here we can find the client_handler function script:

```python
# Client handler function
def client_handle(conn, addr):

    with conn:
        print(f"Connected with {addr}")

        while True:
            data = conn.recv(1024)

            if not data:
                break

            print(f"Received string: {data.decode()} from {addr}")

            encoded_string = bwt(data.decode())

            conn.send(encoded_string.encode())
```

We can find here the main steps of the client handling algorithm:

1. (The server has already received the connection request and gave it to the thread.
2. The server receives the strings to encode.
3. Through the transformer scripts, the server converts the string.
4. The server sends back the encoded string.
5. This cycle (2-4) repeats until there is no data from the client.

4.1.3 Client

The client script creates a socket and connects to the calculator server. After establishing the connection, it asks the user to insert a string to send to the server. After receiving back the encoded string, the program prints it on the terminal.

4.2 FastAPI

In this solution, we find a quite more complex implementation of the client-server architecture.

4.2.1 Server

The server script creates an api server which exposes endpoints for the different necessities.

- */calculator/{value}* : in this endpoint we send (through a GET request) the string we want to be transformed, and then we receive it from the server.
- */* (root): in this endpoint we can check if the service is up or not available.
- */health* : here we can check for the service status.

4.2.2 Client

The client script makes it possible to send a string to the */calculator/{value}* endpoint to receive back our string encoded.

```python
original = str(input("Insert string to transform:"))

if(len(original) == 0):
    print("Invalid string!")
else:
    # Elaboration endpoint URL
    url = f"http://{HOST}:{PORT}/calculator/{original}"
    # Encoding request to the server
    response = requests.get(url)
    # Get transformed from the response body
    transformed = response.json()['encoded']
    # Print results
    print(f"Original: {original}")
    print(f"Transformed: {transformed}")
```

Let's check the request script:

Here we can see the client sending the request to the calculator endpoint and then print it on the user screen.

5.0 Discussion

5.1 Encoder algorithm

To develop the encoder script, two solutions have been found for the rotations calculation algorithm:

- **Recursive**;
- **Iterative**.

We can check the recursive solution here:

```python
def rotations_calculator_recursive(line:str, rotations:list):
    ## INDUCTIVE CASE ##
    # Adding original line to the rotations list when starting #
    if(len(rotations) == 0):
        rotations.append(line)

    # calcolo shift
    line = line[1:] + line[0]

    ## BASE CASE ##
    # If the shift is the same as the first and the string is longer than 1 char, return the list #
    if(line == rotations[0] and len(rotations) > 0):
        return rotations
    ## BASE CASE END ##

    # Append shifted line to the rotations list #
    rotations.append(line)
    return rotations_calculator_recursive(line, rotations)
    ## INDUCTIVE CASE END ##
```

In this solution, every recursion calculates a new rotation, generating heavy headers and making the solution slower (in average), making the program also crash when the string is too long.

While the iterative solution doesn't:

```python
def rotations_calculator_while(line:str):
    rotations = [line]
    shifted_line = line
    if len(line) > 2:
        while rotations[0] != shifted_line or len(rotations) <= 1:
            rotations.append(shifted_line)
            shifted_line = shifted_line[1:] + shifted_line[0]
        rotations = rotations[1:]
    else:
        shifted_line = line [1] + line[0]
        rotations.append(shifted_line)

    return rotations
```

This is why the iterative solution is the one we are actually using inside our Burrows Wheeler Transform encoder.

6.0 Conclusions

In the end all the developed solutions work fully. And so we can find different execution possibilities:

- Debug;
- Terminal app;
- Normal Client - Server;
- FastAPI Client - Server.

In particular, this project made me able to fully understand how I can develop client-server applications with both socket and api communication.

7.0 Project Links

- GitHub Repository:
  https://github.com/GiZano/burrows-wheeler-calculator
- Web Presentation (for author personal portfolio use):
  https://gizano.github.io/Pages/Works/burrows-wheeler.html

8.0 Bibliography

- FastAPI documentation: https://fastapi.tiangolo.com/
- Visual Studio Code installation: https://code.visualstudio.com/
- Docker documentation: https://docs.docker.com/