LLM Part 1 : Tokeniser

Reference text :

https://www.manning.com/books/build-a-large-language-model-from-scratch

Problem Statement :

The text we will tokenize for LLM training is a short story by Edith Wharton called The Verdict, which has been released into the public domain and is thus permitted to be used for LLM training tasks. The text is available on Wikisource at https://en.wikisource.org/wiki/The_Verdict,

Step 1: Read input file

In [26]:
```
import pandas as pd

file1 = open("the-verdict.txt", "r+", encoding="utf-8")

#print("Output of Read function is ")
corpus = file1.read()
#print(text)

# check count of words
print("word count is -> ", len(corpus))
```

```
word count is ->  20479
```

Step 2: Split text to tokens

In [2]:
```
import re
```

Strategy 1 : Split on white spaces

Note

- The simple tokenization scheme below mostly works for separating the example text into individual words.
- However, some words are still connected to punctuation characters that we want to have as separate list entries.
- We also refrain from making all text lowercase because capitalization helps LLMs distinguish between proper nouns and common nouns.

In [3]:
```
text = "Hello, world. This, is a test."
result = re.split(r'(\s)', text)
print(result[0:10])
```

```
['Hello,', ' ', 'world.', ' ', 'This,', ' ', 'is', ' ', 'a', ' ']
```

Strategy 2 : Split on white space or comma or period.

Pattern Explanation

r'([,.]|\s)': This is a raw string containing the regular expression pattern used to split the text. [,.]: Matches a comma , or a period .. |: This is the OR operator in regex, meaning that the pattern will match either the part before it or the part after it. \s: Matches any whitespace character (spaces, tabs, newlines, etc.). () (parentheses): These are used to create a capturing group. Capturing groups save the matched text, so it appears in the result.

In [4]:
```
text = "Hello, world. This, is a test."

result = re.split(r'([,.]|\s)', text)

print(result[0:10])
```

```
['Hello', ',', '', ' ', 'world', '.', '', ' ', 'This', ',']
```

Strategy 3 : Split on white space or comma or period.

The tokenization scheme we devised above works well on the simple sample text. Let's modify it a bit further so that it can also handle other types of punctuation, such as question marks, quotation marks, and the double-dashes we have seen earlier in the first 100 characters of Edith Wharton's short story, along with additional special characters:

Pattern Explanation

[,.:;?_!"()\']: A character class that matches any single character inside the square brackets. This includes:

- Comma
- Period
- Colon
- Semicolon
- Question mark
- Underscore
- Exclamation mark
- Parentheses ()
- Single quote
- --: Matches the double hyphen --.
- \s: Matches any whitespace character (space, tab, newline, etc.).

Notes The parentheses around the pattern create a capture group, meaning the matched delimiters are also included in the result.

In [5]:
```
text = "Hello, world. Is this-- a test?"
result = re.split(r'([,.:;?_!"()\']|--|\s)', text)
print(result)
```

```
['Hello', ',', '', ' ', 'world', '.', '', ' ', 'Is', ' ', 'this', '--', '',
' ', 'a', ' ', 'test', '?', '']
```

Step 3: Data cleaning - remove white space characters

Note on white space character removal

- When developing a simple tokenizer, whether we should encode whitespaces as separate characters or just remove them depends on our application and its requirements.
- Removing whitespaces reduces the memory and computing requirements.
- However, keeping whitespaces can be useful if we train models that are sensitive to the exact structure of the text (for example, Python code, which is sensitive to indentation and spacing).

In [6]:
```
result = [item for item in result if item.strip()]
print(result[0:10])
```

```
['Hello', ',', 'world', '.', 'Is', 'this', '--', 'a', 'test', '?']
```

Step 4 : Create. function "text_to_tokens"

In [7]:
```
from typing import List
import re

def text_to_tokens(text: str) -> List[str]:
    """
    Create an array of tokens from a given input text data. White spaces are
    Split takes care of special characters , which are treated as tokens als

    Parameters:
    tokens (text: str ): A text string which needs to be tokenized

    Returns:
    List[str]: an list of tokens
    """

    # split text into tokens
    result = re.split(r'([,.:;?_!"()\']|--|\s)', text)

    # remove white spaces
    result = [item for item in result if item.strip()]

    return result
```

In [8]:
```
# check the function

# function called
tokenized = text_to_tokens(text)

# check length of original text
print(len(text))

# check length after tokenization and removal of whitespace characters
print(len(tokenized))

# display 1st ten tokens
print(tokenized[0:10])
```

```
31
10
['Hello', ',', 'world', '.', 'Is', 'this', '--', 'a', 'test', '?']
```

Step 5 : Build a Vocabulary.

Key Steps

- Take the tokenized text as input
- Sort Alphabetically
- Remove Duplicates
- Create a Dictionary mapping individual tokens to a unique numeric ID

For this we will define a function "create_vocab"

In [9]:
```
from typing import List, Dict

def create_vocab(tokens: List[str], ) -> List [int]:
    """
    Creates a Dictionary which maps a token to its token ID. The token input
    removed before a dictionary is mapped

    Parameters:
    tokens (tokens: List[str]): A list of tokens

    Returns:
    Dict[str, int]: a vocabulary dictionary which maps a token to a unique t

    """

    # remove duplicates
    unq_tokens = list(set(tokens))

    # sort
    srt_tokens = sorted(unq_tokens)

    # create vocabulary
    vocabulary = {token:tokenid for tokenid,token in enumerate(srt_tokens)}

    return vocabulary
```

In [10]:
```
# call function
vocab = create_vocab(tokenized)

# Check - print 1st ten items of the vocabulary
for i, item in enumerate(vocab.items()):
    print(item)
    if i > 10:
        break
```

```
(',', 0)
('--', 1)
('.', 2)
('?', 3)
('Hello', 4)
('Is', 5)
('a', 6)
('test', 7)
('this', 8)
('world', 9)
```

Step 6 : Create the Encoder

Key Steps

- Take any input text string and the pre defined vocabulary as input
- Split the text to tokens

- Use the vocabulary to generate tokenid for the input tokens
- If a token does not exists in the vocabulary encode it with -99

For this we will create a function "encode"

In [11]:
```
import re
from typing import List, Dict

def encode(text: str, vocabulary: Dict[str, int]) -> List[int]:
    """
    Encode the input text into a list of token IDs using the given vocabula

    Parameters:
    text (str): The input text string of tokens.
    vocabulary (Dict[str, int]): A dictionary mapping tokens to integer valu

    Returns:
    List[int]: A list of integers representing the token IDs.
    """

    # Split the input text into tokens
    result = re.split(r'([,.:;?_!"()\']|--|\s)', text)

    # remove white spaces
    tokens = [item for item in result if item.strip()]


    # Generate the list of token IDs using the vocabulary
    token_ids = []
    for token in tokens:
        if token.strip() and token in vocabulary:
            token_ids.append(vocabulary[token])
        else:
            # Handle unknown tokens if necessary (e.g., append a special tok
            # For example, let's append -1 for unknown tokens
            token_ids.append(-99)

    return token_ids
```

In [12]:
```
# Example usage
vocabulary = {
    "Hello": 1,
    "world": 2,
    "!": 3,
    "This": 4,
    "is": 5,
    "an": 6,
    "example": 7
}

text = "Hello world! This is an example."
encoded_text = encode(text, vocabulary)
print(encoded_text)
```

```
[1, 2, 3, 4, 5, 6, 7, -99]
```

Step 7: Create the Decoder

Notes

- When we want to convert the outputs of an LLM from numbers back into text, we also need a way to turn token IDs into text.
- For this, we can create an inverse version of the vocabulary that maps token IDs back to corresponding text tokens.
- **If an unknown token is passed for encoding - return a -99 for the same**

We develop the "decoder' function as shown below

In [13]:
```
from typing import List, Dict

def decode(vocabulary: Dict[str, int], token_ids: List[int]) -> List[str]:
    """
    Decode the input list of token IDs into a list of string tokens using th

    Parameters:
    vocabulary (Dict[str, int]): A dictionary mapping tokens to integer valu
    token_ids (List[int]): A list of integers representing the token IDs.

    Returns:
    List[str]: A list of string tokens.
    """

    # Create a reverse dictionary from the vocabulary
    int_to_str = {v: k for k, v in vocabulary.items()}

    # Generate the list of string tokens using the reverse dictionary
    tokens = []
    for token_id in token_ids:
        if token_id in int_to_str:
            tokens.append(int_to_str[token_id])
        else:
            # Handle unknown token IDs if necessary (e.g., append a special
            # For example, let's append -99 for unknown token IDs
            tokens.append(-99)

    return tokens
```

In [14]:
```
# Example usage

# define a test vocab
vocabulary = {
    "Hello": 1,
    "world": 2,
    "!": 3,
    "This": 4,
    "is": 5,
    "an": 6,
    "example": 7
}


token_ids = [1, 2, 3, 4, 5, 6, 7]
decoded_tokens = decode(vocabulary, token_ids)
print(decoded_tokens)
```

```
['Hello', 'world', '!', 'This', 'is', 'an', 'example']
```

Step 8: Build a final modified vocabulary function

- Takes a list of raw strings
- tokenizes them and removes white spaces
- sorts the list
- Adds a special token for unknown token
- Adds a special token to mark end of text of a particular text source.
- generates token id list as output

In [22]:
```
from typing import List, Dict

def create_vocab(rawtext: List[str], ) -> List [int]:
    """
    Creates a Dictionary which maps a token to its token ID.
    Takes a list of raw strings
    tokenizes them and removes white spaces
    sorts the list
    adds a special token for unknown token
    adds a special token to mark end of text of a particular text source.
    generates token id list as output


    Parameters:
    rawtext (rawtext: List[str]): A list of raw text strings

    Returns:
    Dict[str, int]: a vocabulary dictionary which maps a token to a unique t

    """

    # tokenize input text string
    tokens = re.split(r'([,.?_!"()\']|--|\s)', rawtext)

    # remove white space
    tokens = [item.strip() for item in tokens if item.strip()]

    # remove duplicates
    unq_tokens = list(set(tokens))

    # sorted tokens
    srt_tokens = sorted(unq_tokens)

    # add special tokens for unknown strings and end of text segment
    srt_tokens.extend(["<|endoftext|>", "<|unk|>"])

    # create vocabulary
    vocabulary = {token:tokenid for tokenid,token in enumerate(srt_tokens)}

    return vocabulary
```

In [27]:
```
# check text
print(text)
```

```
Hello world! This is an example.
```

In [29]:
```
# Create Vocabulary from  text
vocab = create_vocab(text)

# check length
lenvocab = len(vocab)
print(lenvocab)
```

```
# print the vocab dict
print(vocab)
```

```
10
{'!': 0, '.': 1, 'Hello': 2, 'This': 3, 'an': 4, 'example': 5, 'is': 6, 'wo
rld': 7, '<|endoftext|>': 8, '<|unk|>': 9}
```

Step 9 Build Vocabulary on short story by Edith Wharton called The Verdict

In [34]:
```
# create vocab
vocab = create_vocab(corpus)

# convert dict to list
items_list = list(vocab.items())

# Extract the first 5 items
first_5_items = items_list[:5]

# display and check
print(first_5_items)

# Extract the last 5 items
last_5_items = items_list[-5:]

# dispay and check
print(last_5_items)
```

```
[('!', 0), ('"', 1), ("'", 2), ('(', 3), (')', 4)]
[('younger', 1156), ('your', 1157), ('yourself', 1158), ('<|endoftext|>', 1
159), ('<|unk|>', 1160)]
```

Step 10 : Create the Tokenizer Class

(Code Reference - Ch 2 : Build a LLM from Scratch by Sebastian Raschka)

- Here we implement a complete tokenizer class with an encode method that splits text into tokens and carries out the string-to-integer mapping to produce token IDs via the vocabulary.
- We add an <|unk|> token to represent new and unknown words that were not part of the training data and thus not part of the existing vocabulary.
- Furthermore, we add an <|endoftext|> token that we can use to separate two unrelated text sources.
- We also implement a decode method that carries out the reverse integer-to-string mapping to convert the token IDs back into text.

Note on the decoder

We add an extra clean up step as follows

- The code **re.sub(r'\s+([,.?!"()\']', r'\1', text)**

removes any whitespace characters that appear immediately before specified punctuation marks, effectively tidying up the text by ensuring no spaces precede punctuation.

In [62]:
```
class TokenizerV1:
    def __init__(self, vocab):
        self.str_to_int = vocab
        self.int_to_str = {tokenid:string for string,tokenid in vocab.items(


    def encode(self, text):

        # split input text into tokens
        preprocessed = re.split(r'([,.?_!"()\']|--|\s)', text)

        # remove white spaces
        preprocessed = [item.strip() for item in preprocessed if item.strip(

        # add special token  unknown
        preprocessed = [
            item if item in self.str_to_int
            else "<|unk|>" for item in preprocessed
        ]

        # Return list of token ids
        ids = [self.str_to_int[s] for s in preprocessed]
        return ids


    def decode(self, ids):

        # join the decoded tokens separated by one space
        text = " ".join([self.int_to_str[i] for i in ids])

        # removes any whitespace characters that appear immediately before s
        text = re.sub(r'\s+([,.?!"()\'])', r'\1', text)
        return text
```

Test Tokenizer with basic text string

In [73]:
```
# define input text

text1 = """ If no mistake have you made, yet losing you are,  a different ga
```

In [64]:
```
# Instantiate tokenizer with vocab
tokenizer = TokenizerV1(vocab)
```

In [74]:
```
# encode and check
ids = tokenizer.encode(text1)
print(ids)
```

```
[56, 725, 1160, 538, 1155, 669, 5, 1154, 1160, 1155, 174, 5, 119, 1160, 116
0, 1155, 904, 1160, 7]
```

In [75]:
```
# decode and check
print(tokenizer.decode(ids))
```

```
If no <|unk|> have you made, yet <|unk|> you are, a <|unk|> <|unk|> you sho
uld <|unk|>.
```

Test Tokenizer with compound text string

In [76]:
```
# define inout text
text1 = "Hello, do you wish to have coffee?"

text2 = "In the shade of the large palm trees"

text = " <|endoftext|> ".join((text1, text2))

print(text)
```

```
Hello, do you wish to have coffee? <|endoftext|> In the shade of the large
palm trees
```

In [77]:
```
# Instantiate tokenizer with vocab
tokenizer = TokenizerV1(vocab)

print(tokenizer.encode(text))
```

```
[1160, 5, 362, 1155, 1135, 1042, 538, 1160, 10, 1159, 57, 1013, 898, 738, 1
013, 1160, 1160, 1160]
```

In [78]:
```
# decode
print(tokenizer.decode(tokenizer.encode(text)))
```

```
<|unk|>, do you wish to have <|unk|>? <|endoftext|> In the shade of the <|u
nk|> <|unk|> <|unk|>
```

End of notebook

In [ ]:
```

```