



# Clickbait Generator

Hardik Jindal

220420

[hardikj22@iitk.ac.in](mailto:hardikj22@iitk.ac.in)

## Overview

This model is a text generation system based on the GPT-2 (Generative Pre-trained Transformer 2) architecture. It can generate relevant text given a seed phrase. The model is trained on a large dataset and has learned to predict the next word or sequence of words based on the previous context. The Flask application is created using the Flask module. The user input is received as JSON data and parsed in the 'generate' function. The 'length' and 'inputText' values are extracted from the JSON data. The generated text is returned as a JSON response.

## Libraries Used

1. **Transformers:** A powerful library for state-of-the-art natural language processing (NLP) models, including pre-trained models like BERT and GPT.
2. **Pandas:** A versatile data manipulation and analysis library in Python, offering data structures and functions for handling structured data.
3. **Scikit-learn:** A machine learning library providing various algorithms and tools for data preprocessing, model training, and evaluation.
4. **NLTK (Natural Language Toolkit):** A library for working with human language data, offering a wide range of text processing and linguistic analysis capabilities.
5. **Random:** A module in Python's standard library used for generating random numbers, sequences, and making random choices.
6. **Flask:** A lightweight and flexible web framework for building web applications and APIs in Python.

## Training Data

The training dataset was taken from <https://www.kaggle.com/datasets/vikassingh1996/news-clickbait-dataset>. I used train1.csv file for my model. It originally had 32000 data entries. However this was taking a lot of time for computation as well as consuming a lot of RAM. so i trimmed the dataset to 800 entries.

## Training process

We first perform a number of preprocessing steps to our dataset in order to refine our dataset and make it more usable.

- Tokenization : it is the most crucial step. This function splits the input text into individual tokens by splitting on whitespace. It returns a list of tokens. These tokens can later be decoded to obtain the text.
- Lowercasing : The lowercasing function converts all tokens to lowercase. This step helps in treating words with different capitalization as the same word, reducing vocabulary size and improving generalization.
- Stopword removal : The `remove_stopwords` function removes common words that do not carry significant meaning, such as articles, pronouns, and prepositions. It utilizes the NLTK library's list of stopwords for English.
- Lemmatization : The lemmatization function applies lemmatization to reduce words to their base or dictionary form. This step helps in grouping together inflected forms of a word. The NLTK library's `WordNetLemmatizer` is used for this purpose.

The code relies on the NLTK library for stopwords and lemmatization, and it downloads the required resources using the `nltk.download` function. Each step is applied to the `train_x`, `test_x`, and `val_x` data using the `apply` function.

Next task is encoding the training data. The code uses a for loop to iterate over each text sample in the `train_x` dataset. For each text, the `tokenizer.encode_plus()` function is called to encode the text into tokens.

The `encode_plus()` function is configured with various parameters:

- `text`: The input text to be encoded.
- `add_special_tokens`: Whether to add special tokens like [CLS] (start token) and [SEP] (end token).
- `max_length`: The maximum length of the encoded sequence.
- `padding`: Padding strategy to make all sequences of the same length.
- `truncation`: Whether to truncate sequences longer than `max_length`.
- `return_tensors`: Specifies the format of the returned tensors. 'pt' indicates PyTorch tensors.

The encoded input tokens (`input_ids`) and attention mask (`attention_mask`) are then appended to the respective lists in `encoded_data_train`.

We will have three dictionaries (encoded\_data\_train, encoded\_data\_val, and encoded\_data\_test), each containing the encoded input tokens and attention masks for the respective datasets. These tensors can be directly used as input to your model for training or evaluation.

## API and integration

I made a JSON script that sets up a basic web form that sends a POST request to a specified endpoint for text generation using a language model.

The script enables users to generate text by making requests to the /generate endpoint of a server. It allows users to provide an input text prompt for context, which is then used as a basis for text generation using a language model.

I integrated the provided script into a Flask application with the help of Flask routes to handle the text generation request in the server-side code.

The major functions include defining routes, calling the text generation function and then returning the response.

```
from flask import Flask, render_template, request, jsonify

app = Flask(__name__)

@app.route('/')
def index():
    return render_template('index.html')

@app.route('/generate', methods=['POST'])
def generate():
    # Get the form data
    data = request.get_json()

    # Process the data and generate the text
    # length = data['length']
    input_string = data['inputText']
    # Your code for text generation here
    generated_text=generate_text(input_string)
    # Return the response
    response = {
        'genString': generated_text
    }
    return jsonify(response)

if __name__ == '__main__':
    app.run(debug=True)
```

Flask app

```

2 <html>
3 <head>
4 | <title>Text Generation</title>
5 </head>
6 <body>
7 | <h1>Text Generation</h1>
8 | <form id="textForm">
9 | | <!-- <label for="length">Length:</label> -->
10 | | <!-- <input type="number" id="length" name="length" required> -->
11 | | <br>
12 | | <label for="inputText">Seed Phrase:</label>
13 | | <input type="text" id="inputText" name="inputText" required>
14 | | <br>
15 | | <button type="submit">Generate Bullshit</button>
16 | </form>
17
18 <h2>Generated Bullshit:</h2>
19 <p id="outputText"></p>
20
21 <script>
22 | document.getElementById("textForm").addEventListener("submit", function(event) {
23 | | event.preventDefault();
24 |
25 | | // var length = +document.getElementById("length").value;
26 | | var inputText = document.getElementById("inputText").value;
27 |
28 | | fetch("/generate", {
29 | | | method: "POST",
30 | | | body: JSON.stringify({
31 | | | | // length: length,
32 | | | | inputText: inputText
33 | | | }),
34 | | | headers: {
35 | | | | "Content-Type": "application/json"
36 | | | }
37 | | })
38 | | .then(response => response.json())
39 | | .then(data => {
40 | | | document.getElementById("outputText").textContent = data.genString;
41 | | })
42 | | .catch(error => {
43 | | | console.error("Error:", error);
44 | | });
45 | });
46 </script>
47
48 </body>
49 </html>
50
51

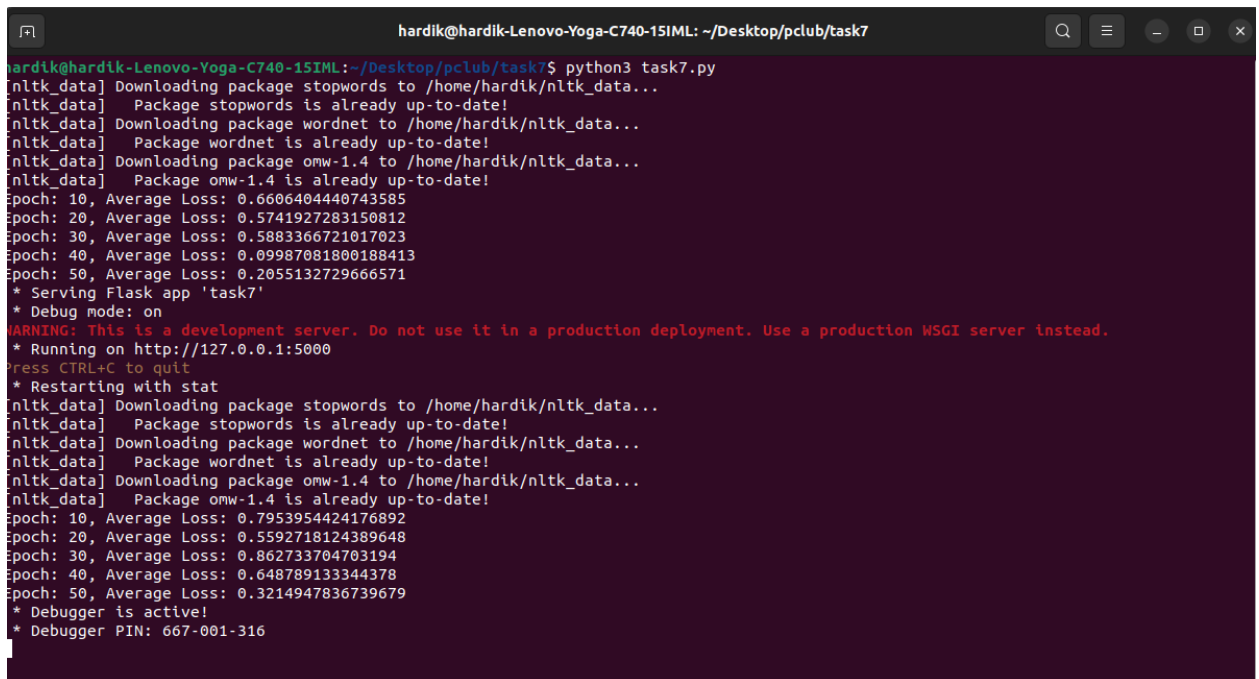
```

## JSON SCRIPT

## Usage

The model and tokenizer are loaded into the environment. The model is then set into evaluation mode. The flask app is served, and it runs on <http://127.0.0.1:5000>.

The user is prompted to enter a seed phrase. The `generate_text()` function is used to generate the required output, which is then displayed to the user on the webpage with the help of an API query.



```
hardik@hardik-Lenovo-Yoga-C740-15IML: ~/Desktop/pclub/task7
hardik@hardik-Lenovo-Yoga-C740-15IML:~/Desktop/pclub/task7$ python3 task7.py
[nltk_data] Downloading package stopwords to /home/hardik/nltk_data...
[nltk_data] Package stopwords is already up-to-date!
[nltk_data] Downloading package wordnet to /home/hardik/nltk_data...
[nltk_data] Package wordnet is already up-to-date!
[nltk_data] Downloading package omw-1.4 to /home/hardik/nltk_data...
[nltk_data] Package omw-1.4 is already up-to-date!
Epoch: 10, Average Loss: 0.6606404440743585
Epoch: 20, Average Loss: 0.5741927283150812
Epoch: 30, Average Loss: 0.5883366721017023
Epoch: 40, Average Loss: 0.09987081800188413
Epoch: 50, Average Loss: 0.2055132729666571
* Serving Flask app 'task7'
* Debug mode: on
WARNING: This is a development server. Do not use it in a production deployment. Use a production WSGI server instead.
* Running on http://127.0.0.1:5000
Press CTRL+C to quit
* Restarting with stat
[nltk_data] Downloading package stopwords to /home/hardik/nltk_data...
[nltk_data] Package stopwords is already up-to-date!
[nltk_data] Downloading package wordnet to /home/hardik/nltk_data...
[nltk_data] Package wordnet is already up-to-date!
[nltk_data] Downloading package omw-1.4 to /home/hardik/nltk_data...
[nltk_data] Package omw-1.4 is already up-to-date!
Epoch: 10, Average Loss: 0.7953954424176892
Epoch: 20, Average Loss: 0.5592718124389648
Epoch: 30, Average Loss: 0.862733704703194
Epoch: 40, Average Loss: 0.648789133344378
Epoch: 50, Average Loss: 0.3214947836739679
* Debugger is active!
* Debugger PIN: 667-001-316
```

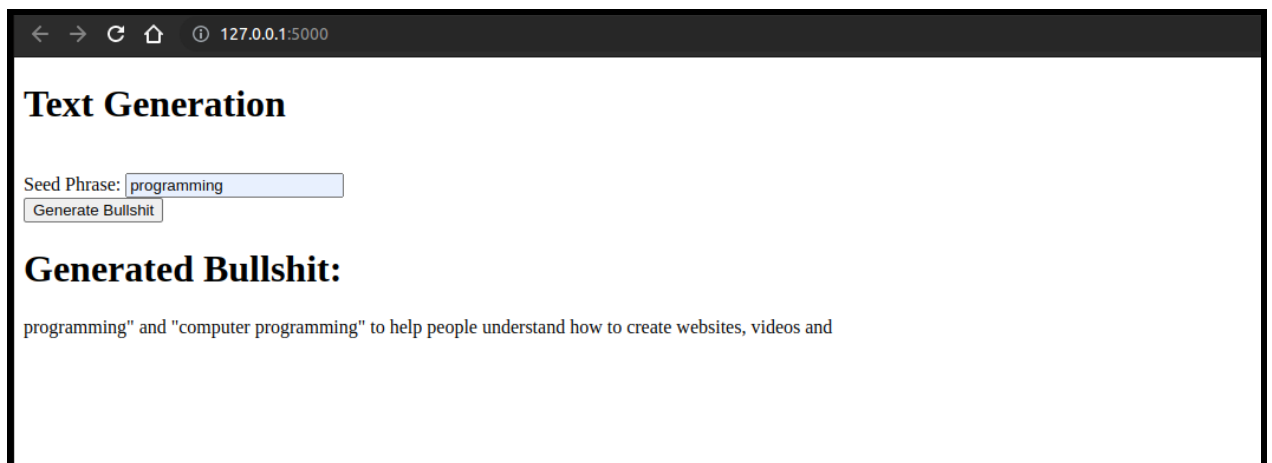
Snippet of running code on command line

## Limitations

Due to a small database, the output given is not always suitable. For instance, it may produce incorrect or nonsensical responses in certain scenarios, have biases due to the training data, or struggle with specific types of prompts.

Also the time taken to produce the output is more than optimum time. It takes about 7 seconds to run through 10 iterations.

Overall, the time required to build the app is approximately 90 seconds.



A snippet of the code in action

