

Deep Learning - 2

Section 1

Introduction:

It is a part of the Blog generation project using fine-tuned Language models. **Language models** are powerful tools in the realm of natural language processing (NLP), designed to understand and generate human-like text. Essentially, a language model is a statistical model trained to predict the probability of a sequence of words in a given language. This process involves representing words as numerical values, usually as vectors, and then employing mathematical models like neural networks to make predictions based on these representations.

In the realm of text generation projects such as blogs, language models play a pivotal role. They have the ability to generate coherent and contextually relevant text, making them invaluable for tasks like content creation, translation, and summarization.

Now for blog generation, we need to tune the language model to generate content that looks like a blog. So, let's see what is tuning a language model.

Fine-tuning:

Pre-trained language models are capable of generating text across a wide range of topics and styles, they may not always align perfectly with the specific requirements of a given task or domain. This is where fine-tuning comes into play.

Fine-tuning refers to the process of further training a pre-existing language model on a specific dataset or task. By exposing the model to domain-specific data or by fine-tuning its parameters on a specific task, we can enhance its performance and adapt it to better suit our needs. This process essentially fine-tunes the model's internal representations to better capture the nuances and patterns present in the target domain or task.

Fine-tuning is necessary in many cases because it allows us to tailor the language model's capabilities to specific contexts or applications. Whether it's generating blog posts, answering questions, or performing sentiment analysis, fine-tuning enables us to leverage the power of language models in a more targeted and effective manner.

So, as far our project Blog generation is concerned, blogs should have proper subheadings and related content. We split this project into two halves, while the 2nd half will be released on Day2. Here comes the 1st half:

Getting started:

This part's objective is to generate subheadings given a paragraph. How are we gonna do that? With the help of language models and web scraping. The first step is to create a dataset that has blog paras along with their subheadings or subtitles. It's there in wikihow. Take a look at <https://www.wikihow.com/Freeze-Pineapple>

1 Place the frozen pineapple in a smoothie or frozen drink. Just place the frozen pineapple in the blender, follow the recipe for the drink or smoothie, and enjoy. Remember to use a bit less ice, since the frozen pineapple will add some extra icy flavor to the beverage.^[6]

2 Eat the pineapple raw. Just take it out of the freezer and take a bite into the tasty, frozen fruit. Frozen fruit is delicious, and you can try the same trick with blueberries, raspberries, or any other fruit.^[7] It'll have an extra icy flavor that will make it taste even more delicious -- a little bit like ice cream.

The above images show some samples. The bold sentences are the subtitles we're looking to generate given the normal text that follows. So we need to gather sets of paras along with their corresponding titles from wikihow in order to construct a brand new dataset. For that we're gonna use **web scraping** with the help of the **Beautifulsoup** package in python.

Wonder what's web scraping? Here's your answer.

In today's digital world, the internet is a goldmine of information waiting to be tapped. But manually collecting this data from websites can be a time-consuming chore. That's where web scraping swoops in to save the day.

Web scraping is an automated method used to extract data from websites swiftly and efficiently. It's like having a digital assistant that can gather information from multiple sources in the blink of an eye. One of the key advantages of web scraping is its ability to save time and effort on manual data collection. Instead of spending hours browsing through multiple web pages and copying information manually, web scraping tools can gather data from numerous sources in a fraction of the time.

Moreover, web scraping allows us to access data that may not be easily available through traditional means. Whether it's competitor pricing information, customer reviews, or market trends, web scraping provides access to valuable insights that can inform decision-making and strategy development.

The magic of web scraping lies in its ability to convert unstructured data found on websites into a structured format that's easy to analyze. By sending requests to websites and parsing the HTML code, web scraping tools can pull out relevant information like a pro. Don't worry, all these steps will be detailed below.

But web scraping isn't just about text. It can also snag images, videos, tables, and lists, making it incredibly versatile. This flexibility means you can extract a wide range of data types to suit your needs.

Web scraping with BeautifulSoup:

So, there are many available tools and libraries that can help streamline the process and make web scraping more efficient and effective. Let's get into 1 such package called **Beautifulsoup**.

Imagine you stumble upon a webpage brimming with valuable information, but it's buried within layers of HTML tags. Trying to make sense of it all can feel like navigating a maze blindfolded. This is where Beautiful Soup comes into play. It acts as your trusty guide, helping you navigate through the HTML jungle with ease. At its core, Beautiful Soup provides a set of intuitive functions that allow you to interact with HTML just like you would with a web page using developer tools. Whether you're scraping data for research, analysis, or building web applications, Beautiful Soup streamlines the process by abstracting away the complexities of HTML parsing.

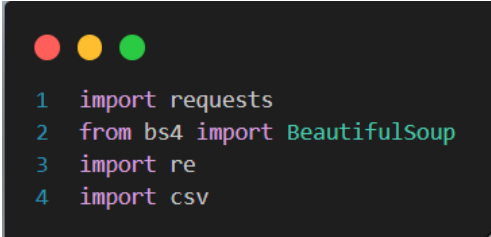
With BeautifulSoup, you can swiftly extract specific elements from HTML, such as paragraphs, headings, links, or tables, without breaking a sweat. Its syntax is clean and straightforward, making it accessible even to those new to web scraping and data parsing. One of the key strengths of BeautifulSoup lies in its ability to handle poorly formatted HTML gracefully. Even if a webpage's HTML is a tangled mess, BeautifulSoup can often still extract the data you need, saving you from headaches and frustration.

In this essence of blog generation, we're gonna be using BeautifulSoup as a web scraper to scrape data from wikihow.

Lets get your hands dirty with some coding.

- Create a kaggle account, if you haven't already.
- Create a new kaggle notebook and name it as "Web scraping"
- Start a new session in that notebook using the power button at top right corner.
- From now on, create a new cell for each code snippet and run it (shift + enter) to see the output of the current cell.

The 1st step is to import necessary packages.



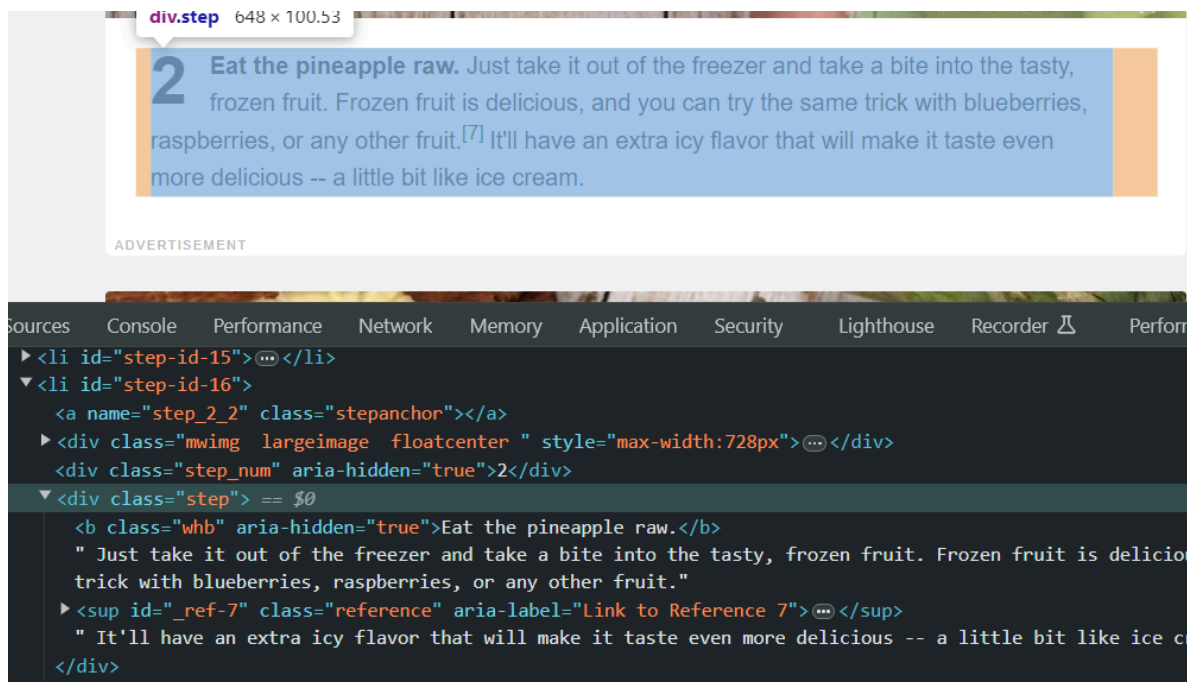
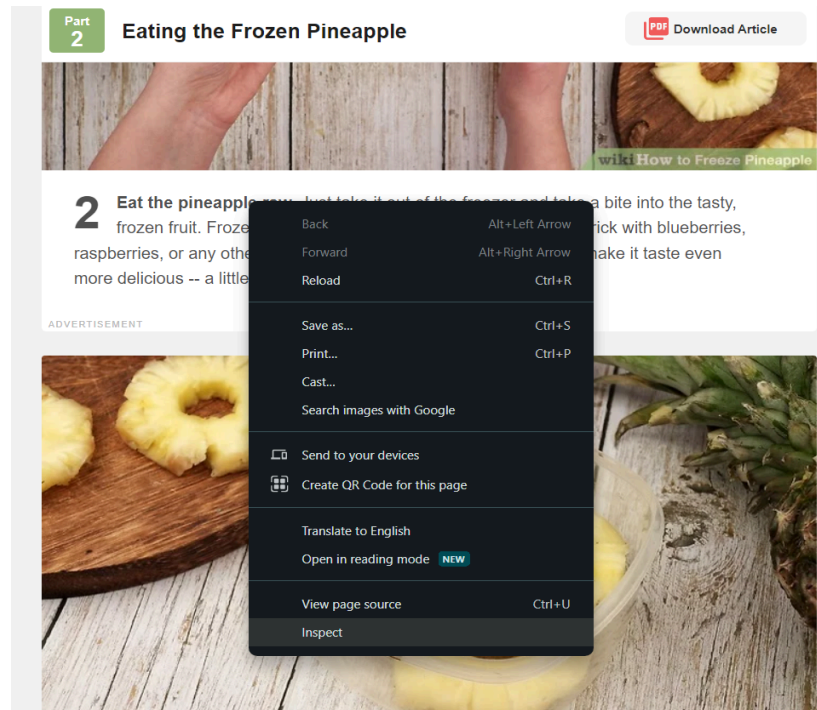
```
1 import requests
2 from bs4 import BeautifulSoup
3 import re
4 import csv
```

- `import requests`: Importing `requests`, a popular HTTP library in Python, simplifies the process of making HTTP requests to web servers. In the context of web scraping, `requests` is often used to fetch the HTML content of web pages that will be parsed.
- `import bs4`: Importing `bs4`, a short-format for Beautiful Soup 4. With Beautiful Soup, you can extract specific elements or data from HTML documents with ease, making it an essential tool for web scraping tasks.
- `import re`: Imports `re`, that stands for regular expressions. Regular expressions are powerful tools for pattern matching and text manipulation. This will be used in situations where there is a need for complex text processing or pattern matching.
- `import csv`: Imports `csv`, which provides functions for reading and writing CSV files. This is used to save extracted (web scraped data) data into CSV files for further analysis or storage.

Now how do we get blog posts from wiki? Using this link:
<https://www.wikihow.com/Special:Randomizer>

Copy paste this link in your browser several times. You will notice that everytime it provides a blog post at random. Let's say we get a blog post using that link, how do we get the content of the webpage in python code? Using `beautifulsoup` package!

First let's analyze where the content resides in the webpage. Remember the pineapple blog we visited earlier? Go to that page and right click on any of the paras or subheadings. Then move on to inspect the page like this:



As we can see, there lies the content that we need. Right inside a div with classname as "step" and inside the bold tag, lies the subtitle of the paragraph that we're seeing right now. We just have to know how to extract that content using

web scraping. Remember that a single blog post will have multiple such divs with classname as “step” and we need to extract all those paragraphs and their corresponding titles.

```
1 # URL of the Wikihow page to scrape
2 url = 'https://www.wikihow.com/Special:Randomizer'
3
4 # Send an HTTP request to the URL and receive the HTML content
5 response = requests.get(url)
6 html_content = response.content
7
8 # Parse the HTML content using BeautifulSoup
9 soup = BeautifulSoup(html_content, 'html.parser')
10 article_title = soup.find('title').text.strip()
11 print(article_title)
```

In the above code, we are first making a HTTP request to the URL in order to receive a random blog post from wikihow. Then using beautifulsoup package, we are parsing the received html. After that, we are printing the content inside the title tag of the received html in the last two lines. Run this code in a cell several times to see different titles of the blog posts. Sample output:

```
article_title = soup.find('title').text.strip()
print(article_title)
```

```
How to Play the Ancient Game of Pai Sho (with Pictures) - wikiHow
```

Next we’re extracting the subheadings and paragraphs using the following code. Take time to understand it before running in kaggle:


```

1 # Extract the subheadings and paragraphs using the appropriate HTML tags
2 subheadings = []
3 paragraphs = []
4 steps = soup.find_all('div', {'class': 'step'})
5 for step in steps:
6     subheading_element = step.find('b')
7     if(subheading_element is not None):
8         subheading_text = subheading_element.text.strip().replace('\n', '')
9         subheading_text = subheading_text.encode('ascii', errors='ignore').decode()
10        subheading_text = re.sub(r'', '', subheading_text)
11        print(subheading_text)
12        subheadings.append(subheading_text)
13
14        # this block removes titles and extra links
15        subheading_element.extract()
16        for span_tag in step.find_all('span'):
17            span_tag.extract()
18
19        paragraph_text = step.text.strip().replace('\n', '').replace(' ', ' ')
20        paragraph_text = paragraph_text.encode('ascii', errors='ignore').decode()
21        paragraph_text = re.sub(r'', '', paragraph_text)
22        print(paragraph_text)
23        paragraphs.append(paragraph_text)

```

Firstly we find all div elements with classname “step”. Then for each such element, we’re extracting subheadings and paragraphs. As said before, subheading is present inside the element b (bold tag in HTML). After grabbing the subheading_element, we’re doing basic processing like replacing newline characters, ASCII encoding and removing extra spaces.

Then we’re using extract() function to remove all the content inside the “step” div tag except the actual para content. We’re removing the heading element and other span tags. Finally the content present in step.text is the actual paragraph content that we’re concerned about. Hence text processing is done similar to what we did for subheading. Run this in a kaggle cell to see the extracted headings and paras.

Sample output:

Recognize a mild case of Pseudomonas.

Pseudomonas usually produce mild symptoms in healthy people with strong immune systems. These infections may be water-borne. There have been reports of:[2]Eye infections in people who use extended-wear contact lenses. To avoid this, change your contact lens solution instead of topping it up. Do not wear your contacts for longer than recommended by your doctor or the manufacturer's instructions.Ear infections in children after swimming in contaminated water. This can occur if the pool does not have enough chlorine to adequately disinfect it.Skin rashes after using a contaminated hot tub. This rash generally manifests as itchy red bumps or blistered filled with fluid around the hair follicles. It may be worse in areas where your skin was covered by a bathing suit.[3]

Know symptoms of different pseudomonas infections.

Signs and symptoms of pseudomonas depend on where the infection occurs.[4]Blood infections are characterized by fever, chills, fatigue, muscle and joint pains, and are extremely serious.Lung infections (pneumonia) include symptoms like chills, fever, a productive cough, difficulty breathing.Skin infections may cause an itchy rash, bleeding ulcers, and/or headache.Ear infections may present with swelling, ear pain, itching inside the ear, discharge from the ear, and difficulty hearing.Eye infections caused by pseudomonas may include the following symptoms: inflammation, pus, swelling, redness, pain in the eye, and impaired vision.

Go to the doctor for a diagnosis.

The doctor will likely want to look at the rash and may take a sample of the bacteria to send to the lab to confirm the diagnosis. This may be done in two ways:[5]Swabbing the infection on your skinTaking a biopsy. Doing a biopsy is rare.

Now everything left is to write the headings and paragraphs present in the list into a csv file. We'll write the article's title, subheading and the paragraph itself (so totally 3 columns).

```

1 if(len(subheadings)):
2     with open('/kaggle/working/wikiHow.csv', mode='a', newline='',encoding='utf-8') as csv_file:
3         writer = csv.writer(csv_file)
4         for i in range(len(subheadings)):
5             writer.writerow([article_title,subheadings[i], paragraphs[i]])

```

But wait, one last thing is there to do. All we did is for one random blog post. Now how to do it for many, say 4000 posts. You guessed it, just run this for 'n' times inside a loop. **Here is the complete code:** (RUN IT in kaggle and it will take around 45 mins to complete: the resultant csv file should have at least 50000 rows)

```

1  for count in range(4000):
2      # URL of the WikiHow page to scrape
3      url = 'https://www.wikihow.com/Special:Randomizer'
4
5      # Send an HTTP request to the URL and receive the HTML content
6      response = requests.get(url)
7      html_content = response.content
8
9      # Parse the HTML content using BeautifulSoup
10     soup = BeautifulSoup(html_content, 'html.parser')
11     article_title = soup.find('title').text.strip()
12     print(article_title+" "+str(count))
13
14     # Extract the subheadings and paragraphs using the appropriate HTML tags
15     subheadings = []
16     paragraphs = []
17     steps = soup.find_all('div', {'class': 'step'})
18     for step in steps:
19         subheading_element = step.find('b')
20         if(subheading_element is not None):
21             subheading_text = subheading_element.text.strip().replace('\n','')
22             subheading_text = subheading_text.encode('ascii', errors='ignore').decode()
23             subheading_text = re.sub(r'', '', subheading_text)
24             subheadings.append(subheading_text)
25             subheading_element.extract()
26             for span_tag in step.find_all('span'):
27                 span_tag.extract()
28             paragraph_text = step.text.strip().replace('\n','').replace(' ', ' ')
29             paragraph_text = paragraph_text.encode('ascii', errors='ignore').decode()
30             paragraph_text = re.sub(r'', '', paragraph_text)
31             paragraphs.append(paragraph_text)
32
33     if(len(subheadings)):
34         with open('/kaggle/working/wikiHow.csv', mode='a', newline='', encoding='utf-8') as csv_file:
35             writer = csv.writer(csv_file)
36             for i in range(len(subheadings)):
37                 writer.writerow([article_title, subheadings[i], paragraphs[i]])

```

After its completion, you'll see a csv file in kaggle's working directory. Save the wikiHow.csv file locally.

Once downloaded locally, open the csv file and add the first row as "title, heading, paragraph" to make it as the column titles. Something like this:

```
wikiHow.csv X
wikiHow.csv
1 title,heading,paragraph
2 How to Hogtie Someone: Step-by-Step Guide for Safe Play,Lay yo
3 How to Hogtie Someone: Step-by-Step Guide for Safe Play,Lift t
4 How to Hogtie Someone: Step-by-Step Guide for Safe Play,Fold a
5 How to Hogtie Someone: Step-by-Step Guide for Safe Play,Wrap t
6 How to Hogtie Someone: Step-by-Step Guide for Safe Play,Cross
7 How to Hogtie Someone: Step-by-Step Guide for Safe Play,Wrap t
8 How to Hogtie Someone: Step-by-Step Guide for Safe Play,Use th
9 How to Hogtie Someone: Step-by-Step Guide for Safe Play,Have y
10 How to Hogtie Someone: Step-by-Step Guide for Safe Play,Bend t
11 How to Hogtie Someone: Step-by-Step Guide for Safe Play,Cross
```

Submission Link : <https://forms.gle/Pja5PrMiuMAdEn8P7>

Section 2

Next step:

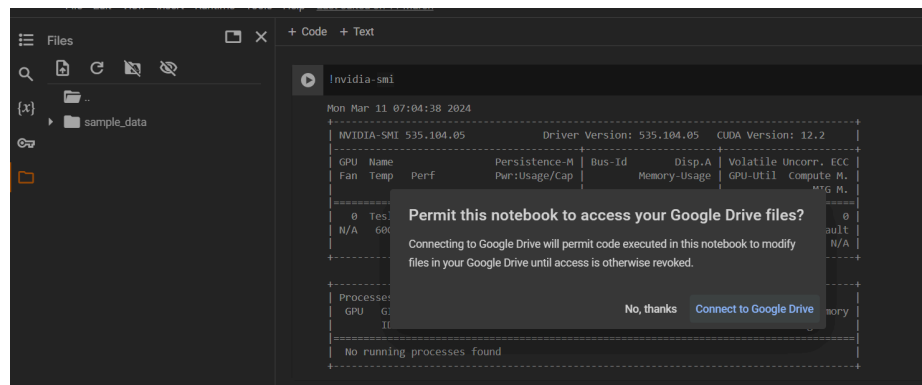
Now with the help of scrapped data, we have to finetune a language model such that it should generate a suitable heading given a paragraph. Let's start and this time we're going to run code in google colab.

Data pre-processing in Colab:

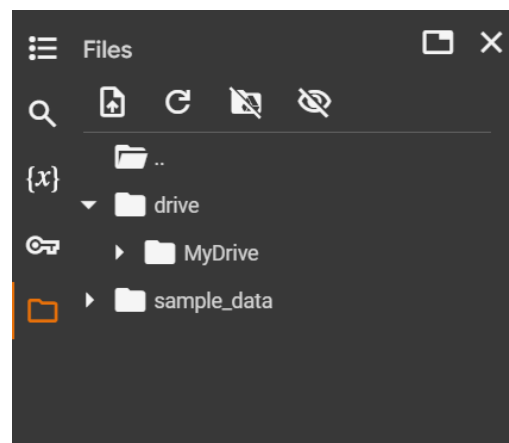
Create a new notebook in google colab and click on the drop down near the "connect" button to change the runtime type as T4 GPU for accelerator and click save.

Now click on the connect button to connect to a colab runtime. Since we need to use the scrapped data (csv) file, we'll make use of google drive. In the google drive of the same google account that you logged in into colab; create a folder named "code cycle" and upload the wikihow.csv file there.

Since we need to access this csv file inside colab, we've to mount this drive in the notebook. Go back to the connected runtime of the notebook. On the left hand side, the last option in the navbar will be "Files". And in the Files section, the third option will be a button indicating "Mount drive". Click on it and give permission for colab to access your drive files.



On successful mounting of google drive, you'll be able to see drive folder in the files section in addition to sample_data folder as shown:



Get your hands into coding now! To start with, we have to install necessary packages. We'll also recheck if we have access to the GPU as we connected to the T4 GPU earlier. Run individual cell code as shown below:

```
[1] !nvidia-smi

Sun Mar 31 11:36:56 2024

+-----+
| NVIDIA-SMI 535.104.05                Driver Version: 535.104.05   CUDA Version: 12.2   |
+-----+-----+-----+
| GPU  Name          Persistence-M   Bus-Id        Disp.A   Volatile Uncorr. ECC |
| Fan  Temp   Perf          Pwr:Usage/Cap     Memory-Usage   GPU-Util  Compute M. |
|=====+=====+=====+
| 0   Tesla T4               Off      00000000:00:04:0 Off      0
| N/A   40C    P8              9W / 70W           0MiB / 15360MiB      0%      Default |
|=====+=====+=====+

+-----+
| Processes:                               GPU Memory |
|  GPU   GI    CI        PID   Type   Process name                  Usage |
|=====+=====+=====+
| No running processes found              |
+-----+

[2] !pip install -U datasets

[3] %%capture
    !pip install transformers==4.19.2
    !pip install rouge_score
```

We'll next use the pandas library and import the csv file we uploaded in our drive. CSV file will be loaded into a pandas dataframe (learn more about [dataframe](#)). Note that the file path should match the path of the file you uploaded.

So just navigate through the drive folder in the files section of colab, find the csv file and copy the path with the help of 3 dots on the right side. Once loaded, we'll see the 1st 5 rows with the help of head function:

```
from datasets import load_metric
import pandas as pd
df = pd.read_csv("/content/drive/MyDrive/Code cycle/wikiHow.csv")
df.head()
```

	title	heading	paragraph
0	How to Clean a Futon Mattress: Washing & Stain...	Vacuum your futon.	To start, give your futon a good vacuuming to ...
1	How to Clean a Futon Mattress: Washing & Stain...	Deodorize your futon.	Futons start to smell over time, so make a pol...
2	How to Clean a Futon Mattress: Washing & Stain...	Get rid of any spots.	If you notice any spots on the futon when clea...
3	How to Clean a Futon Mattress: Washing & Stain...	Clean and examine the frame.	While you're cleaning the futon, you should al...
4	How to Clean a Futon Mattress: Washing & Stain...	Blot out as much of the stain as you can.	When a stain is not lifted through conventiona...

Even though we scraped the data ourselves, there might be some duplicates, null values present in the dataset which should be dealt with before training. This is called **data pre-processing** step. This could be understood with the help of following two cells where we're dealing with NA values and duplicate entries. We can clearly see that out of 70K rows, around 15K entries are duplicate entries because we made use of randomizer URL which can return the same blog post some times.

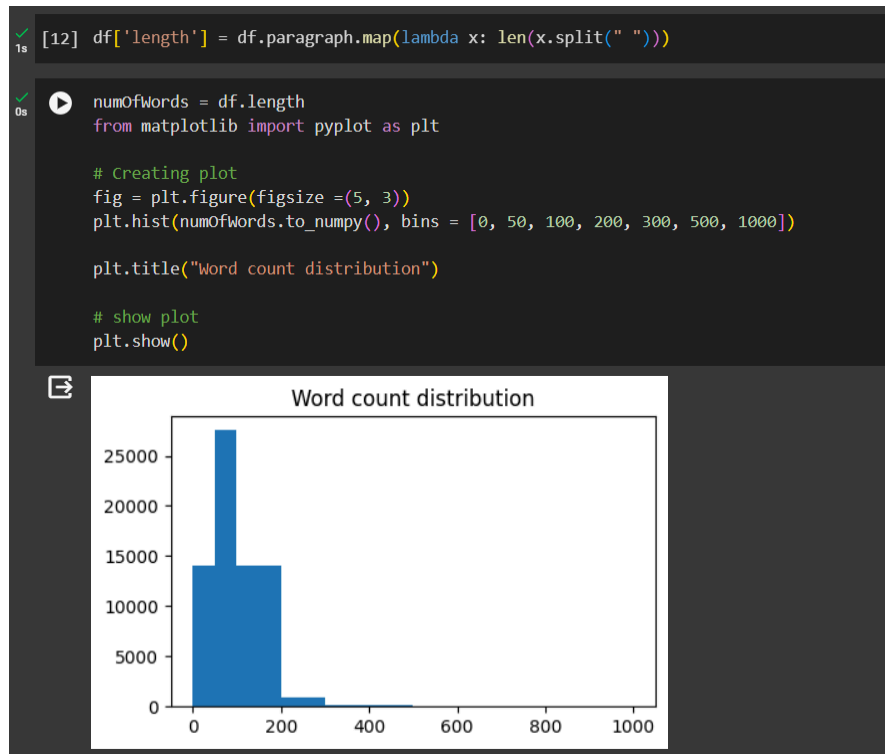
```
[10] print(df.shape)
      df = df.dropna()
      print(df.shape)

(72038, 3)
(70988, 3)

[11] print(df.shape)
      df = df.drop_duplicates()
      print(df.shape)

(70988, 3)
(56655, 3)
```

Next we'll deal with one of the common issues with dataset i.e., **outliers**. In this dataset outliers are those entries where the length of paragraph is either too low or too high. This could be visualized with the help of matplotlib library. But before that we need to add an extra column which will have the number of words in each paragraph. As shown in the histogram, it is clear that there are some very few paragraphs which have upto 1000 words.



We shall assume that those entries which have words more than 200 are outliers. After removing them, the dataframe size becomes 55K which means that we removed around 1000 outliers.

```
[17] tempDf = df[df.length <= 200]
      tempDf.shape






(55643, 4)
```

Fine-tuning with LED:

We'll be using a language model: **Longformer Encoder Decoder** (LED) and finetune it to generate heading for a paragraph (our usecase).

So the 1st step is to import a model's version:


```
[ ] from transformers import AutoTokenizer
tokenizer = AutoTokenizer.from_pretrained("allenai/led-base-16384")
```

Downloading: 100%		27.0/27.0 [00:00<00:00, 1.78kB/s]
Downloading: 100%		1.07k/1.07k [00:00<00:00, 71.2kB/s]
Downloading: 100%		878k/878k [00:00<00:00, 13.0MB/s]
Downloading: 100%		446k/446k [00:00<00:00, 6.59MB/s]
Downloading: 100%		772/772 [00:00<00:00, 53.2kB/s]

Wonder what's tokenization is and why it's needed?

Imagine a bridge connecting human language and machine understanding; tokenization is the very construction of this bridge. Tokenization emerges as a fundamental process akin to building a bridge between the two worlds. Another example to make you clear, imagine a vast sea of text, and tokenization acts as the compass, breaking down this sea into navigable chunks—tokens—that machines can comprehend.

But what exactly is tokenization, and why is it indispensable for language models?

Tokenization is the art of segmenting text into smaller units called tokens. These tokens could be words, subwords, or even characters—each representing a fundamental building block of language for machines. By breaking down text into tokens, we pave the way for machines to process and understand human language more effectively.

But why do we need tokenization?

Consider this: language models operate on numerical data, not raw text. To bridge the gap between text and numerical data, tokenization steps in. It encodes words, phrases, and characters into numerical representations, creating a language that algorithms can interpret.

Tokenization serves as the crucial first step in preparing data for language models. It transforms text into a format that these models can digest, enabling them to analyze, generate, or understand human language with precision and accuracy.

Moreover, the choice of tokenizer holds immense significance. Different language models may require specific tokenization techniques to ensure optimal performance. Using the right tokenizer ensures that the text is segmented in a manner that aligns with the model's understanding, enhancing its ability to comprehend and generate language effectively.

So, next when it comes to training the model, you might have heard the word batches. So How does Batch Size impact your model learning?

Firstly, **batch size** impacts the efficiency of the training process. By feeding the model batches of data instead of the entire dataset at once, training becomes more manageable and computationally efficient. This allows the model to update its parameters more frequently, leading to faster convergence and potentially shorter training times.

Moreover, batch size influences the generalization ability of the model. Larger batch sizes often lead to faster convergence but may result in overfitting, where the model memorizes the training data instead of learning meaningful patterns. On the other hand, smaller batch sizes promote better generalization by exposing the model to more diverse examples and preventing it from memorizing the training data.

Additionally, batch size impacts the utilization of computational resources. Larger batch sizes require more memory and computational power, which may pose challenges, especially when working with limited resources or large datasets. In contrast, smaller batch sizes consume fewer resources but may lead to slower convergence.

Finding the optimal batch size is a balancing act. It involves experimenting with different batch sizes and monitoring the model's performance on validation data to strike the right balance between efficiency, generalization, and resource utilization.

```
1 max_input_length = 1024
2 max_output_length = 64
3 batch_size = 16
4
5 def process_data_to_model_inputs(batch):
6     # tokenize the inputs and labels
7     inputs = tokenizer(
8         batch["paragraph"],
9         padding="max_length",
10        truncation=True,
11        max_length=max_input_length,
12    )
13    outputs = tokenizer(
14        batch["heading"],
15        padding="max_length",
16        truncation=True,
17        max_length=max_output_length,
18    )
19
20    batch["input_ids"] = inputs.input_ids
21    batch["attention_mask"] = inputs.attention_mask
22
23    # create 0 global_attention_mask lists
24    batch["global_attention_mask"] = len(batch["input_ids"]) * [
25        0 for _ in range(len(batch["input_ids"])[0])]
26    ]
27
28    # since above lists are references, the following line changes the 0 index for all samples
29    batch["global_attention_mask"][0][0] = 1
30    batch["labels"] = outputs.input_ids
31
32    # We have to make sure that the PAD token is ignored
33    batch["labels"] = [
34        [-100 if token == tokenizer.pad_token_id else token for token in labels]
35        for labels in batch["labels"]
36    ]
37
38    return batch
```

Whenever it comes to training a model, data splitting plays an important role. But why is it needed? Get through the below lines.

The **train-test-validation** split stands as a crucial compass, guiding practitioners on the journey to developing robust and reliable models. This fundamental process involves dividing a dataset into three distinct subsets—training, testing, and validation—each playing a unique role in the model development pipeline.

Imagine you're building a machine learning model tasked with predicting housing prices based on various features like location, size, and amenities. You begin by feeding the model with a dataset containing historical housing data, eager to see how well it performs. This is where the train-test-validation split comes into play.

The training set serves as the foundation upon which your model learns to make predictions. By exposing the model to a subset of the data containing known outcomes, it gradually hones its predictive abilities, adjusting its parameters to minimize errors and improve accuracy.

But how do you ensure that your model isn't just memorizing the training data, performing well only on familiar examples? Enter the validation set. This subset acts as a litmus test, allowing you to fine-tune your model's parameters while gauging its performance on unseen data. By regularly evaluating the model on the validation set, you can prevent overfitting—where the model becomes too attuned to the training data and fails to generalize to new instances.

Finally, the testing set serves as the ultimate reality check. This subset contains data that the model has never encountered during training or validation. By assessing the model's performance on this unseen data, you gain valuable insights into its real-world applicability and generalization capabilities. A well-performing model should exhibit robustness across all three subsets—training, validation, and testing—signifying its ability to make accurate predictions on new data.

So, as same for our blog generation project also, we'll be splitting the dataset (pre-processed dataframe) into train, test and validation sets. Validation set is reduced to a small number just for the sake of this project in order to speed up the training process. In reality, it shouldn't be reduced. Also we'll be finetuning the model only for an hour (almost a single epoch). In reality, it should be trained for more epochs.

```
import numpy as np
train, validate, test = np.split(tempDf.sample(frac=1, random_state=42), [int(.6*len(df)), int(.7*len(df))])
print(train.shape)
print(validate.shape)
print(test.shape)

(33993, 4)
(5665, 4)
(15985, 4)

[ ] validate = validate[:20]

[ ] validate.shape

(20, 4)
```

```
[ ] from datasets import Dataset
train_dataset = Dataset.from_pandas(train)
val_dataset = Dataset.from_pandas(validate)

[ ] train_dataset = train_dataset.map(
    process_data_to_model_inputs,
    batched=True,
    batch_size=batch_size,
    remove_columns=["title", "heading", "paragraph", "length", "__index_level_0__"],
)

Map: 100% ██████████ 33993/33993 [00:37<00:00, 1053.54 examples/s]

[ ] val_dataset = val_dataset.map(
    process_data_to_model_inputs,
    batched=True,
    batch_size=batch_size,
    remove_columns=["title", "heading", "paragraph", "length", "__index_level_0__"],
)

Map: 100% ██████████ 20/20 [00:00<00:00, 358.52 examples/s]

train_dataset.set_format(
    type="torch",
    columns=["input_ids", "attention_mask", "global_attention_mask", "labels"],
)
val_dataset.set_format(
    type="torch",
    columns=["input_ids", "attention_mask", "global_attention_mask", "labels"],
)
```

We won't be giving the inputs as the same dataframe columns. As you can see in the above snippet, we are using `process_data_to_model_inputs` in order to feed the input to the model in a way that it can understand. That is by using tokenizer for the chosen language model.

FINALLY, it's time to train the model with our data. So the final step is to set configurations and run it for some time. We'll be using [rouge](#) metric for measuring performance.

```
1 from transformers import AutoModelForSeq2SeqLM
2 led = AutoModelForSeq2SeqLM.from_pretrained("allenai/led-base-16384", gradient_checkpointing=True, use_cache=False)
3 led.config.num_beams = 2
4 led.config.max_length = 64
5 led.config.min_length = 2
6 led.config.length_penalty = 2.0
7 led.config.early_stopping = True
8 led.config.no_repeat_ngram_size = 3
9 rouge = load_metric("rouge")
10
11 def compute_metrics(pred):
12     labels_ids = pred.label_ids
13     pred_ids = pred.predictions
14
15     pred_str = tokenizer.batch_decode(pred_ids, skip_special_tokens=True)
16     labels_ids[labels_ids == -100] = tokenizer.pad_token_id
17     label_str = tokenizer.batch_decode(labels_ids, skip_special_tokens=True)
18
19     rouge_output = rouge.compute(
20         predictions=pred_str, references=label_str, rouge_types=["rouge2"]
21     )["rouge2"].mid
22
23     return {
24         "rouge2_precision": round(rouge_output.precision, 4),
25         "rouge2_recall": round(rouge_output.recall, 4),
26         "rouge2_fmeasure": round(rouge_output.fmeasure, 4),
27     }
28
29 from transformers import Seq2SeqTrainer, Seq2SeqTrainingArguments
30 import transformers
31 transformers.logging.set_verbosity_info()
32
33 training_args = Seq2SeqTrainingArguments(
34     predict_with_generate=True,
35     evaluation_strategy="steps",
36     per_device_train_batch_size=batch_size,
37     per_device_eval_batch_size=batch_size,
38     output_dir=".",
39     logging_steps=5,
40     eval_steps=10,
41     save_steps=10,
42     save_total_limit=2,
43     gradient_accumulation_steps=4,
44     num_train_epochs=10
45 )
```

Make the fine tuning run for around 100 steps. It will take around an hour. After each step, the model is stored in the checkpoints folder in the “files” section of colab.

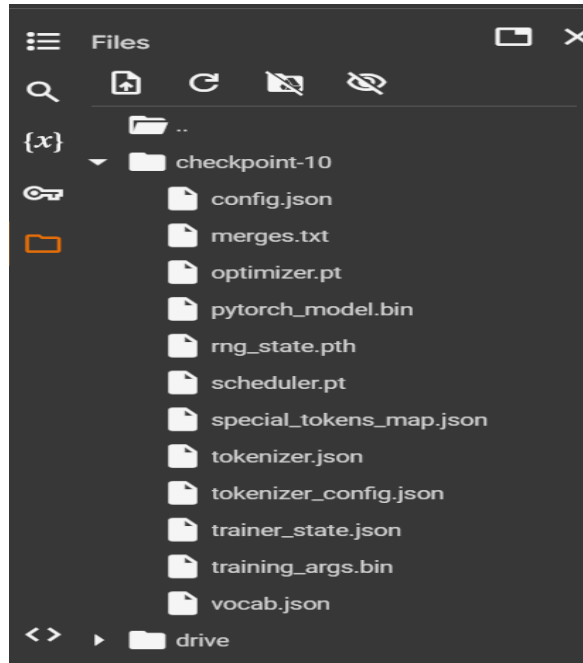
```
[ ] trainer = Seq2SeqTrainer(  
    model=led,  
    tokenizer=tokenizer,  
    args=training_args,  
    compute_metrics=compute_metrics,  
    train_dataset=train_dataset,  
    eval_dataset=val_dataset,  
)  
  
[ ] trainer.train()
```

/usr/local/lib/python3.10/dist-packages/transformers/optimization.py:306: FutureWarning: This implementation of AdamW is deprecated. Please use the implementation in torch.optim.AdamW instead. Warning is thrown since the implementation of AdamW is deprecated in the future.
warnings.warn(
***** Running training *****
 Num examples = 40920
 Num Epochs = 10
 Instantaneous batch size per device = 16
 Total train batch size (w. parallel, distributed & accumulation) = 64
 Gradient Accumulation steps = 4
 Total optimization steps = 6390
/usr/local/lib/python3.10/dist-packages/torch/utils/checkpoint.py:429: UserWarning: torch.utils.checkpoint: please pass use_reentrant=False if PyTorch version is < 1.10.0
warnings.warn(
[103/6390 48:14 < 50:02:40, 0.03 it/s, Epoch 0.16/10]

Step	Training Loss	Validation Loss	Rouge2 Precision	Rouge2 Recall	Rouge2 Fmeasure
10	2.092300	2.266404	0.130400	0.078000	0.086400
20	2.109100	2.220366	0.116700	0.049300	0.065900
30	2.063400	2.245599	0.156700	0.079200	0.093700
40	2.192200	2.179573	0.105400	0.061700	0.071300

We’ve set the epochs to 10. But don't run it for all time, as we run it on free colab resources (even if you try, it will disconnect automatically after sometime 😊). It's better to stop the execution by clicking the pause icon on the running cell after 45 mins.

After training it for 45 mins, you will see two checkpoint folders in the files section of colab. Those folders will contain all information about the model that we fine tuned. It will have the layer structure, hyper parameter values, fine tuned parameter values and so on. You will probably see checkpoint-90 and checkpoint-100 folders. We can use the checkpoint-100 for evaluation and testing if the model we trained can generate heading for a given paragraph. Ready???



Let's consider this paragraph as a sample:

“ The reason why I loved the top-down culture at Apple is that important decisions are taken faster. Having an expert giving you green light or not keeps the momentum. How many times in a bottom-up culture do we spend weeks and weeks, sometimes even months, trying to get alignment with +10 people, because every single person needs to agree with the point of view? It is exhausting. So again, my experience is that having that one leader to look up to to help guide decisions is time-saving, it helps us focus on the design craft, instead of project management.”


```

import pandas as pd
sample_paragraph = "The reason why I loved the top-down culture at Apple is that important decisions are taken fas
data = [sample_paragraph]
df = pd.DataFrame(data, columns=['paragraph'])
df['paragraph'][0]
from datasets import Dataset
df_test = Dataset.from_pandas(df)
df_test

Dataset({
  features: ['paragraph'],
  num_rows: 1
})

```

```

from datasets import load_metric
import torch

from datasets import load_dataset, load_metric
from transformers import LEDTokenizer, LEDForConditionalGeneration

# load tokenizer
tokenizer = LEDTokenizer.from_pretrained("/content/checkpoint-130")
model = LEDForConditionalGeneration.from_pretrained("/content/checkpoint-130").to("cuda").half()

def generate_answer(batch):
    inputs_dict = tokenizer(batch["paragraph"], padding="max_length", max_length=512, return_tensors="pt", truncation=True)
    input_ids = inputs_dict.input_ids.to("cuda")
    attention_mask = inputs_dict.attention_mask.to("cuda")
    global_attention_mask = torch.zeros_like(attention_mask)
    # put global attention on token
    # global_attention_mask[:, 0] = 1

    predicted_abstract_ids = model.generate(input_ids, attention_mask=attention_mask, global_attention_mask=global_attention_mask)
    batch["generated_heading"] = tokenizer.batch_decode(predicted_abstract_ids, skip_special_tokens=True)
    return batch

result = df_test.map(generate_answer, batched=True, batch_size=2)

```

Hurray!! It generated a heading like this.

```

result["generated_heading"]

['Have a leader to guide you.']

```

Make sure to move the latest checkpoint folder to the codecycle folder in your drive.

CONGRATULATIONS! You have successfully scrapped data from the web and used it for fine tuning a language model to generate headings for a given paragraph.

Submission Link : <https://forms.gle/Pja5PrMiuMAdEn8P7>

Section 3

Here's the additional task that you can try to get even more knowledge.

Summarization and Tag Generation for Paragraphs:

Enhance your model's functionality by not only generating headings but also:

- 1. Summarizing the paragraph into 2-3 concise sentences.**
- 2. Generating 3-5 relevant tags/keywords that best represent the key themes of the paragraph.**

Example:

Paragraph:

"In recent years, artificial intelligence (AI) has made significant strides in various industries, revolutionizing the way we live and work. From autonomous vehicles to personalized recommendations, AI-powered solutions have enhanced efficiency and productivity. However, with these advancements come concerns about ethics, privacy, and the future of employment. As AI continues to evolve, it is essential to strike a balance between innovation and responsibility."

Output:

- 1. Heading:** "Advancements and Ethical Considerations in Artificial Intelligence"
- 2. Summary:** "AI has transformed industries, improving efficiency through

automation and personalized services. Despite its benefits, AI poses challenges related to ethics, privacy, and job security. Ensuring responsible AI development is crucial for future growth."

3. **Tags:** AI, automation, ethics, privacy, job security

This task will help make your model more versatile by generating summaries and tags, which are useful for content organization and SEO.

Submission has to be done in Unstop platform itself...

THE END