Give Meaningful Names to Your Photos with AI

Introduction

Images, rich with untapped information, often come under the radar of search engines and data systems. Transforming this visual data into machine-readable language is no easy task, but it's where image captioning AI is useful. Here's how image captioning AI can make a difference:

- Improves accessibility: Helps visually impaired individuals understand visual content.
- Enhances SEO: Assists search engines in identifying the content of images.
- Facilitates content discovery: Enables efficient analysis and categorization of large image databases.
- Supports social media and advertising: Automates engaging description generation for visual content.
- Boosts security: Provides real-time descriptions of activities in video footage.
- Aids in education and research: Assists in understanding and interpreting visual materials.
- Offers multilingual support: Generates image captions in various languages for international audiences.
- Enables data organization: Helps manage and categorize large sets of visual data.
- Saves time: Automated captioning is more efficient than manual efforts.
- Increases user engagement: Detailed captions can make visual content more engaging and informative.

Learning objectives

At the end of this project, you will be able to:

- Implement an image captioning tool using the BLIP model from Hugging Face's Transformers
- Use Gradio to provide a user-friendly interface for your image captioning application
- Adapt the tool for real-world business scenarios, demonstrating its practical applications

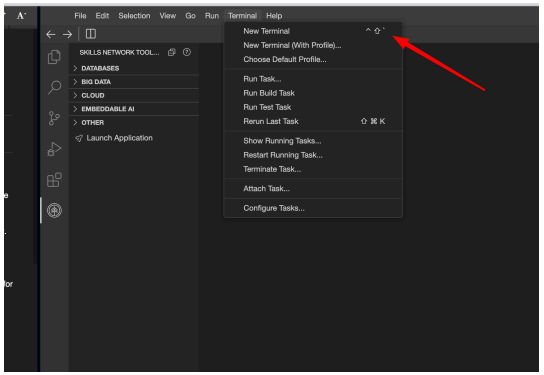
Generate by AI and enhanced by human

Setting up the environment and installing libraries

In this project, to build an AI app, you will use Gradio interface provided by Hugging Face.

Let's set up the environment and dependencies for this project. Open up a new terminal and make sure you are in the `home/project` directory.

Open a new terminal:



Open a new terminal

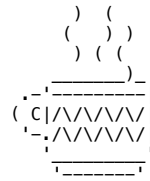
Create a Python virtual environment and install Gradio using the following commands in the terminal:

```
pip3 install virtualenv
virtualenv my_env # create a virtual environment my_env
source my_env/bin/activate # activate my_env
```

Then, install the required libraries in the environment:

```
# installing required libraries in my_env
pip install langchain==0.1.11 gradio==4.44.0 transformers==4.38.2 bs4==0.0.2 requests==2.31.0 torch==2.2.1
```

Have a cup of coffee, it will take 5 minutes.



Now, your environment is ready to create Python files.

Generating image captions with the BLIP model

Introducing: Hugging Face, Tranformers, and BLIP

Hugging Face is an organization that focuses on natural language processing (NLP) and artificial intelligence (AI).The organization is widely known for its open-source library called "***Transformers***" which provides thousands of pre-trained models to the community. The library supports a wide range of NLP tasks, such as translation, summarization, text generation, and more. ***Transformers*** has contributed significantly to the recent advancements in NLP, as it has made state-of-the-art models, such as BERT, GPT-2, and GPT-3, accessible to researchers and developers worldwide.

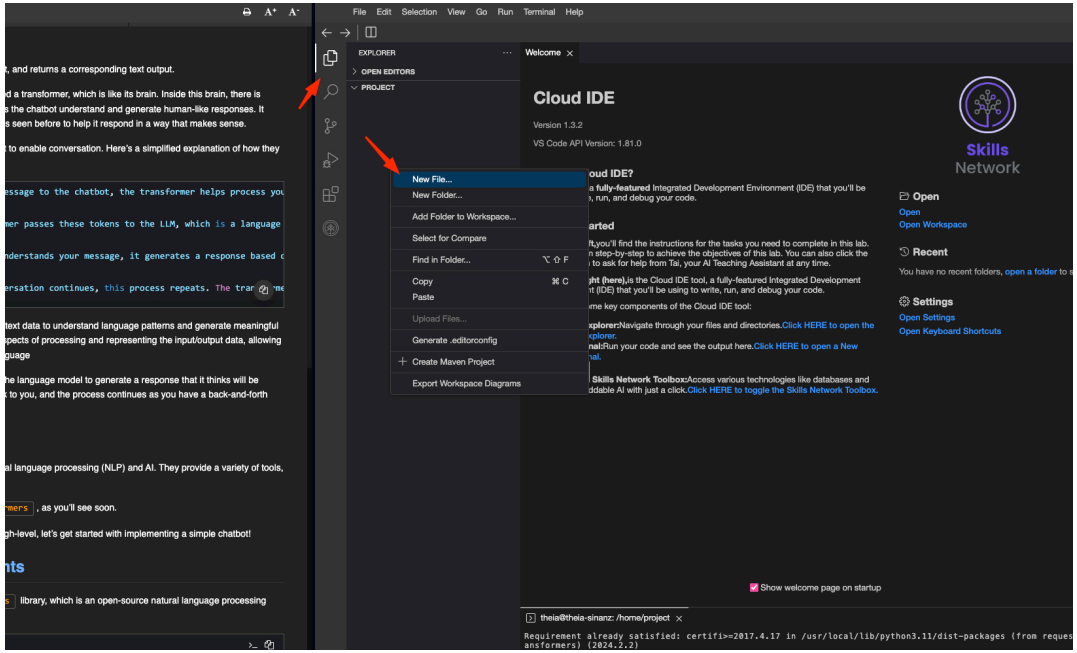
Tranformers library includes a model that can be used to capture information from images. The BLIP, or Bootstrapping Language-Image Pre-training, model is a tool that helps computers understand and generate language based on images. It's like teaching a computer to look at a picture and describe it, or answer questions about it.

Alright, now that you know what BLIP can do, let's get started with implementing a simple image captioning AI app!

Step 1: Import your required tools from the transformers library

You have already installed the package `transformers` during setting up the environment.

In the `project` directory, create a Python file, Click on `File Explorer`, then right-click in the explorer area and select `New File`. Name this new file `image_cap.py`. copy the various code segments below and paste them into the Python file.



New file

You will be using `AutoProcessor` and `BlipForConditionalGeneration` from the `transformers` library.

"Blip2Processor" and "Blip2ForConditionalGeneration" are components of the BLIP model, which is a vision-language model available in the Hugging Face Transformers library.

- **AutoProcessor** : This is a processor class that is used for preprocessing data for the BLIP model. It wraps a BLIP image processor and an OPT/T5 tokenizer into a single processor. This means it can handle both image and text data, preparing it for input into the BLIP model.

    Note: A tokenizer is a tool in natural language processing that breaks down text into smaller, manageable units (tokens), such as words or phrases, enabling models to analyze and understand the text.

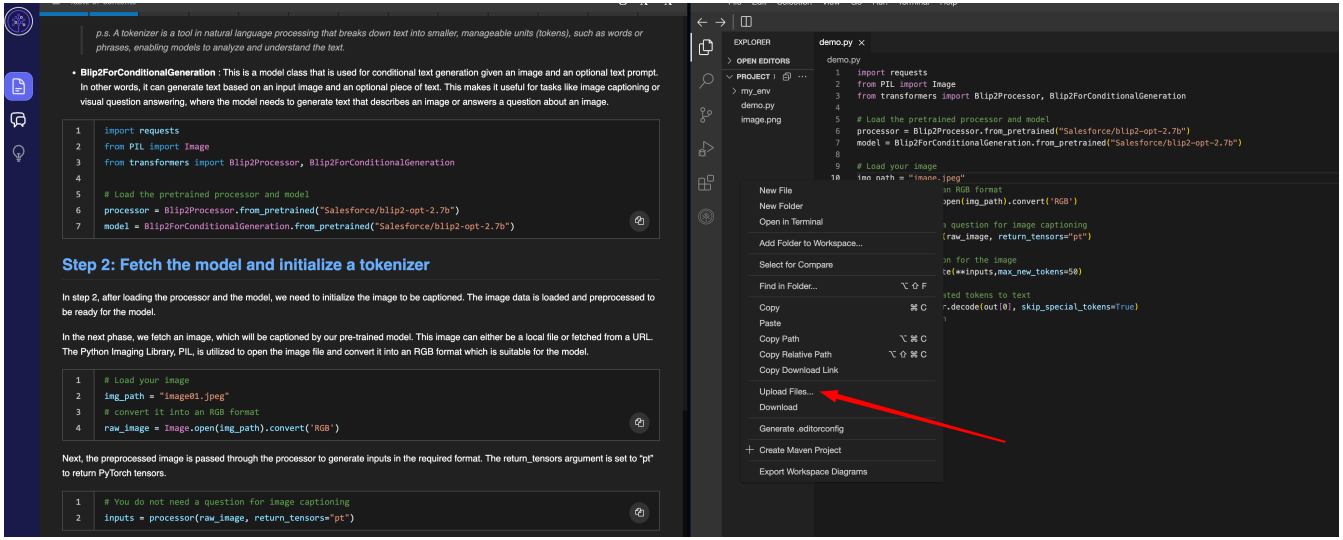
- **BlipForConditionalGeneration** : This is a model class that is used for conditional text generation given an image and an optional text prompt. In other words, it can generate text based on an input image and an optional piece of text. This makes it useful for tasks like image captioning or visual question answering, where the model needs to generate text that describes an image or answer a question about an image.

```
import requests
from PIL import Image
from transformers import AutoProcessor, BlipForConditionalGeneration
# Load the pretrained processor and model
processor = AutoProcessor.from_pretrained("Salesforce/blip-image-captioning-base")
model = BlipForConditionalGeneration.from_pretrained("Salesforce/blip-image-captioning-base")
```

Step 2: Fetch the model and initialize a tokenizer

After loading the processor and the model, you need to initialize the image to be captioned. The image data needs to be loaded and pre-processed to be ready for the model.

To load the image right-click anywhere in the Explorer (on the left side of code pane), and click `Upload Files...` (shown in image below). You can upload any image from your local files, and modify the `img_path` according to the name of the image.



Upload files

In the next phase, you fetch an image, which will be captioned by your pre-trained model. This image can either be a local file or fetched from a URL. The Python Imaging Library, PIL, is used to open the image file and convert it into an RGB format which is suitable for the model.

```
# Load your image, DONT FORGET TO WRITE YOUR IMAGE NAME
img_path = "YOUR IMAGE NAME.jpeg"
# convert it into an RGB format
image = Image.open(img_path).convert('RGB')
```

Next, the pre-processed image is passed through the processor to generate inputs in the required format. The return_tensors argument is set to "pt" to return PyTorch tensors.

```
# You do not need a question for image captioning
text = "the image of"
inputs = processor(images=image, text=text, return_tensors="pt")
```

You then pass these inputs into your model's generate method. The argument `max_new_tokens=50` specifies that the model should generate a caption of up to 50 tokens in length.

> The two asterisks (**) in Python are used in function calls to unpack dictionaries and pass items in the dictionary as keyword arguments to the function. `**inputs` is unpacking the inputs dictionary and passing its items as arguments to the model.

```
# Generate a caption for the image
outputs = model.generate(**inputs, max_length=50)
```

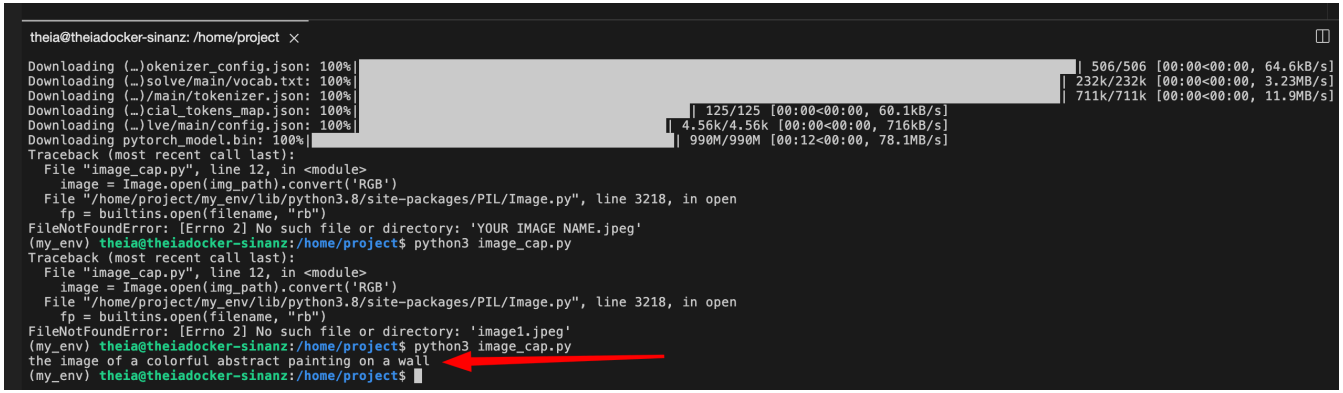
Finally, the generated output is a sequence of tokens. To transform these tokens into human-readable text, you use the decode method provided by the processor. The `skip_special_tokens` argument is set to `True` to ignore special tokens in the output text.

```
# Decode the generated tokens to text
caption = processor.decode(outputs[0], skip_special_tokens=True)
# Print the caption
print(caption)
```

Save your Python file and run it to see the result.

```
python3 image_cap.py
```

And you have the image's caption, generated by your model! This caption is a textual representation of the content of the image, as interpreted by the BLIP model.



Caption output

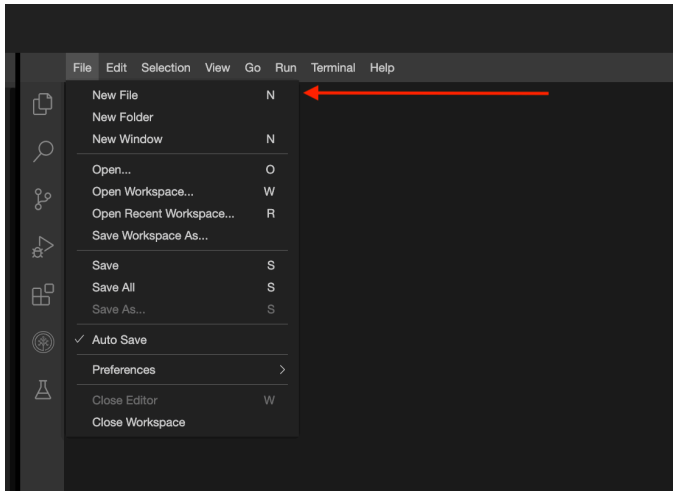
Image captioning app with Gradio

Now that you understand the mechanism of image captioning, let's create a proper application with an intuitive interface. You can utilize Gradio, a tool provided by Hugging Face, for this purpose. To begin, you will have a brief introduction to Gradio. Following that, as an exercise, you will be tasked with implementing the image captioning application using the Gradio interface.

Quickstart Gradio: Creating a simple demo

Let's get familiar with Gradio by creating a simple app:

Still in the `project` directory, create a Python file and name it `hello.py`.



New file

Open `hello.py`, copy and paste the following Python code and save the file.

```
import gradio as gr
def greet(name):
    return "Hello " + name + "!"
demo = gr.Interface(fn=greet, inputs="text", outputs="text")
demo.launch(server_name="0.0.0.0", server_port= 7860)
```

The above code creates a **gradio.Interface** called `demo`. It wraps the `greet` function with a simple text-to-text user interface that you could interact with.

The **gradio.Interface** class is initialized with 3 required parameters:

- fn: the function to wrap a UI around
- inputs: which component(s) to use for the input (e.g. “text”, “image” or “audio”)
- outputs: which component(s) to use for the output (e.g. “text”, “image” or “label”)

The last line `demo.launch()` launches a server to serve your `demo`.

Launching the demo app

Now go back to the terminal and make sure that the `my_env` virtual environment name is displayed at the begining of the line.

Now run the following command to execute the Python script.

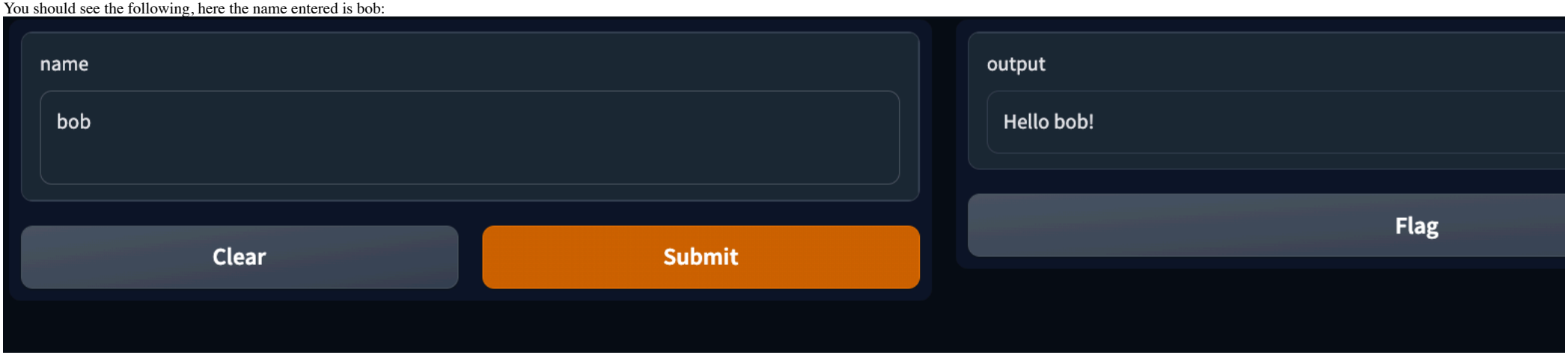
```
python3 hello.py
```

As the Python code is served by local host, click the button below and you will be able to see the simple application you created. Feel free to play around with the input and output of the web app!

Click here to see the application:

Web Application

You should see the following, here the name entered is bob:



Input and output

If you finish playing with the app and want to exit, **press `ctrl+c` in the terminal and close the application tab**.

You just had a first taste of the Gradio interface, it's easy right? If you wish to learn a little bit more about customization in Gradio, you are invited to take the guided project called **Bring your Machine Learning model to life with Gradio**. You can find it under **Courses & Projects** on cognitiveclass.ai!

Exercise: Implement image captioning app with Gradio

In this exercise, you will walk through the steps to create a web application that generates captions for images using the BLIP-2 model and the Gradio library. Follow the steps below:

Step 1: Set up the environment

- Make sure you have the necessary libraries installed. Run `pip install gradio transformers Pillow` to install Gradio, Transformers, and Pillow.

- Import the required libraries:

Now, let's create a new Python file and call it `image_captioning_app.py`.

```
import gradio as gr
import numpy as np
from PIL import Image
from transformers import AutoProcessor, BlipForConditionalGeneration
```

Step 2: Load the pretrained model

- Load the pretrained processor and model:

```
processor = # write your code here
model = # write your code here
```

Step 3: Define the image captioning function

- Define the `caption_image` function that takes an input image and returns a caption:

```
def caption_image(input_image: np.ndarray):
    # Convert numpy array to PIL Image and convert to RGB
    raw_image = Image.fromarray(input_image).convert('RGB')

    # Process the image

    # Generate a caption for the image
    # Decode the generated tokens to text and store it into `caption`

    return caption
```

Step 4: Create the Gradio interface

- Use the `gr.Interface` class to create the web app interface:

```
iface = gr.Interface(
    fn=caption_image,
    inputs=gr.Image(),
    outputs="text",
    title="Image Captioning",
    description="This is a simple web app for generating captions for images using a trained model."
)
```

Step 5: Launch the Web App

- Start the web app by calling the `launch()` method:

```
iface.launch()
```

▶ Click here for the answer

Step 6: Run the application

- Save the complete code to a Python file, for example, `image_captioning_app.py`.
- Open a terminal or command prompt, navigate to the directory where the file is located, and run the command

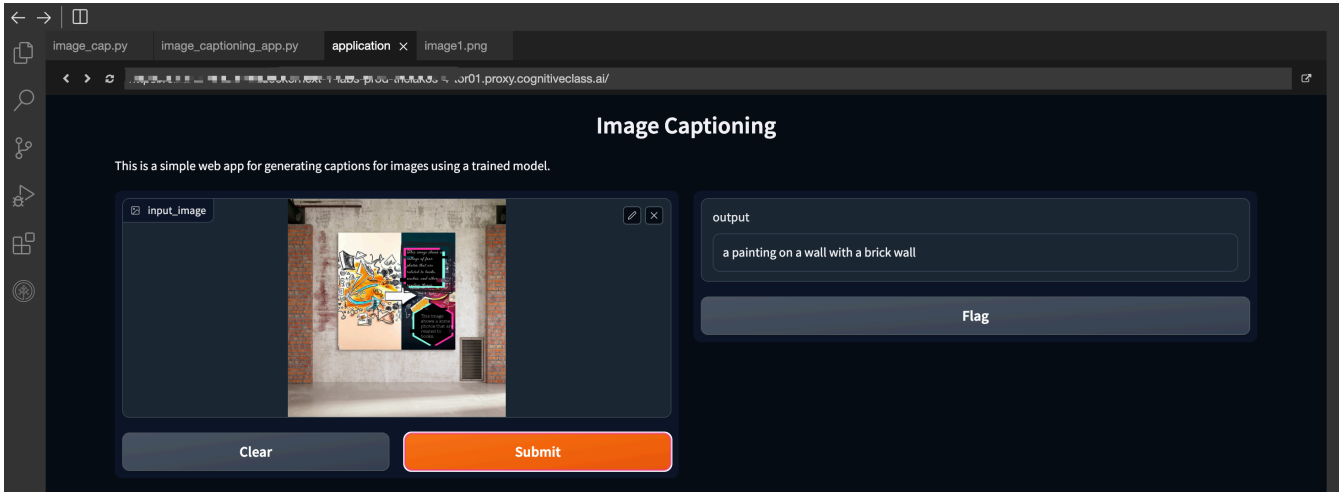
```
python3 image_captioning_app.py
```

Click here to start your web app:

Web Application

Press `ctrl + c` to quit the application.

You will have such output in the new windows:



If you are running locally: Interact with the web App:

- The web app should start running and display a URL where you can access the interface.
- Open the provided URL in a web browser (in the terminal).
- You should see an interface with an image upload box.

Congratulations! You have created an image captioning web app using Gradio and the BLIP model. You can further customize the interface, modify the code, or experiment with different models and settings to enhance the application's functionality.

Scenario: How image captioning helps a business

Business scenario on news and media:

A news agency publishes hundreds of articles daily on its website. Each article contains several images relevant to the story. Writing appropriate and descriptive captions for each image manually is a tedious task and might slow down the publication process.

In this scenario, your image captioning program can expedite the process:

1. Journalists write their articles and select relevant images to go along with the story.
2. These images are then fed into the image captioning program (instead of manually insert description for each image).
3. The program processes these images and generates a text file with the suggested captions for each image.
4. The journalists or editors review these captions. They might use them as they are, or they might modify them to better fit the context of the article.
5. These approved captions then serve a dual purpose:
    - Enhanced accessibility: The captions are integrated as alternative text (alt text) for the images in the online article. Visually impaired users, using screen readers, can understand the context of the images through these descriptions. It helps them to have a similar content consumption experience as sighted users, adhering to the principles of inclusive and accessible design.
    - Improved SEO: Properly captioned images with relevant keywords improve the article's SEO. Search engines like Google consider alt text while indexing, and this helps the article to appear in relevant search results, thereby driving organic traffic to the agency's website. This is especially useful for image search results.
6. Once the captions are approved, they are added to the images in the online article.

By integrating this process, the agency not only expedites its publication process but also ensures all images come with appropriate descriptions, enhancing the accessibility for visually impaired readers, and improving the website's SEO. This way, the agency broadens its reach and engagement with a more diverse audience base.

Let's implement automated image captioning tool

In this section, you implement an automated image captioning program that works directly from a URL. The user provides the URL, and the code generates captions for the images found on the webpage. The output is a text file that includes all the image URLs along with their respective captions (like the image below). To accomplish this, you use BeautifulSoup for parsing the HTML content of the page and extracting the image URLs.

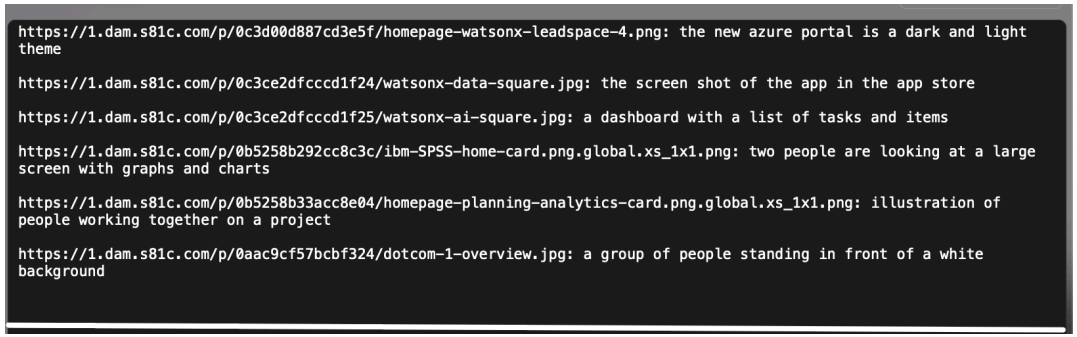


Image urls

Let's get started:

Firstly, you send a HTTP request to the provided URL and retrieve the webpage's content. This content is then parsed by BeautifulSoup, which creates a parse tree from page's HTML. You look for 'img' tags in this parse tree as they contain the links to the images hosted on the webpage.

```
# URL of the page to scrape
url = "https://en.wikipedia.org/wiki/IBM"
# Download the page
response = requests.get(url)
# Parse the page with BeautifulSoup
soup = BeautifulSoup(response.text, 'html.parser')
```

After extracting these URLs, you iterate through each one of them. You send another HTTP request to download the image data associated with each URL.

> It's important to note that this operation is performed synchronously in your current implementation. That means each image is downloaded one at a time, which could be slow for webpages with a large number of images. For a more efficient approach, one could explore asynchronous programming methods or the concurrent.futures library to download multiple images simultaneously.

```
# Find all img elements
img_elements = soup.find_all('img')
# Iterate over each img elements
for img_element in img_elements:
...
```

Complete the code below to make it work:

Create a new python file and call it `automate_url_captioner.py`, and copy the below code. Complete the blank part to make it work.

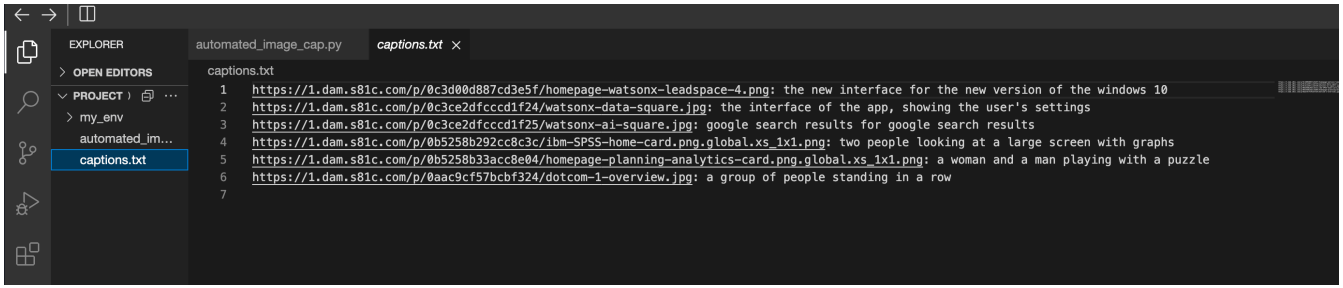
```
import requests
from PIL import Image
from io import BytesIO
from bs4 import BeautifulSoup
from transformers import AutoProcessor, BlipForConditionalGeneration
# Load the pretrained processor and model
processor = # fill the pretrained model
model = # load the blip model
# URL of the page to scrape
url = "https://en.wikipedia.org/wiki/IBM"
# Download the page
response = requests.get(url)
# Parse the page with BeautifulSoup
```

```
soup = BeautifulSoup(response.text, 'html.parser')
# Find all img elements
img_elements = soup.find_all('img')
# Open a file to write the captions
with open("captions.txt", "w") as caption_file:
    # Iterate over each img element
    for img_element in img_elements:
        img_url = img_element.get('src')
        # Skip if the image is an SVG or too small (likely an icon)
        if 'svg' in img_url or '1x1' in img_url:
            continue
        # Correct the URL if it's malformed
        if img_url.startswith('//'):
            img_url = 'https:' + img_url
        elif not img_url.startswith('http://') and not img_url.startswith('https://'):
            continue  # Skip URLs that don't start with http:// or https://
        try:
            # Download the image
            response = requests.get(img_url)
            # Convert the image data to a PIL Image
            raw_image = Image.open(BytesIO(response.content))
            if raw_image.size[0] * raw_image.size[1] < 400:  # Skip very small images
                continue

            raw_image = raw_image.convert('RGB')
            # Process the image
            inputs = processor(raw_image, return_tensors="pt")
            # Generate a caption for the image
            out = model.generate(**inputs, max_new_tokens=50)
            # Decode the generated tokens to text
            caption = processor.decode(out[0], skip_special_tokens=True)
            # Write the caption to the file, prepended by the image URL
            caption_file.write(f"{img_url}: {caption}\n")
        except Exception as e:
            print(f"Error processing image {img_url}: {e}")
            continue
```

▶ Click here for the answer

As an output, you will have a new file in the explorer (same directory) with the name `captions.txt` (as shown in the image below).



Captions.txt

Bonus: Image captioning for local files (Run locally if using Blip2)

With a few modifications, you can adapt the code to operate on local images. This involves utilizing the glob library to sift through all image files in a specific directory and then writing the generated captions to a text file.

Additionally, you can make use of the Blip2 model, which is a more powerful pre-trained model for image captioning. In fact, you can easily incorporate any new pre-trained model that becomes available, as they are continuously developed to be more powerful. In the example below, we demonstrate the usage of the Blip2 model. However, please be aware that the Blip2 model requires 10GB of space, which prevents us from running it in the CloudIDE environment.

```
import glob
...
# Specify the directory where your images are
image_dir = "/path/to/your/images"
image_exts = ["jpg", "jpeg", "png"]  # specify the image file extensions to search for
```

```
# Open a file to write the captions
with open("captions.txt", "w") as caption_file:
# Iterate over each image file in the directory
    for image_ext in image_exts:
        for img_path in glob.glob(os.path.join(image_dir, f"*.{image_ext}")):
            # Load your image
            raw_image = Image.open(img_path).convert('RGB')
            ...
```

Try to implement yourself.

▼ Click here to see the complete version

```
import os
import glob
import requests
from PIL import Image
from transformers import Blip2Processor, Blip2ForConditionalGeneration #Blip2 models
# Load the pretrained processor and model
processor = Blip2Processor.from_pretrained("Salesforce/blip2-opt-2.7b")
model = Blip2ForConditionalGeneration.from_pretrained("Salesforce/blip2-opt-2.7b")
# Specify the directory where your images are
image_dir = "/path/to/your/images"
image_exts = ["jpg", "jpeg", "png"]  # specify the image file extensions to search for
# Open a file to write the captions
with open("captions.txt", "w") as caption_file:
    # Iterate over each image file in the directory
    for image_ext in image_exts:
        for img_path in glob.glob(os.path.join(image_dir, f"*.{image_ext}")):
            # Load your image
            raw_image = Image.open(img_path).convert('RGB')
            # You do not need a question for image captioning
            inputs = processor(raw_image, return_tensors="pt")
            # Generate a caption for the image
            out = model.generate(**inputs, max_new_tokens=50)
            # Decode the generated tokens to text
            caption = processor.decode(out[0], skip_special_tokens=True)
            # Write the caption to the file, prepended by the image file name
            caption_file.write(f"{os.path.basename(img_path)}: {caption}\n")
```

Conclusion

Congratulations on completing this guided project! You have now mastered image captioning AI using Gradio and IBM Code Engine.

Next steps

You can deploy your app on the internet using IBM Code Engine. In the following optional section, we offer a step-by-step guide to assist you in doing so.

At the end of this guided project, you deployed an application to a Kubernetes cluster using IBM Code Engine. You used a shared cluster provided to you by the IBM Developer Skills Network. If you wish to deploy your containerized app and get a permanent URL of the app outside of the Code Engine CLI on your local machine, you can learn more about Kubernetes and containers. You can get your own free Kubernetes cluster and your own free IBM Container Registry.

Author(s)

Sina Nazeri