Building Your First GenAI-Powered Image-Based Web Application: AI Nutrition Coach

Estimated time needed: 30 minutes

Welcome! In this project, you'll learn to build your very own AI-powered Nutrition Coach! In this 30-minute, hands-on workshop, you'll dive into the world of generative AI and visual recognition, leveraging the powerful Llama 4 Maverick 17B 128E Instruct FP8 model to estimate caloric content from food images. You'll integrate this technology with comprehensive nutritional databases to deliver instant calorie counts and personalized dietary advice.

Learning objectives

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

- **Develop** a Flask web application integrated with AI capabilities.
- **Utilize** the `ibm-watsonx-ai` library to interact with the Llama 4 Maverick 17B 128E Instruct FP8 model to recognize and analyze food items.
- **Integrate** visual recognition with nutritional databases for accurate calorie estimations.

Let's embark on this journey to transform your development skills and create an intelligent, AI-driven application!

> **Disclaimer**: Since we are utilizing generative AI, the output may not always be fully accurate or consistent. Please apply your discretion in assessing each response.

Tips for the best experience!

Keep the following tips handy and refer to them at any point of confusion throughout the lab. Do not worry if they seem irrelevant now. You will go through everything step by step later.

- At any point throughout the lab, if you are lost, click on **Table of Contents** icon on the top left of the page and navigate to your desired content.

- Whenever you make changes to a file, be sure to save your work. Cloud IDE automatically saves any changes you make to your files immediately. You can also save from the toolbar.
- **After setting up your virtual environment in the next step**, be sure to always have it activated for the rest of the lab. You'll know it's activated when you see `(my_env)` before the prompt, like this:
- If `(my_env)` doesn't appear, activate the environment by running the following command in the terminal:

  ```
  source my_env/bin/activate
  ```

- For running the application, always ensure application file is running in the background before opening the Web Application.
- You run a code block by clicking `>_` on bottom right.

  ```
  python app.py
  ```

- Always ensure that your current directory is `/home/project/cal_coach_app`. If you are not in the correct folder, certain code files may fail to run. Use the `cd` command to navigate to the correct location.
- Make sure you are accessing the application through port `5000`. Clicking the purple Web Application button will run the app through port 5000 automatically.

Web Application

Note: If this "Web Application" button does not work, follow the following picture instructions to launch the application.

- If you get an error about not being able to access another port (for example, 8888), just `refresh` the app by clicking the small refresh icon. In case of other errors related to the server, simply refresh the page as well. (The following image is for illustration purpose only.)
- To stop execution of `app.py` in addition to closing the application tab, press `Ctrl+C` in terminal.
- If you encounter an error running the application or after you entered your desired keyword, try refreshing the app using the button on the top of the application's page. You can try inputting a different query too.
- Typically, using the models provided by watsonx.ai would require watsonx credentials, including an API key and a project ID. However, in this lab, these credentials are not needed.

Setting up your development environment

Before diving into development, let's set up your project environment in the Cloud IDE. This environment is based on Ubuntu 22.04 and provides all the tools you need to build your AI-driven Flask application.

Step 1: Create your project directory

Open the terminal in Cloud IDE and run:

```
mkdir cal_coach_app
cd cal_coach_app
```

This creates a new directory for your project and navigates to it.

Step 2: Set up a Python virtual environment

Initialize a new Python virtual environment:

```
python3.11 -m venv my_env
source my_env/bin/activate
```

Step 3: Install the library and packages

You'll now install `ibm-watsonx-ai`, which has many watsonx.ai features, along with other neccessary packages. In this lab, you'll use this library to help you configure and utilize your vision-instruct model.

With your virtual environment activated, install the neccessary packages:

```
pip install ibm-watsonx-ai==1.1.20 image==1.5.33 flask requests==2.32.0
```

Now that your environment is set up, you're ready to start building your application!

Let's build your app!

NOTE: At any point throughout the project, if you encounter issues, click on Table of Contents and navigate to the solutions.

Now that you understand the model, let's start by creating the backbone of your Flask application. We'll set up a basic structure that we'll enhance with AI capabilities in the following steps.

Our project is structured as follows:

```
cal_coach_app/
|-- app.py
|-- templates/
|   |-- index.html
|-- static/
    |-- style.css
```

Make sure that all your files are in the intended folders before running the application. Click on the Execute button below to create the folders and files. Once they are created, return to the main directory where `app.py` is located.

```
touch app.py
mkdir templates
cd templates
touch index.html
cd ..
mkdir static
cd static
touch style.css
cd ..
```

Click on Explorer to make sure that all the files have been created correctly.

App overview: AI Nutrition Coach

The **AI Nutrition Coach** app empowers users with the ability to make informed dietary choices using advanced AI technology. This app leverages the **Llama 4 Maverick 17B 128E Instruct FP8 model** model to analyze food images and provide valuable nutritional information. Here's a breakdown of the app's key features:

1. Food identification and calorie estimation

The app allows users to upload images of their meals. Once the image is uploaded, the AI model identifies the food items using its visual recognition capabilities. It then estimates the **caloric content** of each item based on a comprehensive nutritional database. This feature eliminates the need for manual entry and guessing, offering a quick and accurate overview of

the meal's calorie count.

2. Nutritional breakdown

Beyond estimating total calories, the app provides a detailed **nutritional breakdown** of key components like fats, proteins, and carbohydrates. This feature helps users better understand the nutritional composition of their meals and how they align with their dietary goals, whether it's weight loss, muscle gain, or maintaining a balanced diet.

3. Personalized nutritional advice

After analyzing the meal, the app offers **personalized dietary advice** based on the user's health objectives and the nutritional content of the meal. For instance, it might suggest increasing fiber intake if the analyzed meal is low in it or provide recommendations to stay within specific macronutrient ranges.

Step 1: Create your main application file

Create a new file named `app.py` by clicking on the button below.

Open **app.py** in IDE

Add the following code to set up a basic Flask app:

```
import requests
import re
import base64
import os
from ibm_watsonx_ai import Credentials
from ibm_watsonx_ai import APIClient
from ibm_watsonx_ai.foundation_models import Model, ModelInference
from ibm_watsonx_ai.foundation_models.schema import TextChatParameters
from ibm_watsonx_ai.metanames import GenTextParamsMetaNames
from PIL import Image
from flask import Flask, render_template, request, redirect, url_for, flash
app = Flask(__name__)
### You will add code from Step 2 here
### Step 3
### Step 4
### Step 5
@app.route('/generate', methods=['POST'])
def index():
    # This is where you'll add your main application logic later, after step 5
    #TODO
if __name__ == '__main__':
    app.run(debug=True)
```

Let's break down this code:

- Import necessary modules from Flask.
- Create a Flask application instance.
- Define a route `/generate` that will handle POST requests. This is where your AI logic will go.
- The `if __name__ == '__main__':` block ensures the Flask development server runs when you execute this file directly.

You've set up the foundation of your application. In the next sections, we'll integrate AI capabilities and enhance its functionality.

Step 2: Initialize model instance

In this step, you will set up your model, just as you did previously in `multimodal_queries.py`. All the codes below are added after `app = Flask(__name__)` where it says `### You will add code from Step 2 here`

```
credentials = Credentials(
                  url = "https://us-south.ml.cloud.ibm.com",
                  # api_key = "<YOUR_API_KEY>" # Normally you'd put an API key here, but we've got you covered here
                 )
client = APIClient(credentials)
model_id = "meta-llama/llama-4-maverick-17b-128e-instruct-fp8"
project_id = "skills-network"
params = TextChatParameters()
model = ModelInference(
    model_id=model_id,
    credentials=credentials,
    project_id=project_id,
    params=params
)
```

Step 3: Define function to handle image encoding

Next, you will define a function to encode the uploaded image into base64 format so that it can be used with the model. Add the code below to `app.py` under `### Step 3`

```
def input_image_setup(uploaded_file):
    """
    Encodes the uploaded image file into a base64 string to be used with AI models.
    Parameters:
    - uploaded_file: File-like object uploaded via a file uploader (Streamlit or other frameworks)
```

```
    Returns:
    - encoded_image (str): Base64 encoded string of the image data
    """
    # Check if a file has been uploaded
    if uploaded_file is not None:
        # Read the file into bytes
        bytes_data = uploaded_file.read()
        # Encode the image to a base64 string
        encoded_image = base64.b64encode(bytes_data).decode("utf-8")
        return encoded_image
    else:
        raise FileNotFoundError("No file uploaded")
```

Code explanation: `input_image_setup`

The `input_image_setup` function is designed to handle an uploaded image file and convert it into a **base64-encoded string**, which is a common format for transmitting binary data in text-based systems.

Function purpose

The purpose of this function is to:

- **Read** the uploaded image file as bytes.
- **Encode** the image data into a base64 string.
- **Return** the encoded string, which can be passed to AI models for processing.

Steps explained

1. **Check for uploaded file**
   The function first checks if a file has been uploaded by ensuring that `uploaded_file` is not `None`. If no file is provided, the function raises a `FileNotFoundError`.
2. **Read the file as bytes** `bytes_data = uploaded_file.read()`
   Once the file is verified to exist, it reads the file into a byte format using the `.read()` method. This step is necessary, as base64 encoding requires the data in a binary format.
3. **Encode the image data to base64**
   The function then encodes the byte data using Python's built-in `base64` library. The encoded string is also decoded into a standard UTF-8 string, making it suitable for sending as a JSON field or transmitting in APIs.
4. **Return the encoded string**
   Finally, the function returns the base64-encoded string, which can be directly used with AI models for image processing tasks.

Define helper functions (cont.)

NOTE: At any point throughout the project, if you encounter issues, click on Table of Contents and navigate to the solutions.

Step 4: Define a function to format the model's output

Next, define a function to format the model's output into consistent, readable format. Add the code below to `app.py` where it says `### Step 4`

```
def format_response(response_text):
    """
    Formats the model response to display each item on a new line as a list.
    Converts numbered items into HTML `<ul>` and `<li>` format.
    Adds additional HTML elements for better presentation of headings and separate sections.
    """
    # Replace section headers that are bolded with '**' to HTML paragraph tags with bold text
    response_text = re.sub(r"\*\*(.*?)\*\*", r"<p><strong>\1</strong></p>", response_text)
    # Convert bullet points denoted by "*" to HTML list items
    response_text = re.sub(r"(?m)^\s*\*\s(.*)", r"<li>\1</li>", response_text)
    # Wrap list items within <ul> tags for proper HTML structure and indentation
    response_text = re.sub(r"(<li>.*?</li>)+", lambda match: f"<ul>{match.group(0)}</ul>", response_text, flags=re.DOTALL)
    # Ensure that all paragraphs have a line break after them for better separation
    response_text = re.sub(r"</p>(?=<p>)", r"</p><br>", response_text)
    # Ensure the disclaimer and other distinct paragraphs have proper line breaks
    response_text = re.sub(r"(\n|\\n)+", r"<br>", response_text)
    return response_text
```

Code explanation: `format_response`

The `format_response` function is designed to process text generated by a model, transforming it into well-structured HTML elements. This function is particularly useful when the model's output contains lists, headings, or separate sections that need to be displayed in a visually organized way.

Function purpose

The main purpose of this function is to:

- **Format section headings** using HTML paragraph and bold tags.
- **Convert bullet points** into HTML unordered lists (`<ul>`) and list items (`<li>`).
- **Ensure proper line breaks** between sections for better readability.

Steps explained

1. **Convert bold headers to HTML paragraph tags**
   The function first looks for text enclosed between double asterisks (**) using a regular expression. It then converts these into HTML paragraph tags with bold text.
2. **Transform bullet points to HTML list items**
   Next, the function identifies lines that start with a * symbol, indicating a bullet point. It replaces each bullet point with an HTML list item (<li>). This helps present the content as a proper HTML list.
3. **Wrap list items within <ul> tags**
   After converting bullet points to list items, the function wraps consecutive list items within <ul> tags to form an unordered list. This creates a structured HTML list format.
4. **Add line breaks between paragraphs**
   The function ensures that all paragraphs have line breaks between them by adding
 tags. This makes the content more readable by separating distinct sections.
5. **Handle newline characters**
   Any remaining newline characters are replaced with
 tags to maintain proper line breaks in the formatted HTML output.

Step 5: Define a function to generate model response

Next, define a function for interacting with the model to generate a response based on an image and a user query. This function also formats the model's response using HTML elements to enhance its presentation. Add the code below to `app.py` where it says `### Step 5`

```
def generate_model_response(encoded_image, user_query, assistant_prompt):
    """
    Sends an image and a query to the model and retrieves the description or answer.
    Formats the response using HTML elements for better presentation.
    """
    # Create the messages object
    messages = [
        {
            "role": "user",
            "content": [
                {"type": "text", "text": assistant_prompt + "\n\n" + user_query},
                {"type": "image_url", "image_url": {"url": "" + encoded_image}}
            ]
        }
    ]
    try:
        # Send the request to the model
        response = model.chat(messages=messages)
        raw_response = response['choices'][0]['message']['content']
        # Format the raw response text using the format_response function
        formatted_response = format_response(raw_response)
        return formatted_response
    except Exception as e:
        print(f"Error in generating response: {e}")
        return "<p>An error occurred while generating the response.</p>"
```

Update the main logic of your application

NOTE: At any point throughout the project, if you encounter issues, click on Table of Contents and navigate to the solutions.

Now that you have finished defining all the helper functions, it's time to update the main logic of your application.

```
@app.route("/", methods=["GET", "POST"])
def index():
    if request.method == "POST":
        # Retrieve user inputs
        user_query = request.form.get("user_query")
        uploaded_file = request.files.get("file")
        if uploaded_file:
            # Process the uploaded image
            encoded_image = input_image_setup(uploaded_file)
            if not encoded_image:
                flash("Error processing the image. Please try again.", "danger")
                return redirect(url_for("index"))
            # Assistant prompt (can be customized)
            assistant_prompt = """
            You are an expert nutritionist. Your task is to analyze the food items displayed in the image and provide a detailed nutritional asses
        1. **Identification**: List each identified food item clearly, one per line.
        2. **Portion Size & Calorie Estimation**: For each identified food item, specify the portion size and provide an estimated number of calor
        - **[Food Item]**: [Portion Size], [Number of Calories] calories
        Example:
        *   **Salmon**: 6 ounces, 210 calories
        *   **Asparagus**: 3 spears, 25 calories
        3. **Total Calories**: Provide the total number of calories for all food items.
        Example:
        Total Calories: [Number of Calories]
        4. **Nutrient Breakdown**: Include a breakdown of key nutrients such as **Protein**, **Carbohydrates**, **Fats**, **Vitamins**, and **Mine
        Example:
        *   **Protein**: Salmon (35g), Asparagus (3g), Tomatoes (1g) = [Total Protein]
        5. **Health Evaluation**: Evaluate the healthiness of the meal in one paragraph.
        6. **Disclaimer**: Include the following exact text as a disclaimer:
        The nutritional information and calorie estimates provided are approximate and are based on general food data.
        Actual values may vary depending on factors such as portion size, specific ingredients, preparation methods, and individual variations.
        For precise dietary advice or medical guidance, consult a qualified nutritionist or healthcare provider.
        Format your response exactly like the template above to ensure consistency.
        """
            # Generate the model's response
            response = generate_model_response(encoded_image, user_query, assistant_prompt)
            # Render the result
            return render_template("index.html", user_query=user_query, response=response)
        else:
            flash("Please upload an image file.", "danger")
            return redirect(url_for("index"))
    return render_template("index.html")
if __name__ == "__main__":
```

```
    app.run(debug=True)
```

This Flask app route defines the core functionality for handling user interactions in your application.

Function purpose

The `index` function serves as the main route (`"/"`) for the web app. It processes both **GET** and **POST** requests and handles the following tasks:

- **Retrieves user inputs** such as an image file and a user query.
- **Processes the uploaded image** by encoding it into a base64 format.
- **Generates a response** using an AI model based on a carefully crafted prompt for nutritional analysis.
- **Displays the output** on a webpage.

Steps explained

1. **Check for HTTP POST requests**
   The route starts by checking if the request is a **POST** request. This means that the user has submitted a form with an image and a query.

2. **Retrieve user inputs**
   The app retrieves the text input from the form (`user_query`) and the uploaded image file (`uploaded_file`). These are obtained using Flask's request object.

3. **Process the uploaded image**
   If an image file is uploaded, the function calls `input_image_setup` to process and encode the image. This encoding is essential for feeding the image data into the AI model. If the image encoding fails, an error message is flashed, and the user is redirected back to the main page.

4. **Assistant prompt explanation**
   The `assistant_prompt` is a critical component of this route. This text is designed to guide the AI model to generate a detailed and structured nutritional assessment. Let's break down the key elements of the prompt:

   - **Expert Role:** The model is instructed to act as an expert nutritionist, setting a clear context for the response.
   - **Food Identification:** The model is prompted to identify each food item clearly and list them separately.
   - **Portion Size & Calorie Estimation:** The model is instructed to estimate the portion size and calorie count for each identified item, using bullet points for better organization.
   - **Total Calories:** A summary of total calories for all food items is requested.
   - **Nutrient Breakdown:** A detailed breakdown of key nutrients like proteins, carbohydrates, fats, vitamins, and minerals is requested. Each nutrient's contribution from each food item should be clearly listed.
   - **Health Evaluation:** The model should provide an overall evaluation of the healthiness of the meal.
   - **Disclaimer:** A standard disclaimer is included to inform users that the provided nutritional information is approximate.

   The assistant prompt template aims to ensure consistency in the AI-generated response by providing a **structured format** and clear guidelines.

5. **Generate the AI model's response**
   The function calls `generate_model_response` with the encoded image, user query, and assistant prompt. This function interacts with the AI model and formats the resulting response using HTML.

6. **Render the result**
   The generated response is passed to the `index.html` template and displayed to the user. If no file is uploaded, an error message is flashed.

At this point, your `app.py` script should look like this in full:

```
import requests
import re
import base64
import os
from ibm_watsonx_ai import Credentials
from ibm_watsonx_ai import APIClient
from ibm_watsonx_ai.foundation_models import Model, ModelInference
from ibm_watsonx_ai.foundation_models.schema import TextChatParameters
from ibm_watsonx_ai.metanames import GenTextParamsMetaNames
from PIL import Image
from flask import Flask, render_template, request, redirect, url_for, flash
app = Flask(__name__)
credentials = Credentials(
                  url = "https://us-south.ml.cloud.ibm.com",
                  # api_key = "<YOUR_API_KEY>" # Normally you'd put an API key here, but we've got you covered here
                 )
client = APIClient(credentials)
model_id = "meta-llama/llama-4-maverick-17b-128e-instruct-fp8"
project_id = "skills-network"
params = TextChatParameters()
model = ModelInference(
    model_id=model_id,
    credentials=credentials,
    project_id=project_id,
    params=params
)
def input_image_setup(uploaded_file):
    """
    Encodes the uploaded image file into a base64 string to be used with AI models.
    Parameters:
    - uploaded_file: File-like object uploaded via a file uploader (Streamlit or other frameworks)
    Returns:
    - encoded_image (str): Base64 encoded string of the image data
    """
    # Check if a file has been uploaded
    if uploaded_file is not None:
        # Read the file into bytes
        bytes_data = uploaded_file.read()
        # Encode the image to a base64 string
        encoded_image = base64.b64encode(bytes_data).decode("utf-8")
        return encoded_image
    else:
```

```
        raise FileNotFoundError("No file uploaded")

def format_response(response_text):
    """
    Formats the model response to display each item on a new line as a list.
    Converts numbered items into HTML `<ul>` and `<li>` format.
    Adds additional HTML elements for better presentation of headings and separate sections.
    """
    # Replace section headers that are bolded with '**' to HTML paragraph tags with bold text
    response_text = re.sub(r"\*\*(.*?)\*\*", r"<p><strong>\1</strong></p>", response_text)
    # Convert bullet points denoted by "*" to HTML list items
    response_text = re.sub(r"(?m)^\s*\*\s(.*)", r"<li>\1</li>", response_text)
    # Wrap list items within <ul> tags for proper HTML structure and indentation
    response_text = re.sub(r"(<li>.*?</li>)+", lambda match: f"<ul>{match.group(0)}</ul>", response_text, flags=re.DOTALL)
    # Ensure that all paragraphs have a line break after them for better separation
    response_text = re.sub(r"</p>(?=<p>)", r"</p><br>", response_text)
    # Ensure the disclaimer and other distinct paragraphs have proper line breaks
    response_text = re.sub(r"(\n|\\n)+", r"<br>", response_text)
    return response_text
def generate_model_response(encoded_image, user_query, assistant_prompt):
    """
    Sends an image and a query to the model and retrieves the description or answer.
    Formats the response using HTML elements for better presentation.
    """
    # Create the messages object
    messages = [
        {
            "role": "user",
            "content": [
                {"type": "text", "text": assistant_prompt + "\n\n" + user_query},
                {"type": "image_url", "image_url": {"url": "" + encoded_image}}
            ]
        }
    ]
    try:
        # Send the request to the model
        response = model.chat(messages=messages)
        raw_response = response['choices'][0]['message']['content']
        # Format the raw response text using the format_response function
        formatted_response = format_response(raw_response)
        return formatted_response
    except Exception as e:
        print(f"Error in generating response: {e}")
        return "<p>An error occurred while generating the response.</p>"

@app.route("/", methods=["GET", "POST"])
def index():
    if request.method == "POST":
        # Retrieve user inputs
        user_query = request.form.get("user_query")
        uploaded_file = request.files.get("file")
        if uploaded_file:
            # Process the uploaded image
            encoded_image = input_image_setup(uploaded_file)
            if not encoded_image:
                flash("Error processing the image. Please try again.", "danger")
                return redirect(url_for("index"))
            # Assistant prompt (can be customized)
            assistant_prompt = """
            You are an expert nutritionist. Your task is to analyze the food items displayed in the image and provide a detailed nutritional asses
        1. **Identification**: List each identified food item clearly, one per line.
        2. **Portion Size & Calorie Estimation**: For each identified food item, specify the portion size and provide an estimated number of calor
        - **[Food Item]**: [Portion Size], [Number of Calories] calories
        Example:
        *   **Salmon**: 6 ounces, 210 calories
        *   **Asparagus**: 3 spears, 25 calories
        3. **Total Calories**: Provide the total number of calories for all food items.
        Example:
        Total Calories: [Number of Calories]
        4. **Nutrient Breakdown**: Include a breakdown of key nutrients such as **Protein**, **Carbohydrates**, **Fats**, **Vitamins**, and **Mine
        Example:
        *   **Protein**: Salmon (35g), Asparagus (3g), Tomatoes (1g) = [Total Protein]
        5. **Health Evaluation**: Evaluate the healthiness of the meal in one paragraph.
        6. **Disclaimer**: Include the following exact text as a disclaimer:
        The nutritional information and calorie estimates provided are approximate and are based on general food data.
        Actual values may vary depending on factors such as portion size, specific ingredients, preparation methods, and individual variations.
        For precise dietary advice or medical guidance, consult a qualified nutritionist or healthcare provider.
        Format your response exactly like the template above to ensure consistency.
        """
            # Generate the model's response
            response = generate_model_response(encoded_image, user_query, assistant_prompt)
            # Render the result
            return render_template("index.html", user_query=user_query, response=response)
        else:
            flash("Please upload an image file.", "danger")
            return redirect(url_for("index"))
    return render_template("index.html")
if __name__ == "__main__":
    app.run(debug=True)
```

Create a simple web interface for the application

Next, you will create a simple web interface for your application.

Open **index.html** in IDE

Update the content of `index.html` with:

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <title>AI Nutrition Coach</title>
    <link rel="stylesheet" href="{{ url_for('static', filename='style.css') }}">
    <link href="https://fonts.googleapis.com/css2?family=Roboto:wght@300;400;500;700&display=swap" rel="stylesheet">
    <script>
        // JavaScript function to display the loading message when the form is submitted
        function showLoading() {
            document.getElementById("loading-message").style.display = "block";
        }
        // JavaScript function to display the uploaded image
        function displayUploadedImage(event) {
            const fileInput = event.target;
            const imageContainer = document.getElementById('uploaded-image');
            const file = fileInput.files[0];
            if (file) {
                const reader = new FileReader();
                reader.onload = function(e) {
                    imageContainer.innerHTML = `<img src="${e.target.result}" alt="Uploaded Image" class="uploaded-image-preview">`;
                };
                reader.readAsDataURL(file);
            } else {
                imageContainer.innerHTML = "";  // Clear the container if no image is selected
            }
        }
    </script>
</head>
<body>
    <div class="container">
        <div class="header">
            <h1>AI Nutrition Coach</h1>
        </div>
        <form method="POST" enctype="multipart/form-data" class="form" onsubmit="showLoading()">
            <div class="input-group">
                <label for="user_query">Ask a question about the uploaded image:</label>
                <input type="text" name="user_query" id="user_query" value="How many calories are in this food?" required>
            </div>

            <div class="input-group">
                <label for="file">Choose an image:</label>
                <input type="file" name="file" id="file" accept="image/*" required onchange="displayUploadedImage(event)">
            </div>
            <button type="submit" class="btn-primary">Tell me the total calories</button>
        </form>
        <!-- Loading message -->
        <div id="loading-message" style="display: none;" class="loading">
            <p>Calculating, please wait...</p>
        </div>
        <!-- Div to display the uploaded image -->
        <div id="uploaded-image" style="margin-top: 20px; text-align: center;"></div>
        <!-- Flash messages for errors or notifications -->
        {% with messages = get_flashed_messages(with_categories=true) %}
            {% if messages %}
                <ul class="flashes">
                    {% for category, message in messages %}
                        <li class="{{ category }}">{{ message }}</li>
                    {% endfor %}
                </ul>
            {% endif %}
        {% endwith %}
        <!-- Display the model's response along with the image -->
        {% if response %}
            <div class="result">
                <h2>Here is a calorie breakdown of the food in this image:</h2>
                <div class="response-content">{{ response|safe }}</div>  <!-- Using safe to allow HTML content -->
            </div>
        {% endif %}
    </div>
</body>
</html>
```

This user-friendly interface lets users upload a food image and receive a detailed response with a calorie breakdown and personalized nutritional advice.

We'll include a basic stylesheet, `style.css`, to enhance the visual appeal of your web app.

Open **style.css** in IDE

Update `style.css` with:

```
body {
    font-family: 'Roboto', sans-serif;
    background: linear-gradient(135deg, #f0f4f7, #e2e8f0);
    color: #333;
    margin: 0;
    padding: 0;
}
.container {
    width: 90%;
    max-width: 700px;
    margin: 50px auto;
    background: #fff;
    padding: 30px;
    box-shadow: 0 4px 15px rgba(0, 0, 0, 0.1);
    border-radius: 12px;
    border-left: 6px solid #007BFF;
    border-right: 6px solid #007BFF;
}
.header h1 {
```

```
    text-align: center;
    font-weight: 700;
    color: #007BFF;
    margin-bottom: 20px;
}
.form {
    display: flex;
    flex-direction: column;
    gap: 20px;
}
.input-group {
    display: flex;
    flex-direction: column;
    gap: 5px;
}
label {
    font-weight: 500;
    color: #555;
}
input[type="text"],
input[type="file"] {
    padding: 10px;
    font-size: 1rem;
    border: 1px solid #ddd;
    border-radius: 4px;
    outline: none;
    transition: border-color 0.2s;
}
input[type="text"]:focus,
input[type="file"]:focus {
    border-color: #007BFF;
}
.btn-primary {
    padding: 12px;
    background: #007BFF;
    color: white;
    border: none;
    border-radius: 4px;
    font-weight: 500;
    cursor: pointer;
    font-size: 1rem;
    transition: background-color 0.3s;
}
.btn-primary:hover {
    background: #0056b3;
}
.result {
    margin-top: 30px;
    background: #f1f9ff;
    padding: 20px;
    border-radius: 8px;
    border: 1px solid #007BFF;
}
.result h2 {
    color: #007BFF;
}
.result div {
    color: #333;
    font-size: 1rem;
    line-height: 1.5;
}
.flashes {
    list-style-type: none;
    padding: 0;
}
.flashes li {
    margin: 10px 0;
    padding: 10px;
    border-radius: 4px;
    background: #f8d7da;
    color: #721c24;
    border: 1px solid #f5c6cb;
}
.result h2 {
    color: #007BFF;
    font-size: 1.5rem;
    margin-bottom: 15px;
}
.result .response-content ul {
    padding-left: 20px;
    margin-top: 0;
    margin-bottom: 0;
    list-style-type: disc;
}
.result .response-content li {
    margin-bottom: 5px;
}
.result .response-content h3 {
    color: #333;
    margin-top: 20px;
    font-size: 1.2rem;
}
.result .response-content p {
    margin: 10px 0;
}
.loading {
    margin-top: 20px;
    padding: 15px;
    background: #fef9e7;
    border: 1px solid #f9e79f;
    border-radius: 6px;
    text-align: center;
    color: #d35400;
    font-weight: 500;
}
.uploaded-image-preview {
    max-width: 50%;
    height: auto;
    border-radius: 8px;
    box-shadow: 0 4px 15px rgba(0, 0, 0, 0.1);
    margin-top: 20px;
}
```

Launch the application

NOTE: At any point throughout the project, if you encounter issues, click on Table of Contents and navigate to the solutions.

Return to the terminal and verify that the virtual environment `my_env` label appears at the start of the line. This means that you are in the `my_env` environment that you just created. Next, run the following command to run the Python script.

```
python app.py
```

After it runs successfully, you will see a message similar to following in the terminal:

Because the web application is hosted locally on port 5000, click on the following button to view the application we've developed.

Web Application

(Note: If the **Web Application** button does not work, follow the instructions in the following images to launch the app.)

Let's take a look at the web application. It shows an example of the calorie breakdown of a burger and some fries. You're encouraged to experiment with the web app's inputs and outputs!

Exercise 1

In this project, you used the **Llama 4 Maverick 17B 128E Instruct FP8** model as the core large language model (LLM) for handling multimodal tasks. However, there is another powerful and enterprise-ready multimodal LLM worth exploring: **IBM Granite 3.2 Vision**. Try replacing Llama 4 with Granite 3.2 Vision in your application and observe how the two models compare in terms of **performance, speed, reasoning quality, and visual understanding**. This side-by-side evaluation can offer valuable insights into choosing the right model for your specific use case.

▶ Click here for hints:

Exercise 2

Experiment with different settings for the model's parameters and assess how they impact the output. Adjust parameters like temperature, top_k, and max_tokens to observe variations in the responses. For more information about these parameters, refer to the documentation here.

▶ Click here for hints:

Solution for `app.py`

On this page, you will find the complete code for `app.py`. To update your code, click on the purple button below to open the corresponding file. If you encounter any issues, simply copy and paste the code into the `app.py` file.

Find solutions for the `index.html` file on next page.

Main logic of the app

Open **app.py** in IDE

```
import requests
import re
import base64
import os
from ibm_watsonx_ai import Credentials
from ibm_watsonx_ai import APIClient
from ibm_watsonx_ai.foundation_models import Model, ModelInference
from ibm_watsonx_ai.foundation_models.schema import TextChatParameters
from ibm_watsonx_ai.metanames import GenTextParamsMetaNames
from PIL import Image
from flask import Flask, render_template, request, redirect, url_for, flash
app = Flask(__name__)
credentials = Credentials(
                  url = "https://us-south.ml.cloud.ibm.com",
                  # api_key = "<YOUR_API_KEY>" # Normally you'd put an API key here, but we've got you covered here
                 )
client = APIClient(credentials)
model_id = "meta-llama/llama-4-maverick-17b-128e-instruct-fp8"
project_id = "skills-network"
params = TextChatParameters()
```

```
model = ModelInference(
    model_id=model_id,
    credentials=credentials,
    project_id=project_id,
    params=params
)
def input_image_setup(uploaded_file):
    """
    Encodes the uploaded image file into a base64 string to be used with AI models.
    Parameters:
    - uploaded_file: File-like object uploaded via a file uploader (Streamlit or other frameworks)
    Returns:
    - encoded_image (str): Base64 encoded string of the image data
    """
    # Check if a file has been uploaded
    if uploaded_file is not None:
        # Read the file into bytes
        bytes_data = uploaded_file.read()
        # Encode the image to a base64 string
        encoded_image = base64.b64encode(bytes_data).decode("utf-8")
        return encoded_image
    else:
        raise FileNotFoundError("No file uploaded")

def format_response(response_text):
    """
    Formats the model response to display each item on a new line as a list.
    Converts numbered items into HTML `<ul>` and `<li>` format.
    Adds additional HTML elements for better presentation of headings and separate sections.
    """
    # Replace section headers that are bolded with '**' to HTML paragraph tags with bold text
    response_text = re.sub(r"\*\*(.*?)\*\*", r"<p><strong>\1</strong></p>", response_text)
    # Convert bullet points denoted by "*" to HTML list items
    response_text = re.sub(r"(?m)^\s*\*\s(.*)", r"<li>\1</li>", response_text)
    # Wrap list items within <ul> tags for proper HTML structure and indentation
    response_text = re.sub(r"(<li>.*?</li>)+", lambda match: f"<ul>{match.group(0)}</ul>", response_text, flags=re.DOTALL)
    # Ensure that all paragraphs have a line break after them for better separation
    response_text = re.sub(r"</p>(?=<p>)", r"</p><br>", response_text)
    # Ensure the disclaimer and other distinct paragraphs have proper line breaks
    response_text = re.sub(r"(\n|\\n)+", r"<br>", response_text)
    return response_text
def generate_model_response(encoded_image, user_query, assistant_prompt):
    """
    Sends an image and a query to the model and retrieves the description or answer.
    Formats the response using HTML elements for better presentation.
    """
    # Create the messages object
    messages = [
        {
            "role": "user",
            "content": [
                {"type": "text", "text": assistant_prompt + "\n\n" + user_query},
                {"type": "image_url", "image_url": {"url": "" + encoded_image}}
            ]
        }
    ]
    try:
        # Send the request to the model
        response = model.chat(messages=messages)
        raw_response = response['choices'][0]['message']['content']
        # Format the raw response text using the format_response function
        formatted_response = format_response(raw_response)
        return formatted_response
    except Exception as e:
        print(f"Error in generating response: {e}")
        return "<p>An error occurred while generating the response.</p>"

@app.route("/", methods=["GET", "POST"])
def index():
    if request.method == "POST":
        # Retrieve user inputs
        user_query = request.form.get("user_query")
        uploaded_file = request.files.get("file")
        if uploaded_file:
            # Process the uploaded image
            encoded_image = input_image_setup(uploaded_file)
            if not encoded_image:
                flash("Error processing the image. Please try again.", "danger")
                return redirect(url_for("index"))
            # Assistant prompt (can be customized)
            assistant_prompt = """
            You are an expert nutritionist. Your task is to analyze the food items displayed in the image and provide a detailed nutritional asses
        1. **Identification**: List each identified food item clearly, one per line.
        2. **Portion Size & Calorie Estimation**: For each identified food item, specify the portion size and provide an estimated number of calor
        - **[Food Item]**: [Portion Size], [Number of Calories] calories
        Example:
        *   **Salmon**: 6 ounces, 210 calories
        *   **Asparagus**: 3 spears, 25 calories
        3. **Total Calories**: Provide the total number of calories for all food items.
        Example:
        Total Calories: [Number of Calories]
        4. **Nutrient Breakdown**: Include a breakdown of key nutrients such as **Protein**, **Carbohydrates**, **Fats**, **Vitamins**, and **Mine
        Example:
        *   **Protein**: Salmon (35g), Asparagus (3g), Tomatoes (1g) = [Total Protein]
        5. **Health Evaluation**: Evaluate the healthiness of the meal in one paragraph.
        6. **Disclaimer**: Include the following exact text as a disclaimer:
        The nutritional information and calorie estimates provided are approximate and are based on general food data.
        Actual values may vary depending on factors such as portion size, specific ingredients, preparation methods, and individual variations.
        For precise dietary advice or medical guidance, consult a qualified nutritionist or healthcare provider.
        Format your response exactly like the template above to ensure consistency.
        """
            # Generate the model's response
            response = generate_model_response(encoded_image, user_query, assistant_prompt)
            # Render the result
            return render_template("index.html", user_query=user_query, response=response)
        else:
            flash("Please upload an image file.", "danger")
            return redirect(url_for("index"))
    return render_template("index.html")
if __name__ == "__main__":
    app.run(debug=True)
```

Solution for `index.html` file

On this page, you will find the complete code for `index.html`. To update your code, click on the purple button below to open the corresponding file. If you encounter any issues, simply copy and paste the code into the `index.html` file.

Find solutions for the `style.css` file on next page.

Update `index.html`

Open **index.html** in IDE

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <title>AI Nutrition Coach</title>
    <link rel="stylesheet" href="{{ url_for('static', filename='style.css') }}">
    <link href="https://fonts.googleapis.com/css2?family=Roboto:wght@300;400;500;700&display=swap" rel="stylesheet">
    <script>
        // JavaScript function to display the loading message when the form is submitted
        function showLoading() {
            document.getElementById("loading-message").style.display = "block";
        }
        // JavaScript function to display the uploaded image
        function displayUploadedImage(event) {
            const fileInput = event.target;
            const imageContainer = document.getElementById('uploaded-image');
            const file = fileInput.files[0];
            if (file) {
                const reader = new FileReader();
                reader.onload = function(e) {
                    imageContainer.innerHTML = `<img src="${e.target.result}" alt="Uploaded Image" class="uploaded-image-preview">`;
                };
                reader.readAsDataURL(file);
            } else {
                imageContainer.innerHTML = "";  // Clear the container if no image is selected
            }
        }
    </script>
</head>
<body>
    <div class="container">
        <div class="header">
            <h1>AI Nutrition Coach</h1>
        </div>
        <form method="POST" enctype="multipart/form-data" class="form" onsubmit="showLoading()">
            <div class="input-group">
                <label for="user_query">Ask a question about the uploaded image:</label>
                <input type="text" name="user_query" id="user_query" value="How many calories are in this food?" required>
            </div>

            <div class="input-group">
                <label for="file">Choose an image:</label>
                <input type="file" name="file" id="file" accept="image/*" required onchange="displayUploadedImage(event)">
            </div>
            <button type="submit" class="btn-primary">Tell me the total calories</button>
        </form>
        <!-- Loading message -->
        <div id="loading-message" style="display: none;" class="loading">
            <p>Calculating, please wait...</p>
        </div>
        <!-- Div to display the uploaded image -->
        <div id="uploaded-image" style="margin-top: 20px; text-align: center;"></div>
        <!-- Flash messages for errors or notifications -->
        {% with messages = get_flashed_messages(with_categories=true) %}
            {% if messages %}
                <ul class="flashes">
                    {% for category, message in messages %}
                        <li class="{{ category }}">{{ message }}</li>
                    {% endfor %}
                </ul>
            {% endif %}
        {% endwith %}
        <!-- Display the model's response along with the image -->
        {% if response %}
            <div class="result">
                <h2>Here is a calorie breakdown of the food in this image:</h2>
                <div class="response-content">{{ response|safe }}</div>  <!-- Using safe to allow HTML content -->
            </div>
        {% endif %}
    </div>
</body>
</html>
```

Solution for `style.css` file

On this page, you will find the complete code for `style.css`. To update your code, click on the purple button below to open the corresponding file. If you encounter any issues, simply copy and paste the code into the `style.css` file.

Update `style.css`

Open **style.css** in IDE

```
body {
    font-family: 'Roboto', sans-serif;
    background: linear-gradient(135deg, #f0f4f7, #e2e8f0);
    color: #333;
    margin: 0;
    padding: 0;
}
.container {
    width: 90%;
    max-width: 700px;
    margin: 50px auto;
    background: #fff;
    padding: 30px;
    box-shadow: 0 4px 15px rgba(0, 0, 0, 0.1);
    border-radius: 12px;
    border-left: 6px solid #007BFF;
    border-right: 6px solid #007BFF;
}
.header h1 {
    text-align: center;
    font-weight: 700;
    color: #007BFF;
    margin-bottom: 20px;
}
.form {
    display: flex;
    flex-direction: column;
    gap: 20px;
}
.input-group {
    display: flex;
    flex-direction: column;
    gap: 5px;
}
label {
    font-weight: 500;
    color: #555;
}
input[type="text"],
input[type="file"] {
    padding: 10px;
    font-size: 1rem;
    border: 1px solid #ddd;
    border-radius: 4px;
    outline: none;
    transition: border-color 0.2s;
}
input[type="text"]:focus,
input[type="file"]:focus {
    border-color: #007BFF;
}
.btn-primary {
    padding: 12px;
    background: #007BFF;
    color: white;
    border: none;
    border-radius: 4px;
    font-weight: 500;
    cursor: pointer;
    font-size: 1rem;
    transition: background-color 0.3s;
}
.btn-primary:hover {
    background: #0056b3;
}
.result {
    margin-top: 30px;
    background: #f1f9ff;
    padding: 20px;
    border-radius: 8px;
    border: 1px solid #007BFF;
}
.result h2 {
    color: #007BFF;
}
.result div {
    color: #333;
    font-size: 1rem;
    line-height: 1.5;
}
.flashes {
    list-style-type: none;
    padding: 0;
}
.flashes li {
    margin: 10px 0;
    padding: 10px;
    border-radius: 4px;
    background: #f8d7da;
    color: #721c24;
    border: 1px solid #f5c6cb;
}
.result h2 {
    color: #007BFF;
    font-size: 1.5rem;
    margin-bottom: 15px;
}
.result .response-content ul {
    padding-left: 20px;
```

```
        margin-top: 0;
        margin-bottom: 0;
        list-style-type: disc;
    }
    .result .response-content li {
        margin-bottom: 5px;
    }
    .result .response-content h3 {
        color: #333;
        margin-top: 20px;
        font-size: 1.2rem;
    }
    .result .response-content p {
        margin: 10px 0;
    }
    .loading {
        margin-top: 20px;
        padding: 15px;
        background: #fef9e7;
        border: 1px solid #f9e79f;
        border-radius: 6px;
        text-align: center;
        color: #d35400;
        font-weight: 500;
    }
    .uploaded-image-preview {
        max-width: 50%;
        height: auto;
        border-radius: 8px;
        box-shadow: 0 4px 15px rgba(0, 0, 0, 0.1);
        margin-top: 20px;
    }
```

Conclusion and next steps

Conclusion

Congratulations on completing the AI Calorie Coach project! By building this application, you've successfully explored the powerful capabilities of the Llama 4 Maverick 17B 128E Instruct FP8 model and integrated it with AI-driven technologies to deliver practical, real-world solutions for dietary management. You've gained hands-on experience in encoding images, constructing effective prompts for generative models, and presenting nutritional data in a user-friendly format.

This project demonstrates how cutting-edge AI can simplify complex tasks like calorie estimation and nutrition assessment, offering tech-savvy health enthusiasts and dietetics professionals an innovative tool to promote healthier lifestyle choices. As AI continues to advance, your understanding of multimodal models and their real-world applications will be invaluable in leveraging technology for positive impacts on everyday life.

Feel free to expand on this foundation by exploring more advanced features such as personalized meal planning or real-time food tracking to further enhance the app's capabilities. Thank you for participating in this journey, and continue to explore the transformative potential of AI in health and wellness!

Next steps for learners

To build on the skills developed in this project and enhance your expertise in AI-powered applications, consider the following steps:

1. **Explore advanced prompt engineering**
   Experiment with more complex prompts to improve the quality and accuracy of AI-generated responses.

2. **Integrate additional AI models**
   Try incorporating other multimodal AI models available on watsonx.ai to compare their performance with the Llama 4 Maverick 17B 128E Instruct FP8 model.

3. **Add personalization features**
   Enhance the application by incorporating user preferences or dietary restrictions. This could involve creating user profiles or integrating machine learning to tailor nutritional advice.

4. **Expand nutritional analysis**
   Extend the application to include features like meal planning, nutrient tracking over time, or suggestions for healthy recipes. Integrate external APIs or databases to enrich the app's functionality.

5. **Optimize for mobile and accessibility**
   Consider developing a mobile-friendly version of your app or adding features to improve accessibility, such as voice recognition for queries or text-to-speech functionality.

6. **Learn about AI ethics and safety**
   Deepen your understanding of responsible AI use, focusing on topics like bias, privacy, and ethical considerations in health-related AI applications. Implement safeguards to protect user data.

These steps will not only solidify your understanding of AI but also open up new avenues to create impactful and innovative applications.

Further learning

- Explore the IBM watsonx.ai documentation for more advanced features.
- Learn about prompt engineering techniques to improve AI model outputs.

HAPPY LEARNING!

Author(s)

Hailey Quach

Other Contributor(s)

Ricky Shi