Beginner's Guide: Running Large Language Models Locally

Running large language models (LLMs) locally provides significant advantages, whether for research, experimentation, or building advanced applications. However, the process of configuring the necessary environment and setting up LLMs on local systems is often complex and resource intensive. This guide will walk you through setting up Ollama, a tool wrapped around llama.cpp, which simplifies using LLMs locally with an easy-to-use interface.

Why Use Local LLMs?

- Privacy: Keep your data secure by running models locally.
- Speed: No cloud dependencies mean faster responses.
- Customization: Fine-tune models for your specific needs.
- Cost Efficiency: Save on cloud service fees.
- Offline Access: Use models without an internet connection.
- Hardware Optimization: Efficiently use your local system resources.

Steps to Get Started

1. Prerequisites

Before starting, ensure you have the following:
- A computer (GPU support is optional but recommended for faster processing).
- Python 3.8 or later installed.
- Enough disk space and RAM for model files.

2. Download Ollama

Ollama works on macOS, Windows, and Linux. Choose one of these options to download it:
- Visit the Ollama GitHub page for installation instructions: https://github.com/ollama
- Go to Ollama's official website and download the installer for macOS or Windows: https://ollama.com

3. Install Ollama

Use Python's package manager, pip, to install Ollama. Open a terminal (or command prompt) and run:

```
pip install ollama
```

💡 Tip: Use a virtual environment like Miniconda for better package management.

4. Download Pre-Trained Models

Ollama gives you access to various pre-trained LLMs. To download a model, run:

ollama pull <model-name>

Example: To download the Llama 3.2 model with 1 billion parameters:

ollama pull llama3.2:1b

You can explore and download other models from the Ollama Model Hub:
https://ollama.com/search

5. Optional: Start the Ollama Server

To enable local interaction with the model, start the Ollama server:

ollama serve

6. Query the Model

Ask the model questions using the ollama query command. For example:

ollama query llama3.2 "What are the benefits of using local LLMs?"

Skip Step 5 and 6 , if you only want to download and use models in specific tools or frameworks.

6. Local Inference – Use case

We present a case example that showcases a practical approach to **local inference** with LLMs, tailored for sentiment analysis tasks. It efficiently processes text data, generates sentiment predictions with detailed reasoning, and evaluates model performance using key metrics like precision and accuracy. You can adapt this example for your own use cases, such as text classification, content analysis, or other NLP applications requiring local, secure, and scalable processing. For details, refer to the provided script **"2-Efficient-Local-Inference.py**

Additional Commands

- List Models:

ollama list

- Stop the Server:

ollama stop

- Delete a Model:

ollama delete <model-name>

Tips for Beginners

- Experiment: Try different models to see what fits your needs.
- Extend with Tools: Use frameworks like LangChain to build pipelines or advanced applications.
- Keep Updated: Regularly update Ollama and your models for the latest features and security.

Use Cases

- Summarization: Condense articles, papers, or documents.
- Chatbots: Create conversational agents without relying on cloud services.
- Secure Data Analysis: Analyze sensitive or proprietary data locally.

Code walkthrough of 2-Efficient-Local-Inference.py

The script reads text data from a CSV file, uses a language model to assess the sentiment of each text entry, and then processes and saves the results to a new CSV file. It also tracks the CO_2 emissions of the computation using the `codecarbon` library.

Importing Necessary Libraries

```
python
Copy code
import pandas as pd
import re
import time
import os
import ollama
from tqdm import tqdm
from codecarbon import EmissionsTracker
```

- **pandas**: Used for data manipulation and analysis, particularly for reading and writing CSV files.
- **re**: Provides regular expression matching operations for string processing.
- **time**: Used for time-related functions, such as tracking execution time.
- **os**: Allows interaction with the operating system, such as checking if a file exists.
- **ollama**: Used to interact with the language model for sentiment analysis.
- **tqdm**: Provides a progress bar for loops, making it easier to track processing progress.
- **codecarbon**: Tracks the carbon emissions of the code execution for sustainability metrics.

Setting the Model Name

```
python
Copy code
model_name = 'llama3.2'
```

- **model_name**: Specifies the name of the language model to be used for sentiment analysis.

Function: `label_and_reason_sentiment(content)`

```
python
Copy code
def label_and_reason_sentiment(content):
    full_prompt = f"""..."""
    try:
        response = ollama.chat(model=model_name, messages=[
            {'role': 'user', 'content': full_prompt}
```

```
        ])
        return response['message']['content']
    except Exception as e:
        print(f"An error occurred: {e}")
        return None
```

Purpose

- This function takes a text **content** as input and constructs a prompt to send to the language model.
- It requests the model to assess the sentiment of the text and provide reasoning.

How It Works

1. **Constructing the Prompt**:
   - The prompt includes instructions for the model to identify sentiment indicators and provide three separate assessments (Positive, Negative, or Neutral) along with concise reasoning.
2. **Interacting with the Language Model**:
   - Uses the `ollama.chat` function to send the prompt to the model and receive a response.
   - The model processes the prompt and returns its sentiment analysis.
3. **Error Handling**:
   - If an exception occurs (e.g., network issues, model errors), it prints the error message and returns `None`.

Function: `convert_analysis_to_dataframe(row, analysis_str)`

```
python
Copy code
def convert_analysis_to_dataframe(row, analysis_str):
    results = row.to_dict()
    analysis_str = analysis_str.replace('\n', ' ').replace('\r', '
').strip()
    pattern =
r'\d+\.\s*(Positive|Negative|Neutral)\s+(.*?)(?=(?:\d+\.\s*(?:Positive|
Negative|Neutral)|$))'
    matches = re.findall(pattern, analysis_str, flags=re.IGNORECASE)
    label_count = {'Positive': 0, 'Negative': 0, 'Neutral': 0}
    reasonings = []
    for label, reasoning in matches:
        label_capitalized = label.capitalize()
        if label_capitalized in label_count:
            label_count[label_capitalized] += 1
            reasonings.append(f"{label_capitalized}:
{reasoning.strip()}")
    max_count = max(label_count.values())
    if max_count == 0:
        print(f"No label assigned for row {row.name}")
        return None
```

```
    labels_with_max_count = [label for label, count in
label_count.items() if count == max_count]
    assigned_label = labels_with_max_count[0]
    results["predicted"] = assigned_label
    results["reasoning"] = " | ".join(reasonings)
    return results
```

Purpose

- Processes the analysis string returned by the language model.
- Extracts sentiment labels and reasonings.
- Determines the most frequent sentiment label.
- Returns a dictionary with the original data and the new sentiment analysis results.

How It Works

1. **Initial Setup**:
    - Converts the input row to a dictionary for easy manipulation.
    - Cleans up the `analysis_str` by removing newlines and extra spaces.
2. **Pattern Matching**:
    - Uses a regular expression to find all sentiment assessments in the analysis string.
    - The pattern looks for lines starting with a number followed by a sentiment label and reasoning.
3. **Counting Labels and Collecting Reasonings**:
    - Initializes a label count dictionary to keep track of how many times each sentiment appears.
    - Iterates over the matches to count labels and collect reasonings.
4. **Determining the Assigned Label**:
    - Finds the label(s) with the highest count.
    - If there's a tie, selects the first label.
    - Assigns the label to the `predicted` field in the results.
5. **Adding Reasonings**:
    - Joins all reasonings into a single string separated by " | ".
    - Adds the reasoning to the results.
6. **Returning Results**:
    - Returns the updated results dictionary.

Function: `process_csv(batch_size=10)`

```
python
Copy code
def process_csv(batch_size=10):
    # file_path = "data/dataset-sentiment.csv" # use this for full data
    file_path = "data/sample.csv" # Replace with your data CSV
    required_columns = ['text', 'label']
    try:
        df = pd.read_csv(file_path)
```

```
        present_columns = [col for col in required_columns if col in
df.columns]
        if len(present_columns) < 2:
            print(f"Warning: Required columns not found in
{file_path}")
            return
        data = df[present_columns]
    except Exception as e:
        print(f"Error reading file {file_path}: {e}")
        return
    processed_file = f"{model_name}.csv"
    processed_texts = set()
    if os.path.exists(processed_file):
        processed_data = pd.read_csv(processed_file)
        processed_texts = set(processed_data['text'].values)
    print("Processed texts: ", len(processed_texts))
    data = data[~data['text'].isin(processed_texts)]
    results_list = []
    tracker = EmissionsTracker()
    tracker.start()
    try:
        for index, row in tqdm(data.iterrows(), total=len(data),
desc='Processing'):
            content = row.get('text', '').replace('\n', '
').replace('\r', ' ').strip()
            if content:
                start_time = time.time()
                analysis_result = label_and_reason_sentiment(content)
                if analysis_result:
                    result = convert_analysis_to_dataframe(row,
analysis_result)
                    if result:
                        results_list.append(result)
                        print(f"Reasoning for row {index}:
{result['reasoning']}")
                else:
                    print(f"Analysis failed for row {index}")
                elapsed_time = time.time() - start_time
                print(f"Processed row index: {index}, Time taken:
{elapsed_time:.2f} seconds.")
            if len(results_list) >= batch_size:
                df_results = pd.DataFrame(results_list)
                columns_to_save = ['text', 'label', 'predicted',
'reasoning']
                df_results = df_results[columns_to_save]
                header = not os.path.exists(processed_file)
                df_results.to_csv(processed_file, mode='a',
header=header, index=False)
                results_list.clear()
    except KeyboardInterrupt:
        print("\nProcess interrupted! Saving current progress...")
    finally:
        if results_list:
            df_results = pd.DataFrame(results_list)
            columns_to_save = ['text', 'label', 'predicted',
'reasoning']
            df_results = df_results[columns_to_save]
```

```
            header = not os.path.exists(processed_file)
            df_results.to_csv(processed_file, mode='a', header=header,
index=False)
        emissions = tracker.stop()
        print(f"Final results saved to {processed_file}.")
        print(f"Estimated CO₂ emissions: {emissions:.6f} kg")
```

Purpose

- Processes the input CSV file containing text data and sentiment labels.
- Performs sentiment analysis on each text entry.
- Saves the processed results to a new CSV file in batches.
- Tracks CO_2 emissions during processing.

How It Works

1. **File Paths and Required Columns**:
   - Specifies the path to the input CSV file (`file_path`).
   - Defines the required columns: `'text'` and `'label'`.
2. **Reading the Input CSV File**:
   - Attempts to read the CSV file using `pandas`.
   - Checks if the required columns are present.
   - If not, prints a warning and exits the function.
3. **Handling Already Processed Data**:
   - Checks if a processed file already exists.
   - If so, reads the processed data and creates a set of texts that have already been processed.
   - Filters out these texts from the data to avoid re-processing.
4. **Initializing Variables**:
   - `results_list`: A list to store processing results.
   - `EmissionsTracker`: Starts tracking CO_2 emissions.
5. **Processing the Data**:
   - Iterates over each row in the data using `tqdm` for progress tracking.
   - Cleans the content by removing newlines and extra spaces.
   - For each content:
     - Records the start time.
     - Calls `label_and_reason_sentiment` to get the sentiment analysis from the model.
     - If successful, processes the analysis result using `convert_analysis_to_dataframe`.
     - Appends the result to `results_list`.
     - Prints the reasoning and processing time.
   - After processing `batch_size` entries, saves the results to the processed file and clears `results_list`.
6. **Handling Interruptions**:
   - If a `KeyboardInterrupt` occurs (e.g., the user stops the script), catches the exception.
   - Saves any remaining results before exiting.
7. **Finalizing and Saving Results**:

- After processing all data or upon interruption, checks if there are any remaining results to save.
- Stops the CO_2 emissions tracker and prints the estimated emissions.
- Notifies the user that the final results have been saved.

Main Execution

```
python
Copy code
if __name__ == "__main__":
    process_csv()
```

- Checks if the script is being run directly (not imported as a module).
- Calls the `process_csv` function to start processing.

Additional Notes

- **Batch Processing**:
  - The script processes data in batches defined by `batch_size` to manage memory usage and periodically save progress.
- **CO_2 Emissions Tracking**:
  - The `codecarbon` library tracks the environmental impact of the script by estimating CO_2 emissions.
- **Error Handling**:
  - The script includes try-except blocks to handle exceptions gracefully and provide informative messages.
- **Progress Tracking**:
  - Uses `tqdm` to display a progress bar, making it easier to monitor long-running processes.
- **Data Persistence**:
  - The script checks for existing processed data to avoid duplicate processing and ensures that progress is saved even if the script is interrupted.

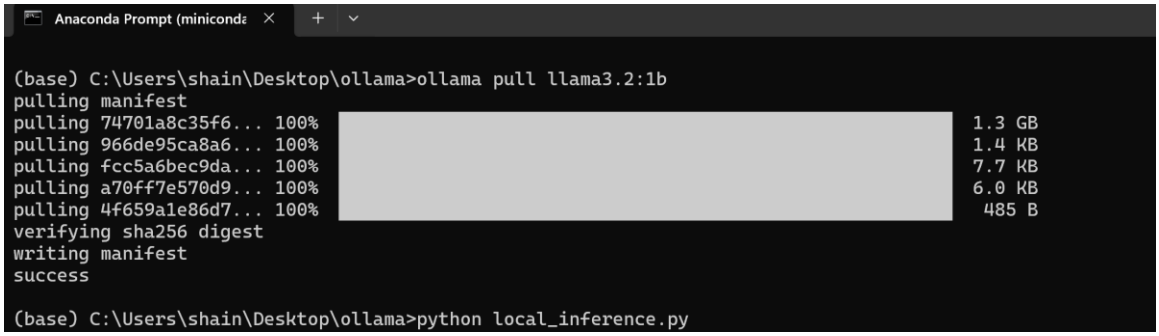
Summary

This script automates the process of sentiment analysis on text data using a language model. It reads data from a CSV file, interacts with the model to get sentiment assessments, processes the results to extract labels and reasonings, and saves the output to a new CSV file. The inclusion of CO_2 emissions tracking and robust error handling makes the script efficient and environmentally conscious.

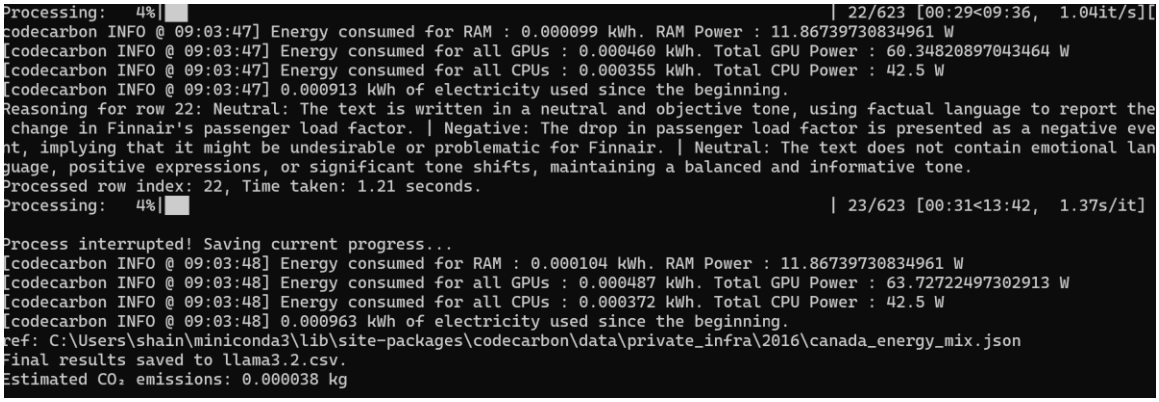
Feel free to run the script and observe the output files to gain a practical understanding of its functionality.

How to run in python.

You should run **2-Efficient-Local-Inference.py**



Expected output.



Prepared By

- **Name**: **Shaina Raza, PhD** shaina.raza@vectorinstitute.ai
- **Affiliation**: Vector Institute for Artificial Intelligence

This tutorial is presented for practical use based on existing methods and open resources (llama.cpp, ollama) for efficient evaluation and optimization of LLMs. It incorporates strategies for reducing carbon emissions and computational costs while highlighting use cases for secure local inference. Special thanks to the llama.cpp, ollama contributors whose work inspired this adaptation.