Few-Shot Learning and In-Context Learning Tutorial

Overview

This tutorial explores the cutting-edge techniques of Few-Shot Learning and In-Context Learning using OpenAI's GPT models and the LangChain library. These methods enable AI models to perform complex tasks with minimal examples, revolutionizing the way we approach machine learning problems.

Motivation

Traditional machine learning often requires large datasets for training, which can be time-consuming and resource-intensive. Few-Shot Learning and In-Context Learning address this limitation by leveraging the power of large language models to perform tasks with just a handful of examples. This approach is particularly valuable in scenarios where labeled data is scarce or expensive to obtain.

Key Components

1. **OpenAI's GPT Models**: State-of-the-art language models that serve as the foundation for our learning techniques.
2. **LangChain Library**: A powerful tool that simplifies the process of working with large language models.
3. **PromptTemplate**: A structured way to format inputs for the language model.
4. **LLMChain**: Manages the interaction between the prompt and the language model.

Method Details

1. Basic Few-Shot Learning

- Implementation of a sentiment classification task using few-shot learning.
- Demonstration of how to structure a prompt with examples for the model to learn from.
- Explanation of how the model generalizes from these examples to new inputs.

2. Advanced Few-Shot Techniques

- Exploration of multi-task learning for sentiment analysis and language detection.
- Discussion on how to design prompts that enable a single model to perform

multiple related tasks.
- Insights into the benefits of this approach, such as improved efficiency and better generalization.

3. In-Context Learning

- Demonstration of in-context learning for a custom task (e.g., text transformation).
- Explanation of how models can adapt to new tasks based solely on examples provided in the prompt.
- Discussion on the flexibility and limitations of this approach.

4. Best Practices and Evaluation

- Guidelines for selecting effective examples for few-shot learning.
- Techniques for prompt engineering to optimize model performance.
- Implementation of an evaluation framework to assess model accuracy.
- Discussion on the importance of diverse test cases and appropriate metrics.

Conclusion

Few-Shot Learning and In-Context Learning represent a significant advancement in the field of artificial intelligence. By enabling models to perform complex tasks with minimal examples, these techniques open up new possibilities for AI applications in areas where data is limited. This tutorial provides a solid foundation for understanding and implementing these powerful methods, equipping learners with the tools to leverage large language models effectively in their own projects.

As the field continues to evolve, mastering these techniques will be crucial for AI practitioners looking to stay at the forefront of natural language processing and machine learning.

In [7]:
```
import os
from dotenv import load_dotenv
from langchain_openai import ChatOpenAI
from langchain.prompts import PromptTemplate
from langchain.chains import LLMChain

load_dotenv()
os.environ["OPENAI_API_KEY"] = os.getenv('OPENAI_API_KEY') # OpenAI API

llm = ChatOpenAI(model="gpt-4o-mini", temperature=0)
print("Setup complete.")
```

```
Setup complete.
```

Basic Few-Shot Learning

We'll implement a basic few-shot learning scenario for sentiment classification.

Sentiment Classification:

- Definition: Determining the emotional tone behind a series of words.
- Applications: Customer service, market research, social media analysis.

Few-Shot Learning Approach:

1. Provide a small set of labeled examples (3 in this case).
2. Structure the prompt to clearly present examples and the new input.
3. Leverage the pre-trained knowledge of the language model.

Key Components:

- PromptTemplate: Structures the input for the model.
- LLMChain: Manages the interaction between the prompt and the language model.

In [18]:

```
def few_shot_sentiment_classification(input_text):
    few_shot_prompt = PromptTemplate(
        input_variables=["input_text"],
        template="""
        Classify the sentiment as Positive, Negative, or Neutral.

        Examples:
        Text: I love this product! It's amazing.
        Sentiment: Positive

        Text: This movie was terrible. I hated it.
        Sentiment: Negative

        Text: The weather today is okay.
        Sentiment: Neutral

        Now, classify the following:
        Text: {input_text}
        Sentiment:
        """
    )

    chain = few_shot_prompt | llm
    result = chain.invoke(input_text).content

    # Clean up the result
    result = result.strip()
    # Extract only the sentiment label
    if ':' in result:
        result = result.split(':')[1].strip()

    return result  # This will now return just "Positive", "Negative", o

test_text = "I can't believe how great this new restaurant is!"
result = few_shot_sentiment_classification(test_text)
print(f"Input: {test_text}")
print(f"Predicted Sentiment: {result}")
```

```
Input: I can't believe how great this new restaurant is!
Predicted Sentiment: Positive
```

Advanced Few-Shot Techniques

We'll now explore multi-task learning for sentiment analysis and language detection.

Multi-task Learning:

- Definition: Training a model to perform multiple related tasks simultaneously.
- Benefits: Improved efficiency, better generalization, reduced overfitting.

Implementation:

1. Design a prompt template that includes examples for multiple tasks.
2. Use task-specific instructions to guide the model's behavior.
3. Demonstrate how the same model can switch between tasks based on input.

In [10]:

```python
def multi_task_few_shot(input_text, task):
    few_shot_prompt = PromptTemplate(
        input_variables=["input_text", "task"],
        template="""
        Perform the specified task on the given text.

        Examples:
        Text: I love this product! It's amazing.
        Task: sentiment
        Result: Positive

        Text: Bonjour, comment allez-vous?
        Task: language
        Result: French

        Now, perform the following task:
        Text: {input_text}
        Task: {task}
        Result:
        """
    )

    chain = few_shot_prompt | llm
    return chain.invoke({"input_text": input_text, "task": task}).conten

print(multi_task_few_shot("I can't believe how great this is!", "sentime
print(multi_task_few_shot("Guten Tag, wie geht es Ihnen?", "language"))
```

```
Positive
Result: German
```

In-Context Learning

In-Context Learning allows models to adapt to new tasks based on examples provided in the prompt.

Key Aspects:

1. No fine-tuning required: The model learns from examples in the prompt.
2. Flexibility: Can be applied to a wide range of tasks.
3. Prompt engineering: Careful design of prompts is crucial for performance.

Example Implementation: We'll demonstrate in-context learning for a custom task (converting text to pig latin).

In [11]:
```
def in_context_learning(task_description, examples, input_text):
    example_text = "".join([f"Input: {e['input']}\nOutput: {e['output']}

    in_context_prompt = PromptTemplate(
        input_variables=["task_description", "examples", "input_text"],
        template="""
        Task: {task_description}

        Examples:
        {examples}

        Now, perform the task on the following input:
        Input: {input_text}
        Output:
        """
    )

    chain = in_context_prompt | llm
    return chain.invoke({"task_description": task_description, "examples

task_desc = "Convert the given text to pig latin."
examples = [
    {"input": "hello", "output": "ellohay"},
    {"input": "apple", "output": "appleay"}
]
test_input = "python"

result = in_context_learning(task_desc, examples, test_input)
print(f"Input: {test_input}")
print(f"Output: {result}")
```

```
Input: python
Output: Output: ythonpay
```

Best Practices and Evaluation

To maximize the effectiveness of few-shot and in-context learning:

1. Example Selection:
    - Diversity: Cover different aspects of the task.
    - Clarity: Use unambiguous examples.
    - Relevance: Choose examples similar to expected inputs.
    - Balance: Ensure equal representation of classes/categories.
    - Edge cases: Include examples of unusual or difficult cases.

2. Prompt Engineering:

    - Clear instructions: Specify the task explicitly.
    - Consistent format: Maintain a uniform structure for examples and inputs.
    - Conciseness: Avoid unnecessary information that may confuse the model.

3. Evaluation:

    - Create a diverse test set.
    - Compare model predictions to true labels.
    - Use appropriate metrics (e.g., accuracy, F1 score) based on the task.

In [20]:

```
def evaluate_model(model_func, test_cases):
    '''
    Evaluate the model on a set of test cases.

    Args:
    model_func: The function that makes predictions.
    test_cases: A list of dictionaries, where each dictionary contains a

    Returns:
    The accuracy of the model on the test cases.
    '''
    correct = 0
    total = len(test_cases)

    for case in test_cases:
        input_text = case['input']
        true_label = case['label']
        prediction = model_func(input_text).strip()

        is_correct = prediction.lower() == true_label.lower()
        correct += int(is_correct)

        print(f"Input: {input_text}")
        print(f"Predicted: {prediction}")
        print(f"Actual: {true_label}")
        print(f"Correct: {is_correct}\n")

    accuracy = correct / total
    return accuracy

test_cases = [
    {"input": "This product exceeded my expectations!", "label": "Positi
    {"input": "I'm utterly disappointed with the service.", "label": "Ne
    {"input": "The temperature today is 72 degrees.", "label": "Neutral"
]

accuracy = evaluate_model(few_shot_sentiment_classification, test_cases)
print(f"Model Accuracy: {accuracy:.2f}")
```

```
Input: This product exceeded my expectations!
Predicted: Positive
Actual: Positive
Correct: True

Input: I'm utterly disappointed with the service.
Predicted: Negative
Actual: Negative
Correct: True

Input: The temperature today is 72 degrees.
Predicted: Neutral
Actual: Neutral
Correct: True

Model Accuracy: 1.00
```