Application or use cases of various design patterns - Software design

A project by Sreeneha Samudrala Snighdha,
Franz Josef Diaz,
Maralgul Korpeyeva

Description of the Project

The main focus of this application is to execute a sentiment analysis of texts. It is designed to analyze an input text to classify if it belongs to either a positive, negative or a neutral sentiment. This was done by using a pre-trained model and a tokenizer along with a classification model. This is a baseline solution in handling large amounts of feedback may it be from businesses, movie reviews, or public opinions.

Source Code

The detailed explanation of the implementation of our code is explained below. You can also find the code on our Github: https://github.com/neha04-12/Design-patterns-Task3

```
# Strategy Interface
# Define an abstract Strategy interface for sentiment analysis
class SentimentStrategy(ABC):
   @abstractmethod
   def analyze(self, text: str) -> str:
       pass # Subclasses will implement this method
```

This block of code defines the class `SentimentStrategy` that serves as a backbone for creating strategies specific to models that are to be implemented. This class will also be inherited in the chosen model along with its methods and attributes. It allows the sentiment analysis logic to be interchangeable depending on the model used.

```
# Local transformer model (RoBERTa - HuggingFace transformer model)
class TransformerSentimentStrategy(SentimentStrategy):
   def __init__(self, model, tokenizer, labels):
       self.model = model
       self.tokenizer = tokenizer
       self.labels = labels

   def analyze(self, text: str) -> str:
       encoded_input = self.tokenizer(text, return_tensors='pt') # Encoding the
text into model input format
       output = self.model(**encoded_input) # Running inference with the model
       scores = output[0][0].detach().numpy() # Getting the raw scores and
converting to numpy
       scores = softmax(scores) # Converting scores to probabilities using
softmax
       ranking = np.argsort(scores) # Getting the indices of scores sorted in
descending order
       ranking = ranking[::-1]
       l = labels[ranking[0]]
       s = np.round(float(scores[ranking[0]]), 4)
       return l, s # Returning label and score
```

This block of code defines the strategy `TransformerSentimentStrategy` that uses the pre-trained model roberta from the huggingface to predict the sentiment of the given input. With the use of a pre-trained RoBERTa model from huggingface, in the analyze method, the text is first tokenized to feed as a compatible format for the model. It then proceeds in inference and then the model returns the predicted sentiment label along with its score.

```
# Simulated External API (using HuggingFace's external API for inference)
class APISentimentStrategy(SentimentStrategy):
   def __init__(self, hf_token, API_URL, headers):
       self.hf_token = hf_token
       self.API_URL = API_URL
       self.headers = headers

   def analyze(self, text: str) -> str:
       payload = dict(inputs=text, options=dict(wait_for_model=True)) #
Preparing the payload to send
       response = requests.post(self.API_URL, headers=self.headers,
json=payload) # Sending a POST request to the API
       sentiment_result = response.json()[0] # Parsing the JSON response
       top_sentiment = max(sentiment_result, key=lambda x: x['score']) #
Getting the sentiment with the higher score
       l = top_sentiment['label']
       s = top_sentiment['score']
       return l, s  # Returning label and score
```

This block of code defines the class `APISentimentStrategy` that uses the API to perform the classification based on the model defined. It also returns the response from the API which is the label and score. This allows the application to depend on an external service for the inference process.

```
# Circuit breaker pattern that falls back to a secondary strategy on failure
# Main one is the External API
# Back up one is the Roberta locl model
class CircuitBreakerStrategy(SentimentStrategy):
   def __init__(self, primary_strategy, fallback_strategy,
failure_threshold=1):
       self.primary = primary_strategy # Main strategy to try first
       self.fallback = fallback_strategy # Fallback if main fails
       self.failure_threshold = failure_threshold # Number of allowed failures
       self.failure_count = 0 # Counter to track failures

   def analyze(self, text: str) -> str:
       try:
           if self.failure_count >= self.failure_threshold:
               raise Exception("Circuit open") # Simulating circuit breaking
           result = self.primary.analyze(text) # Trying main strategy
           self.failure_count = 0 # Resetting failure count on success
           return result
       except:
           self.failure_count += 1
           return self.fallback.analyze(text) # Using fallback strategy
```

This block of code defines the class `CircuitBreakerStrategy` which initializes the choses primary strategy and fallback strategy based on the values sent to the model. It also monitors the threshold at which the strategies should be switched.

```
# Template Method - defining the high-level steps of text analysis
class SentimentAnalyzer(ABC):
   def analyze_text(self, texts):
       sentiments = [self.run_sentiment_model(text) for text in texts] #
Running the sentiment model on each tweet
       return self.aggregate_results(sentiments) # Printing aggregated results

   @abstractmethod
   def run_sentiment_model(self, text): # Subclasses implement this to choose
the strategy
     pass

   def aggregate_results(self, sentiments):
     results = []
     for sentiment, score in sentiments:  # Looping through results and print
them
       print(f"Sentiment: {sentiment}, Score: {round(score, 4)}")
```

This block of code is the pipeline on how an input text is sent into the chosen model along with how the predicted results are aggregated to display.

```
# Final Analyzer using API with fallback to transformer
class CustomTextAnalyzer(SentimentAnalyzer):
   def __init__(self, tokenizer, model, labels, hf_token, API_URL, headers):
       self.strategy = CircuitBreakerStrategy(
           primary_strategy=APISentimentStrategy(hf_token, API_URL, headers),
           fallback_strategy=TransformerSentimentStrategy(model, tokenizer,
labels)
       )

   def run_sentiment_model(self, text):
       return self.strategy.analyze(text) # Using the strategy to analyze a
tweet
```

This block of code defines the API Strategy as the primary strategy and the transformer strategy as the fallback strategy. It also calls the analyze function that predicts the sentiment of the input text based on the appropriate strategy.

```
# Define the task and load the pre-trained model and tokenizer
task='sentiment'
MODEL = f"cardiffnlp/twitter-roberta-base-{task}"  # Model name
tokenizer = AutoTokenizer.from_pretrained(MODEL)  # Loading tokenizer

# Loading sentiment labels from GitHub mapping file
labels=[]
mapping_link =
f"https://raw.githubusercontent.com/cardiffnlp/tweeteval/main/datasets/{task}/m
apping.txt"
with urllib.request.urlopen(mapping_link) as f:
   html = f.read().decode('utf-8').split("\n") # Reading and splitting by line
   csvreader = csv.reader(html, delimiter='\t')  # Parsing as TSV
labels = [row[1] for row in csvreader if len(row) > 1] # Extracting  labels

model = AutoModelForSequenceClassification.from_pretrained(MODEL) # Loading
actual sentiment classification model
model.save_pretrained(MODEL) # Optionally saving locally
```

This code defines all the model details and requirements for the roberta model from huggingface, tokenizers and mapping labels that have to be sent to the model for prediction.

```
# API inferencing - Setting up the HuggingFace API access
model2 = "cardiffnlp/twitter-roberta-base-sentiment-latest" # Updated version
of the model
```

```
hf_token = "..." # Input your API token here
API_URL = "https://api-inference.huggingface.co/models/" + model2
headers = {"Authorization": "Bearer %s" % (hf_token)}  # Formatting the
authorization header
```

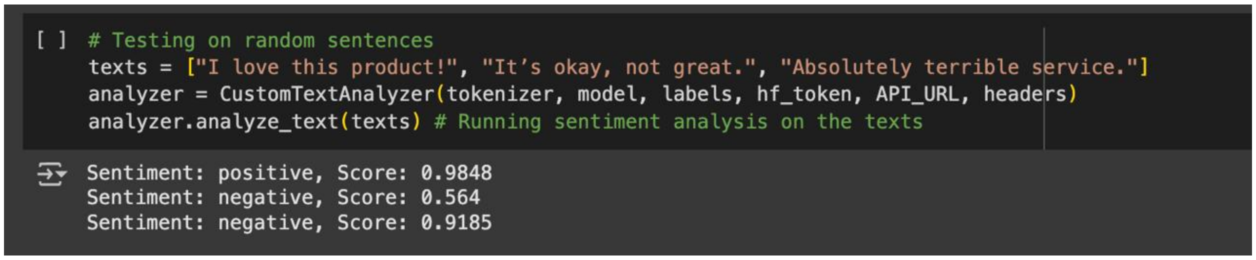
This code defines all the model details and requirements for the use of the API, for example the token, choice of model and URL related parameters that have to be sent to the model for prediction.

```
# Testing on random sentences
texts = ["I love this product!", "It's okay, not great.", "Absolutely terrible
service."]
analyzer = CustomTextAnalyzer(tokenizer, model, labels, hf_token, API_URL,
headers)
analyzer.analyze_text(texts) # Running sentiment analysis on the texts
```

This code sends random sentences that have been declared by us to the Analyzer. The analyzer then predicts and returns the sentiment and the score.

Reflection

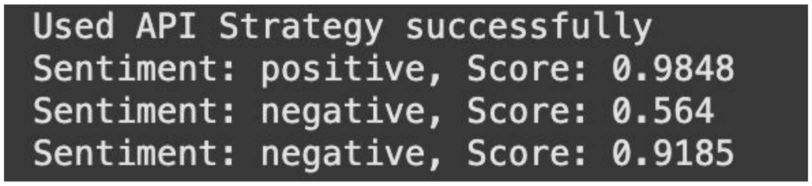
The first design pattern chosen was the Strategy Design Pattern which enables the application to easily switch between different models without having to change the main logic. In this specific project, it can be a choice of using a pre-trained model RoBERTa which is locally deployed or an inference API from HuggingFace. Another pattern implemented is the Circuit Breaker Pattern which defaults to a strategy to be the primary one which in this case is the implementation of the API. If there is something wrong with running the API, it then sends the inference service to RoBERTa model. An example of what could go wrong is importing the wrong package, not importing the package, wrong API key and such as shown below:



```
[ ] # Testing on random sentences
    texts = ["I love this product!", "It’s okay, not great.", "Absolutely terrible service."]
    analyzer = CustomTextAnalyzer(tokenizer, model, labels, hf_token, API_URL, headers)
    analyzer.analyze_text(texts) # Running sentiment analysis on the texts
```

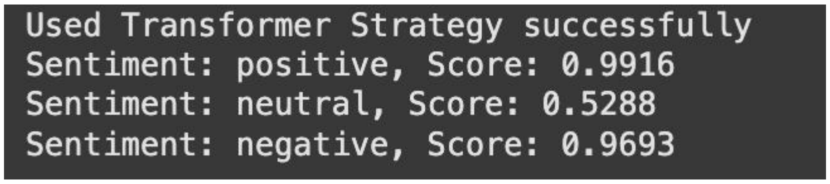
```
Sentiment: positive, Score: 0.9848
Sentiment: negative, Score: 0.564
Sentiment: negative, Score: 0.9185
```

This should be the result for when the application is run normally.



```
Used API Strategy successfully
Sentiment: positive, Score: 0.9848
Sentiment: negative, Score: 0.564
Sentiment: negative, Score: 0.9185
```

Here we added a small statement to see if the circuit breaker pattern was working properly which in if it’s the case, it will print the statement “Used API Strategy successfully” along with the labels and scores.



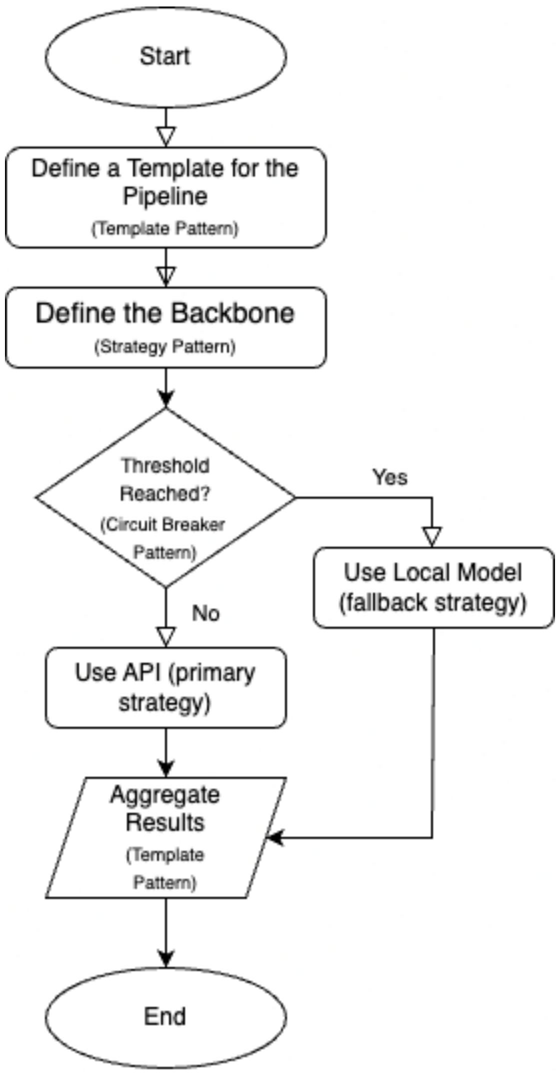
```
Used Transformer Strategy successfully
Sentiment: positive, Score: 0.9916
Sentiment: neutral, Score: 0.5288
Sentiment: negative, Score: 0.9693
```

And this is an example on how the circuit breaker strategy pattern works if a different API key was given and or no package was imported.

The last strategy that was implemented was the template method which gives a structure that can be used repeatedly which allows subclasses to define the specific strategy to use.

Integration of design patterns

As you can see in the flowchart below the selected strategies have been implemented in the project in the following ways. The Traditions Template pattern and Strategy pattern have been used as AI patterns to integrate the AI workflow. The resilience pattern chosen in the circuit breaker pattern. In the flow chart you can see the steps at which each of these patterns were implemented during the flow of the project. The template pattern defines the flow/pipeline of execution starting from processing text input, to running the sentiment model to finally formatting and displaying the results. The Second piece of the template chooses the flow of execution depending on the user input of which strategy to use. Therefore the Strategy pattern is used to define the backbone of the execution of sentiment analysis for the selected methods. The Specific strategies like API and Local Transformer model, then inherit the properties for the Strategy pattern. If the requirements for the primary strategy (API) have not been met, the circuit breaker strategy opts to choose the fallback strategy (Local Transformer). The results from the suitable model are then collected and displayed along with the scores and sentiments as a part of the third piece of the template strategy.



Insights

Some of the insights we gained from implementing this project are discussed below. The integration of Strategies allows a clear modular representation of the code. It helped us understand the flow in a clear sequential manner. The circuit breaker strategy helped us understand the switching of strategies depending on the threshold conditions. This taught us the importance of having a fallback strategy if there is a dependency of API or external processes in case of their failure. Working with both the locally pretrained RoBERTa model and the API-based version showed key differences in performance, reliability and usability. The local model offered us greater control over the inference process, allowing us direct access to prediction scores and making it easier to understand and debug outputs. Once downloaded it doesn’t rely on an internet connection, which improves speed and ensures consistent availability. However, it requires more computational resources and a proper setup environment (importing proper libraries etc). In contrast, the API-based model was much easier to integrate and use, requiring no model downloads or hardware setup. It always uses the most up-to-date version, but its dependence on network connectivity, which introduces latency and potential instability due to rate limits or service interruptions. For us, combining both approaches through the Circuit Breaker pattern proved to be an effective solution, ensuring reliable sentiment analysis by automatically switching to the local model when the API is unavailable, making the whole system more flexible.