FISTA

What is TDD?

TDD in prompt engineering?

Prepared By:

[FISTA SOLUTIONS]

Prepared for:

[**Team**]

Date of Document:

[2024-10-10]

Document Version:

[1]

https://www.fistasolutions.com/

+92-301-3949480

What is TDD?

Test Driven Development (TDD) is a software development approach in which tests are written before the actual code. This methodology is part of Agile development practices and emphasizes writing small, incremental tests that guide the development process. TDD focuses on the idea that a well-written test serves as a blueprint for coding, helping to clarify the requirements and design upfront.

Key Phases of TDD:

1. **Write a Test**:
    - Before writing any functional code, you write a unit test that defines a new functionality or feature. The test will initially fail because the feature doesn't exist yet.
2. **Run the Test**:
    - Run the test to confirm that it fails. This failure indicates that the feature hasn't been implemented yet, and the test is effective in detecting the lack of functionality.
3. **Write Code to Pass the Test**:
    - Write the minimum amount of code necessary to pass the failing test. At this stage, focus only on writing enough code to make the test pass, not on creating a perfect or optimized solution.
4. **Run the Test Again**:
    - Run the test again to ensure that the newly written code passes the test.
5. **Refactor the Code**:
    - Improve the code without changing its external behavior, ensuring that it remains clean, efficient, and maintainable. After refactoring, all tests should still pass.
6. **Repeat**:

- Continue the process by writing the next test and repeating the cycle.

Benefits of TDD:

- **Improved Code Quality**: Writing tests before coding ensures that every feature is well-tested, reducing the chance of bugs.
- **Better Design and Clarity**: TDD helps developers think through the design and requirements of a feature before implementing it.
- **Less Debugging**: Since code is constantly tested during development, developers spend less time debugging later.
- **More Confidence in Refactoring**: Since tests are already in place, you can refactor the code with confidence, knowing that the tests will catch any unintended issues.
- **Documentation**: Tests serve as documentation, making it easier for other developers to understand the code.

Example of a TDD Workflow:

Test: Write a test for adding two numbers.

```
def test_add():
    assert add(2, 3) == 5
```

Code: Write the minimal code to pass the test.

```
def add(a, b):
    return a + b
```

1. **Run**: Ensure the test passes.

2. **Refactor**: Improve the code, if necessary, and rerun the test to confirm everything works.

How to implement TDD in prompt engineering?

Implementing **Test-Driven Development (TDD)** in **prompt engineering** can be highly effective for creating reliable and consistent prompts for AI models like ChatGPT. The process involves defining specific expected outputs for your prompts, testing the model's behavior, and iterating the prompt design based on the results. Here's how you can apply TDD principles to prompt engineering:

Steps to Implement TDD in Prompt Engineering:

1. **Write a Test for the Prompt:**
    - Before crafting a prompt, define the expected outcome of the model's response. This involves creating a test case that specifies the input prompt and the desired output.

Example: Suppose you want to generate a description of a product. Write a test that describes the ideal response:

```
def test_product_description():
    prompt = "Describe the features of a new smartphone."
    expected_output = "A new smartphone with a high-resolution
display, fast processor, and long battery life."
    assert generate_response(prompt) == expected_output
```

2. **Run the Test and Observe Failure:**
    - Run the test by using the initial prompt with the AI model. In the beginning, this test will likely fail because the prompt has not been optimized yet.

Example: The model might generate a response like:

```
"This smartphone is nice."
```

3. **Write the Prompt to Pass the Test:**
    - Modify the prompt to get closer to the desired outcome. This could involve restructuring the prompt, adding clarifications, or experimenting with prompt formats until the model's output meets expectations.

Example: Modify the prompt to:

```
"Describe the key features of a high-end smartphone, focusing on display quality, processor speed, and battery life."
```

4. **Run the Test Again:**
    - After modifying the prompt, run the test again. If the model's response aligns with the expected output, the test will pass.

Example Response:

```
"This smartphone features a 6.7-inch high-resolution display, a lightning-fast processor for seamless multitasking, and a long-lasting battery for all-day use."
```

    - If the response is still not as expected, continue refining the prompt.
5. **Refactor the Prompt:**
    - Once the prompt successfully generates the expected result, consider refactoring it to be cleaner or more efficient. Ensure the prompt is optimized for both clarity and conciseness.

Example Refactor:

```
"List the top three features of a high-end smartphone: display, processor, and battery life."
```

-

6. **Repeat the Process:**
    - Continue by writing the next test for another aspect of the prompt or for a new prompt entirely. Ensure each test focuses on a specific feature or expected output to maintain clarity.

Practical Example of TDD in Prompt Engineering:

Let's apply TDD to a simple chatbot prompt that gives basic travel recommendations.

Write a Test for a Travel Recommendation Prompt:

```
def test_travel_recommendation():
    prompt = "Give a travel recommendation for someone visiting Japan."
    expected_output = "Visit Tokyo to experience modern architecture, or Kyoto for traditional temples and cultural heritage."
    assert generate_response(prompt) == expected_output
```

Run the Test (Initial Prompt):

```
"Japan is a great place to visit. You can find many interesting things to do."
```

- This is a vague and general response, so the test will fail.

Improve the Prompt:

```
"Provide a detailed travel recommendation for someone visiting Japan. Mention key cities, cultural experiences, and major attractions."
```

Run the Test Again:

New response:
`"For a visit to Japan, explore the bustling city of Tokyo with its modern architecture and attractions like Shibuya Crossing and Tokyo Tower. For a more traditional experience, visit Kyoto to see famous temples like Kinkaku-ji and Fushimi Inari Shrine."`

This response is closer to the expected output, so the test may now pass.

2. **Refactor the Prompt:**

Refactor for more precision or clarity:
`"Recommend top cities and experiences for a first-time visitor to Japan, highlighting both modern and traditional attractions."`

Tools and Techniques for TDD in Prompt Engineering:

- **Unit Testing**: If you have custom testing environments or APIs for the model, create automated tests for each prompt and its expected response.
- **Version Control**: Track changes to prompts as you modify and improve them. This will help ensure consistency and reproducibility of results.
- **Test Coverage**: Write tests that cover various edge cases or different inputs, ensuring your prompts work for a wide range of situations.
- **Continuous Improvement**: Prompt engineering is iterative. Run tests for different types of inputs to optimize performance for diverse cases (e.g., varying languages, domains, or audiences).

Example Framework for Test Automation:

If you're using a custom environment, you can automate TDD for prompt engineering using Python's `unittest` or similar testing frameworks:

```
import unittest
```

```
def generate_response(prompt):
    # Simulate the model generating a response (replace with
actual model API)
    return "Simulated model response"

class TestPromptEngineering(unittest.TestCase):
    def test_basic_prompt(self):
        prompt = "Describe a sunset."
        expected_output = "A sunset with vibrant hues of orange,
pink, and purple."
        self.assertEqual(generate_response(prompt),
expected_output)

    def test_travel_recommendation(self):
        prompt = "Give a travel recommendation for Paris."
        expected_output = "Visit the Eiffel Tower, the Louvre,
and enjoy a walk along the Seine River."
        self.assertEqual(generate_response(prompt),
expected_output)

if __name__ == '__main__':
    unittest.main()
```

By applying TDD to prompt engineering, you ensure that the prompts are consistent, meet specific goals, and are continuously refined to achieve optimal results.