MS Docs: Prompt engineering techniques with Azure OpenAI - Azure OpenAI Service | Microsoft Learn

Prompt engineering techniques with Azure OpenAI - Azure OpenAI Service | Microsoft Learn

Source: https://github.com/microsoft/generative-ai-for-beginners/tree/main/05-advanced-prompts

Let's recap some learnings from the previous chapter:

Prompt *engineering* is the process by which we **guide the model towards more relevant responses** by providing more useful instructions or context.

There are also two steps to writing prompts, constructing the prompt, by providing relevant context and the second part is *optimization*, how to gradually improve the prompt.

At this point, we have some basic understanding of how to write prompts, but we need to go deeper. In this chapter, you will go from trying out various prompts to understanding why one prompt is better than another. You will learn how to construct prompts following some basic techniques that can be applied to any LLM.

Introduction

In this chapter, we will cover the following topics:

- Extend your knowledge of prompt engineering by applying different techniques to your prompts.
- Configuring your prompts to vary the output.

Learning goals

After completing this lesson, you'll be able to:

- Apply prompt engineering techniques that improve the outcome of your prompts.
- Perform prompting that is either varied or deterministic.

Prompt engineering

Prompt engineering is the process of creating prompts that will produce the desired outcome. There's more to prompt engineering than just writing a text prompt. Prompt engineering is not an engineering discipline, it's more a set of techniques that you can apply to get the desired outcome.

An example of a prompt

Let's take a basic prompt like this one:

Generate 10 questions on geography.

In this prompt, you are actually applying a set of different prompt techniques.

Let's break this down.

- **Context**, you specify it should be about "geography".
- **Limiting the output**, you want no more than 10 questions.

Limitations of simple prompting

You may or may not get the desired outcome. You will get your questions generated, but geography is a big topic and you may not get what you want to due the following reasons:

- **Big topic**, you don't know if it's going to be about countries, capitals, rivers and so on.
- **Format**, what if you wanted the questions to be formatted in a certain way?

As you can see, there's a lot to consider when creating prompts.

So far, we've seen a simple prompt example, but generative AI is capable of much more to help people in a variety of roles and industries. Let's explore some basic techniques next.

Techniques for prompting

First, we need to understand that prompting is an *emergent* property of an LLM meaning that this is not a feature that is built into the model but rather something we discover as we use the model.

There are some basic techniques that we can use to prompt an LLM. Let's explore them.

- **Zero-shot prompting**, this is the most basic form of prompting. It's a single prompt requesting a response from the LLM based solely on its training data.
- **Few-shot prompting**, this type of prompting guides the LLM by providing 1 or more examples it can rely on to generate its response.
- **Chain-of-thought**, this type of prompting tells the LLM how to break down a problem into steps.
- **Generated knowledge**, to improve the response of a prompt, you can provide generated facts or knowledge additionally to your prompt.
- **Least to most**, like chain-of-thought, this technique is about breaking down a problem into a series of steps and then ask these steps to be performed in order.
- **Self-refine**, this technique is about critiquing the LLM's output and then asking it to improve.
- **Maieutic prompting**. What you want here is to ensure the LLM answer is correct and you ask it to explain various parts of the answer. This is a form of self-refine.

Zero-shot prompting

This style of prompting is very simple, it consists of a single prompt. This technique is probably what you're using as you're starting to learn about LLMs. Here's an example:

- Prompt: "What is Algebra?"
- Answer: "Algebra is a branch of mathematics that studies mathematical symbols and the rules for manipulating these symbols."

Few-shot prompting

This style of prompting helps the model by providing a few examples along with the request. It consists of a single prompt with additional a task-specific data. Here's an example:

- Prompt: "Write a poem in the style of Shakespeare. Here are a few examples of Shakespearean sonnets.: Sonnet 18: 'Shall I compare thee to a summer's day? Thou art more lovely and more temperate...' Sonnet 116: 'Let me not to the marriage of true minds Admit impediments. Love is not love Which alters when it alteration finds...' Sonnet 132: 'Thine eyes I love, and they, as pitying me, Knowing thy heart torment me with disdain,...' Now, write a sonnet about the beauty of the moon."
- Answer: "Upon the sky, the moon doth softly gleam, In silv'ry light that casts its gentle grace,..."

Examples provide the LLM with the context, format or style of the desired output. They help the model understand the specific task and generate more accurate and relevant responses.

Chain-of-thought

Chain-of-thought is a very interesting technique as it's about taking the LLM through a series of steps. The idea is to instruct the LLM in such a way that it understands how to do something. Consider the following example, with and without chain-of-thought:

```
- Prompt: "Alice has 5 apples, throws 3 apples, gives 2 to Bob and Bob gives one back, how many apples does Alice have?"
- Answer: 5
```

LLM answers with 5, which is incorrect. Correct answer is 1 apple, given the calculation (5 -3 -2 + 1 = 1).

So how can we teach the LLM to do this correctly?

Let's try chain-of-thought. Applying chain-of-thought means:

1. Give the LLM a similar example.
2. Show the calculation, and how to calculate it correctly.
3. Provide the original prompt.

Here's how:

- Prompt: "Lisa has 7 apples, throws 1 apple, gives 4 apples to Bart and Bart gives one back: 7 -1 = 6 6 -4 = 2 2 +1 = 3
  Alice has 5 apples, throws 3 apples, gives 2 to Bob and Bob gives one back, how many apples does Alice have?" Answer: 1

Note how we write substantially longer prompts with another example, a calculation and then the original prompt and we arrive at the correct answer 1.

As you can see chain-of-thought is a very powerful technique.

Generated knowledge

Many times when you want to construct a prompt, you want to do so using your own company's data. You want part of the prompt to be from the company and the other part should be the actual prompt you're interested in.

As an example, this is what your prompt then can look like if you're in the insurance business:

```text
{{company}}: {{company_name}}
{{products}}:
{{products_list}}
Please suggest an insurance given the following budget and requirements:
Budget: {{budget}}
Requirements: {{requirements}}
```

Above, you see how the prompt is constructed using a template. In the template there's a number of variables, denoted by `{{variable}}`, that will be replaced with actual values from a company API. Here's an example of how the prompt could look like once the variables have been replaced by content from your company:

```text
Insurance company: ACME Insurance
Insurance products (cost per month):
- Car, cheap, 500 USD
- Car, expensive, 1100 USD
- Home, cheap, 600 USD
- Home, expensive, 1200 USD
- Life, cheap, 100 USD

Please suggest an insurance given the following budget and requirements:
Budget: $1000
Requirements: Car, Home
```

Running this prompt through an LLM will produce a response like this:

```output
, and Life insurance

Given the budget and requirements, we suggest the following insurance package from ACME Insurance:
- Car, cheap, 500 USD
- Home, cheap, 600 USD
```

```
- Life, cheap, 100 USD
Total cost: $1,200 USD
```

As you can see, it also suggests the Life insurance, which it shouldn't. This result is an indication that we need to optimize the prompt by changing the prompt to be clearer on what it can allow. After some *trial and error*, we arrive at the following prompt:

```text
Insurance company: ACME Insurance
Insurance products (cost per month):
- type: Car, cheap, cost: 500 USD
- type: Car, expensive, cost: 1100 USD
- type: Home, cheap, cost: 600 USD
- type: Home, expensive, cost: 1200 USD
- type: Life, cheap, cost: 100 USD

Please suggest an insurance given the following budget and requirements:
Budget: $1000 restrict choice to types: Car, Home
```

Note how adding *type* and *cost* and also using the keyword *restrict* helps the LLM to understand what we want.

Now we get the following response:

```output
Given the budget and requirements, we suggest the Car, Cheap insurance product which costs 500 USD per month.
```

The point of this example was to show that even though we're using a basic technique like *generated knowledge*, we still need to optimize the prompt in most cases to get the desired outcome.

Least-to-most

The idea with Least-to-most prompting is to break down a bigger problem into subproblems. That way, you help guide the LLM on how to "conquer" the bigger problem. A good example could be for data science where you can ask the LLM to divide up a problem like so:

Prompt: How to perform data science in 5 steps?

With your AI assistant answering with:

1. Collect data
2. Clean data
3. Analyze data
4. Plot data
5. Present data

Self-refine, critique the results

With generative AIs and LLMs, you can't trust the output. You need to verify it. After all, the LLM is just presenting you what's the next most likely thing to say, not what's correct. Therefore, a good idea is to ask the LLM to critique itself, which leads us to the self-refine technique.

How it works is that you follow the following steps:

1. Initial prompt asking the LLM to solve a problem
2. LLM answers
3. You critique the answer and ask the AI to improve
4. LLM answers again, this time considering the critique and suggest solutions it came up with

You can repeat this process as many times as you want.

Here's an example using this technique:

Prompt: "Create a Python Web API with routes products and customers"

AI Answer:

```
```python
import flask
app = Flask(__name__)

@app.route('/products')
def products():
    return 'Products'

@app.route('/customers')
def customers():
    return 'Customers'
```
```

Prompt: suggest 3 improvements of the above code

AI Answer:

```
```text
Sure, here are three suggestions to improve the code:

Import the correct module: The first line should import Flask instead of flask. The correct import
statement should be from flask import Flask.

Add a main function: Add a if __name__ == '__main__': block to the end of the file to run the
application when the file is executed.

Use f-strings: Use f-strings to format the return statements in the route functions. For example,
return 'Products' can be changed to return f'This is the {request.path} page'.
```

```python
from flask import Flask, request

app = Flask(__name__)

@app.route('/products')
def products():
```
```

```
    return f'This is the {request.path} page'

@app.route('/customers')
def customers():
    return f'This is the {request.path} page'

if __name__ == '__main__':
    app.run()
```

As you can see, the above AI response is improving the first suggested code thanks to critiquing the first answer.

Maieutic prompting

Maieutic prompting is a technique that is similar to self-refine but it's more about asking the LLM to explain itself. The goal is to reduce inconsistencies in the LLM's output so to ensure it arrives at the correct answer. The workflow to follow is:

1. Ask the LLM to answer a question
2. For each part of the answer, ask the LLM to explain it more in depth.
3. If there are inconsistencies, discard the parts that are inconsistent.

Repeat 2 and 3 until you've gone through all the parts and you're satisfied with the answer.

Here's an example prompt:

prompt: How can I create a crisis plan to mitigate a pandemic in 5 steps? LLM answer:

```text
1. Identify the risks
1. Identify the stakeholders
1. Identify the resources
1. Identify the actions
1. Identify the communication channels
```

It's identified 5 steps, but can we determine if this is correct? Let's ask the LLM to explain each step:

prompt: Explain the first step in more detail, what are the risks in detail with a pandemic?

```text
The risks are the following:

- Loss of life
- Loss of business
- Loss of reputation
- Loss of customers
- Loss of employees
```

At this point, we want to make sure the LLM is correct so we ask it to explain the first risk in more detail and hope it's consistent with the answer above:

prompt: In a Pandemic, which is the biggest risk and why?

```text
The biggest risk is loss of life because it's the most important thing to protect.
```

What are the two biggest risks in a pandemic?

```text
The two biggest risks are loss of life and loss of business.
```

At this point, the LLM is consistent and mentions "life" and "business" as the top two risks. We can now continue to the next step and feel fairly confident. However, we shouldn't trust the LLM blindly, we should always verify the output.

Vary your output

LLMs are nondeterministic by nature, meaning that you will get different results each time you run the same prompt. Try the following prompt for example:

"Generate code for a Python Web API"

```text
# Import necessary modules
import flask
from flask import request, jsonify

# Create a Flask app
app = flask.Flask(__name__)

# Create a route for the API
@app.route('/api', methods=['GET'])
def api():
    # Get the data from the request
    data = request.args

    # Process the data
    result = process_data(data)

    # Return the result as JSON
    return jsonify(result)

# Function to process the data
def process_data(data):
    # Do something with the data
    result = data + 1

    # Return the result
    return result

# Run the app
if __name__ == '__main__':
    app.run()

```

Running the same prompt again generates a slightly different response:

```text
#import necessary packages
import flask
from flask import request, jsonify

#create the Flask application
app = flask.Flask(__name__)
app.config["DEBUG"] = True

#create a list of books
books = [
    {'id': 0,
     'title': 'A Fire Upon the Deep',
     'author': 'Vernor Vinge',
     'first_sentence': 'The coldsleep itself was dreamless.',
     'year_published': '1992'},
    {'id': 1,
     'title': 'The Ones Who Walk Away From Omelas',
     'author': 'Ursula K. Le Guin',
     'first_sentence': 'With a clamor of bells that set the swallows soaring, the Festival of
Summer came to the city Omelas, bright-towered by the sea.',
     'published': '1973'},
    {'id': 2,
     'title': 'Dhalgren',
     'author': 'Samuel R. Delany',
     'first_sentence': 'to wound the autumnal city.',
     'published': '1975'}
]

#create an endpoint for the API
@app.route('/', methods=['GET'])
def home():
    return '''<h1>Books API</h1>
<p>A prototype API for retrieving books.</p>'''

#create an endpoint to return all books
@app.route('/api/v1/resources/books/all', methods=['GET'])
def api_all():
    return jsonify(books)

#create an endpoint to return a single book
@app.route('/api/v1/resources/books', methods=['GET'])
def api_id():
    #check if an ID was provided as part of the URL
    #if ID is provided, assign it to a variable
    #if no ID is provided, display an error
    if 'id' in request.args:
        id = int(request.args['id'])
    else:
        return "Error: No id field provided. Please specify an id."

    #create an empty list for our results
    results = []

    #loop through the data and match results that fit the requested ID
    #IDs are unique, but other fields might return many results
    for book in books:
        if book['id'] == id:
            results.append(book)

    #use the jsonify function from Flask to convert our list of
```

```
    #Python dictionaries to the JSON format
    return jsonify(results)

app.run()
```

So is the varied output a problem?

Depends on what you're trying to do. If you want a specific response then it's a problem. If you're ok with a varied output like "Generate any 3 questions on geography", then it's not a problem.

Using temperature to vary your output

Ok, so we've decided we want to limit the output to be more predictable, that is more deterministic. How do we do that?

Temperature is a value between 0 and 1, where 0 is the most deterministic and 1 is the most varied. The default value is 0.7. Let's see what happens with two runs of the same prompt with temperature set to 0.1:

"Generate code for a Python Web API"

```
```python
#import necessary libraries
import flask
from flask import request, jsonify

#create an instance of the Flask class
app = flask.Flask(__name__)

#create an endpoint for the API
@app.route('/api/v1/endpoint', methods=['GET'])
def api_endpoint():
    #get the request data
    data = request.get_json()

    #process the data
    result = process_data(data)

    #return the result
    return jsonify(result)

#function to process the data
def process_data(data):
    #process the data
    result = {'result': 'success'}

    #return the result
    return result

#run the Flask app
if __name__ == '__main__':
    app.run()

```
```

Running the prompt again gives us this outcome:

```python
#import necessary libraries
import flask
from flask import request, jsonify

#create a Flask app
app = flask.Flask(__name__)

#create a route for the API
@app.route('/api', methods=['GET'])
def api_call():
    #get the data from the request
    data = request.get_json()

    #process the data
    result = process_data(data)

    #return the result
    return jsonify(result)

#define the process_data function
def process_data(data):
    #do something with the data
    result = data + 1

    #return the result
    return result

#run the app
if __name__ == '__main__':
    app.run()

```

There's only a tiny difference between these two outputs. Let's do the opposite this time, let's set the temperature to 0.9:

```python
# Import necessary libraries
import flask
from flask import request, jsonify

# Create a Flask app
app = flask.Flask(__name__)

# Create a route for the API
@app.route('/api', methods=['GET'])
def api_call():
    # Get the data from the request
    data = request.args

    # Process the data
    result = process_data(data)

    # Return the result
    return jsonify(result)

# Function to process the data
def process_data(data):
    # Do something with the data
    result = data + 1
```

```
    # Return the result
    return result

# Run the app
if __name__ == '__main__':
    app.run()

```

and the second attempt at 0.9 as temperature value:

```python
import flask
from flask import request, jsonify

# create the Flask app
app = flask.Flask(__name__)
app.config['DEBUG'] = True

# create some test data
books = [
    {'id': 0, 'title': 'A Fire Upon The Deep', 'author': 'Vernor Vinge', 'first_sentence': 'The coldsleep itself was dreamless.', 'year_published': '1992'},
    {'id': 1, 'title': 'The Ones Who Walk Away From Omelas', 'author': 'Ursula K. Le Guin', 'first_sentence': 'With a clamor of bells that set the swallows soaring, the Festival of Summer came to the city Omelas, bright-towered by the sea.', 'published': '1973'},
    {'id': 2, 'title': 'Dhalgren', 'author': 'Samuel R. Delany', 'first_sentence': 'to wound the autumnal city.', 'published': '1975'}
]

# create an endpoint
@app.route('/', methods=['GET'])
def home():
    return '''<h1>Welcome to our book API!</h1>'''

@app.route('/api/v1/resources/books

```

As you can see, the results couldn't be more varied.

Note, that there are more parameters you can change to vary the output, like top-k, top-p, repetition penalty, length penalty and diversity penalty but these are outside the scope of this curriculum.

Good practices

There are many practices you can apply to try to get what you want. You will find your own style as you use prompting more and more.

Additionally to the techniques we've covered, there are some good practices to consider when prompting an LLM.

Here are some good practices to consider:

- **Specify context**. Context matters, the more you can specify like domain, topic, etc. the better.

- Limit the output. If you want a specific number of items or a specific length, specify it.
- **Specify both what and how**. Remember to mention both what you want and how you want it, for example "Create a Python Web API with routes products and customers, divide it into 3 files".
- **Use templates**. Often, you will want to enrich your prompts with data from your company. Use templates to do this. Templates can have variables that you replace with actual data.
- **Spell correctly**. LLMs might provide you with a correct response, but if you spell correctly, you will get a better response.