```
print(df)
```

Output:

```
  variant                                             prompt
  \
0       A  Product description: A pair of shoes that can ...
1       A  Product description: A pair of shoes that can ...
2       A  Product description: A pair of shoes that can ...
3       A  Product description: A pair of shoes that can ...
4       A  Product description: A pair of shoes that can ...
5       B  Product description: A home milkshake maker.\n...
6       B  Product description: A home milkshake maker.\n...
7       B  Product description: A home milkshake maker.\n...
8       B  Product description: A home milkshake maker.\n...
9       B  Product description: A home milkshake maker.\n...

                                            response
0  1. Adapt-a-Fit Shoes \n2. Omni-Fit Footwear \n...
1  1. OmniFit Shoes\n2. Adapt-a-Sneaks \n3. OneFi...
2  1. Adapt-a-fit\n2. Flexi-fit shoes\n3. Omni-fe...
3  1. Adapt-A-Sole\n2. FitFlex\n3. Omni-FitX\n4. ...
4  1. Omni-Fit Shoes\n2. Adapt-a-Fit Shoes\n3. An...
5  Adapt-a-Fit, Perfect Fit Shoes, OmniShoe, OneS...
6       FitAll, OmniFit Shoes, SizeLess, AdaptaShoes
7       AdaptaFit, OmniShoe, PerfectFit, AllSizeFit.
8  FitMaster, AdaptoShoe, OmniFit, AnySize Footwe...
9        Adapt-a-Shoe, PerfectFit, OmniSize, FitForm
```

Here we're using the OpenAI API to generate model responses to a set of prompts and storing the results in a dataframe, which is saved to a CSV file. Here's how it works:

1. Two prompt variants are defined, and each variant consists of a product description, seed words, and potential product names, but `prompt_B` provides two examples.
2. Import statements are called for the pandas library, openai library, and os library.

3. The `get_response` function takes a prompt as input and returns a response from the `gpt-3.5-turbo` model. The prompt is passed as a user message to the model, along with a system message to set the model's behavior.
4. Two prompt variants are stored in the `test_prompts` list.
5. An empty list `responses` is created to store the generated responses, and the variable `num_tests` is set to 5.
6. A nested loop is used to generate responses. The outer loop iterates over each prompt, and the inner loop generates `num_tests` (5 in this case) number of responses per prompt.
    - The `enumerate` function is used to get the index and value of each prompt in `test_prompts`. This index is then converted to a corresponding uppercase letter (e.g., 0 becomes *A*, 1 becomes *B*) to be used as a variant name.
    - For each iteration, the `get_response` function is called with the current prompt to generate a response from the model.
    - A dictionary is created with the variant name, the prompt, and the model's response, and this dictionary is appended to the `responses` list.
7. Once all responses have been generated, the `responses` list (which is now a list of dictionaries) is converted into a pandas dataframe.
8. This dataframe is then saved to a CSV file with the Pandas built-in `to_csv` function, naking the file *responses.csv* with `index=False` so as to not write row indices.
9. Finally, the dataframe is printed to the console.

Having these responses in a spreadsheet is already useful, because you can see right away even in the printed response that `prompt_A` (zero-shot) in the first five rows is giving us a numbered list, whereas `prompt_B` (few-shot) in the last five rows, tends to output the desired format of a comma separated in-line list. The next step is to give a rating on each of the responses, which is best done blind and randomized in order to avoid favoring one prompt over another.

Input:

```
import ipywidgets as widgets
from IPython.display import display
import pandas as pd

# load the responses.csv file
df = pd.read_csv("responses.csv")

# Shuffle the dataframe
df = df.sample(frac=1).reset_index(drop=True)

# df is your dataframe and 'response' is the column with the
# text you want to test
response_index = 0
# add a new column to store feedback
df['feedback'] = pd.Series(dtype='str')

def on_button_clicked(b):
    global response_index
    #  convert thumbs up / down to 1 / 0
    user_feedback = 1 if b.description == "\U0001F44D" else 0

    # update the feedback column
    df.at[response_index, 'feedback'] = user_feedback

    response_index += 1
    if response_index < len(df):
        update_response()
    else:
        # save the feedback to a CSV file
        df.to_csv("results.csv", index=False)

        print("A/B testing completed. Here's the results:")
        # Calculate score and num rows for each variant
        summary_df = df.groupby('variant').agg(
            count=('feedback', 'count'),
            score=('feedback', 'mean')).reset_index()
        print(summary_df)

def update_response():
    new_response = df.iloc[response_index]['response']
    if pd.notna(new_response):
        new_response = "<p>" + new_response + "</p>"
    else:
```

```
            new_response = "<p>No response</p>"
        response.value = new_response
        count_label.value = f"Response: {response_index + 1}"
        count_label.value += f"/{len(df)}"

    response = widgets.HTML()
    count_label = widgets.Label()

    update_response()

    thumbs_up_button = widgets.Button(description='\U0001F44D')
    thumbs_up_button.on_click(on_button_clicked)

    thumbs_down_button = widgets.Button(
        description='\U0001F44E')
    thumbs_down_button.on_click(on_button_clicked)

    button_box = widgets.HBox([thumbs_down_button,
    thumbs_up_button])

    display(response, button_box, count_label)
```

The result, if you run this in a Jupyter Notebook, is a widget that displays each response, with a thumbs up or down (see Figure 1-12). This provides a simple interface for quickly labeling responses, with minimal overhead. If you were to do this outside of a Jupyter Notebook, you could change the thumbs up and down emojis for *Y* and *N*, and a loop using the built-in `input()` function, as a text-only replacement for iPyWidgets.

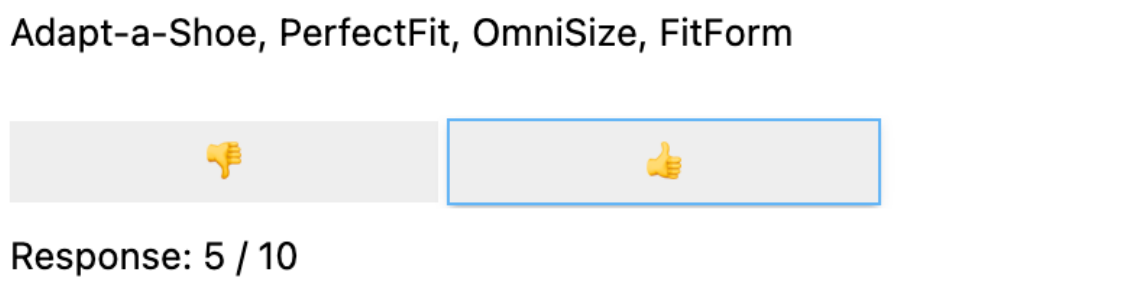

Figure 1-12. Thumbs up/down rating system

Once you've finished labeling the responses, you get the output, which shows you how each prompt performs.

Output:

```
A/B testing completed. Here's the results:
  variant  count  score
0       A      5    0.2
1       B      5    0.6
```

The dataframe was shuffled at random, and each response was labelled blind (without seeing the prompt), so you get an accurate picture of how often each prompt performed. Here is the step-by-step explanation:

1. Three modules are imported: `ipywidgets`, `IPython.display`, and `pandas`.
    - `ipywidgets` are interactive HTML widgets for Jupyter notebooks and the IPython kernel.
    - `IPython.display` module provides classes for displaying various types of output like images, sound, displaying HTML, etc.
    - `pandas` is a powerful data manipulation library.
2. The `pandas` library is used to read in a CSV file named *responses.csv*, which contains the responses that you want to test. This creates a pandas dataframe `df`.
3. The dataframe `df` is shuffled using the `sample()` function with `frac=1` which means it uses all the rows. The `reset_index(drop=True)` is used to reset the indices to standard 0, 1, 2, ..., n index.
4. The script defines `response_index` as 0. This is used to track which response from the dataframe the user is currently viewing.
5. A new column `feedback` is added to the dataframe `df` with the data type as `str` or string.
6. Next, the script defines a function `on_button_clicked(b)`, which will execute whenever one of the two buttons in the interface is clicked.
    - The function first checks the `description` of the button clicked was the thumbs up (“ ?button, and sets `user_feedback` as 1, or if it was the thumbs down (“ button, it sets `user_feedback` as 0.

- Then it updates the `feedback` column of the dataframe at the current `response_index` with `user_feedback`.
- After that, it increments `response_index` to move to the next response.
- If `response_index` is still less than the total number of responses (i.e., the length of the dataframe), it calls the function `update_response()`.
- If there are no more responses, it saves the dataframe to a new CSV file *results.csv*, then it prints a message, and also prints a summary of the results by variant, showing the count of feedback received and the average score (mean) for each variant.

7. The function `update_response()` fetches the next response from the dataframe, wraps it in paragraph HTML tags (if it's not null), updates the `response` widget to display the new response, and updates the `count_label` widget to reflect the current response number and total number of responses.
8. Two widgets, `response` (an HTML widget) and `count_label` (a Label widget), are instantiated. The `update_response()` function is then called to initialize these widgets with the first response and the appropriate label.
9. Two more widgets, `thumbs_up_button` and `thumbs_down_button` (both Button widgets), are created with thumbs up and thumbs down emoji as their descriptions, respectively. Both buttons are configured to call the `on_button_clicked()` function when clicked.
10. The two buttons are grouped into a horizontal box (`button_box`) using the `HBox` function.
11. Finally, the `response`, `button_box`, and `count_label` widgets are displayed to the user using the `display()` function from the `IPython.display` module.

A simple rating system such as this one can be useful in judging prompt quality and encountering edge cases. Usually in under 10 test runs of a prompt you uncover a deviation, which you otherwise wouldn't have caught until you started using it in production. The downside is that it can get tedious rating lots of responses manually, and your ratings might not represent the preferences of your intended audience. However, even

small numbers of tests can reveal large differences between two prompting strategies, and reveal non-obvious issues before reaching production.

Iterating on and testing prompts can lead to radical decreases in the length of the prompt, and therefore the cost and latency of your system. If you can find another prompt that performs equally as well (or better) but uses a shorter prompt, you can afford to scale up your operation considerably. Often you'll find in this process that many elements of a complex prompt are completely superflouous, or even counter productive.

The *thumbs up* or other manually labelled indicators of quality doesn't have to be the only judging criteria. Human evaluation is generally considered to be the most accurate form of feedback. However it can be tedious and costly to rate many samples manually. In many cases, as in math or classification use cases, it may be possible to establish *ground truth* (reference answers to test cases) in order to programmatically rate the results, allowing you to scale up considerably your testing and monitoring efforts. The following is not an exhaustive list, for there are many motivations for evaluating your prompt programmatically:

- *Cost*: Prompts that use a lot of tokens, or only work with more expensive models, might be impractical for production use.
- *Latency*: Equally the more tokens there are, or the larger the model required, the longer it takes to complete a task, which can harm user experience.
- *Calls*: Many AI systems require multiple calls in a loop to complete a task, which can seriously slow down the process.
- *Performance*: Implement some form of external feedback system, for example a physics engine or other model for predicting real-world results.
- *Classification*: Determine how often a prompt correctly labels given text, using another AI model or rules-based labeling.
- *Reasoning*: Work out which instances the AI fails to apply logical reasoning or gets the math wrong vs reference cases.
- *Hallucinations*: See how frequently you encouner hallucinations, as measured by invention of new terms not included in the prompt's context.

- *Safety*: Flag any scenarios where the system might return unsafe or undesirable results using a safety filter or detection system.
- *Refusals*: Find out how often the system incorrectly refuses to fulfill a reasonable user request by flagging known refusal language.
- *Adversarial*: Make the prompt robust against known prompt injection attacks, that can get the model to run undesirable prompts instead of what you programmed.
- *Similarity*: use shared words and phrases (BLEU or ROGUE) or vector distance (explained in Chapter 5) to measure similarity between generated and reference text.

Once you start rating which examples were good, you can more easily update the examples used in your prompt, as a way to continuously make your system smarter over time. The data from this feedback can also feed into examples for fine-tuning, which starts to beat prompt engineering once you can supply a few thousand examples, as is shown in Figure 1-13.

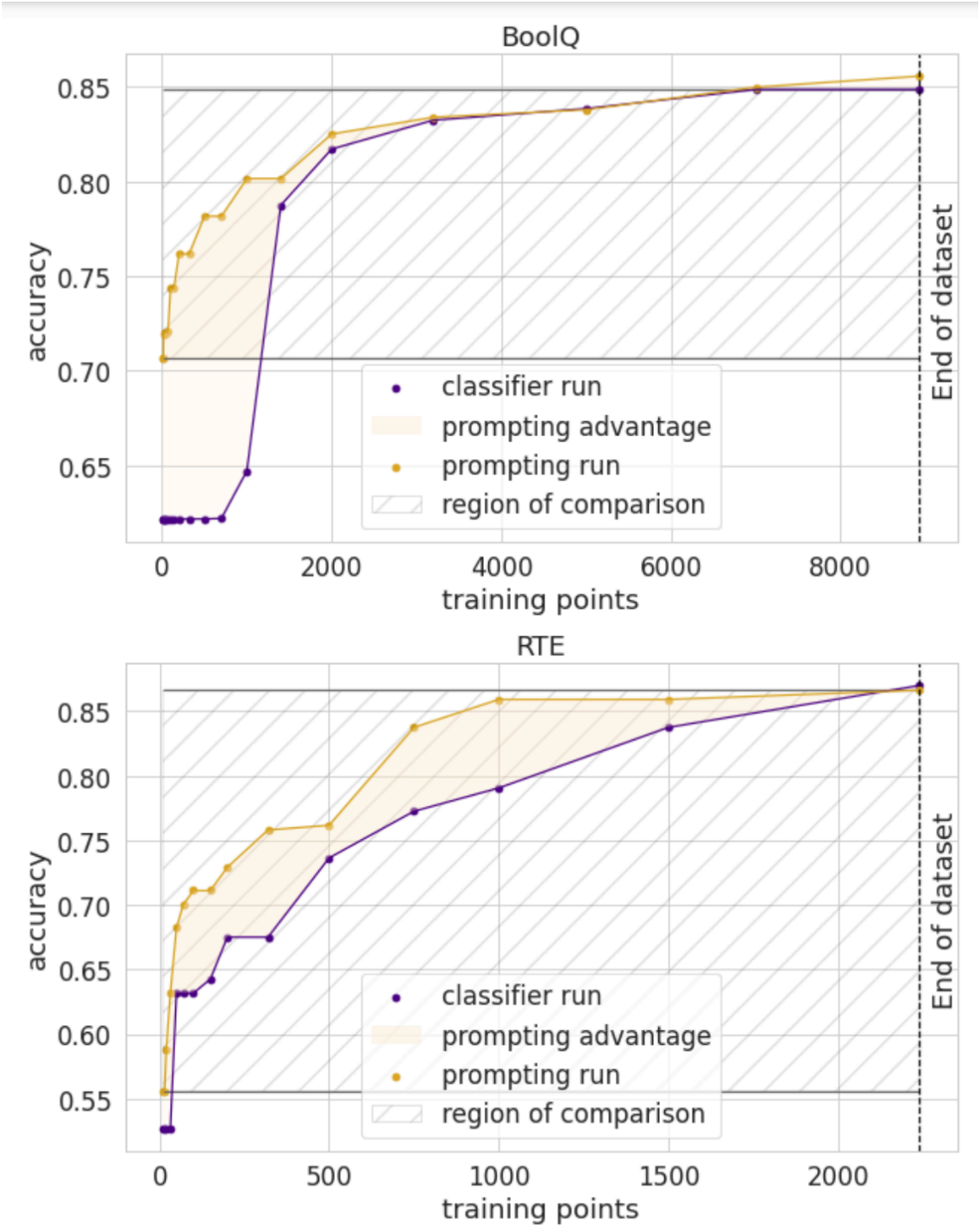

Figure 1-13. How Many Data Points is a Prompt Worth?

Graduating from thumbs up or down, you can implement a 3, 5, or 10 point rating system in order to get more fine-grained feedback on the quality of your prompts. It's also possible to determine aggregate relative performance through comparing responses side by side, rather than looking at responses one at a time. From this you can construct a fair across-model comparison using an *elo rating*, as is popular in chess and used in the ChatBot Arena by lmsys.org.

For image generation, evaluation usually takes the form of *permutation* prompting, where you input multiple directions or formats and generate an image for each combination. Images can than be scanned or later arranged in a grid to show the effect that different elements of the prompt can have on the final image.

Input:

```
{stock photo, oil painting, illustration} of business
meeting of {four, eight} people watching on white MacBook on
top of glass-top table
```

In Midjourney this would be compiled into six different prompts, one for every combination of the three formats (stock photo, oil painting, illustration) and two numbers of people (four, eight).

Input:

```
1. stock photo of business meeting of four people watching
on white MacBook on top of glass-top table

2. stock photo of business meeting of eight people watching
on white MacBook on top of glass-top table

3. oil painting of business meeting of four people watching
on white MacBook on top of glass-top table

4. oil painting of business meeting of eight people watching
on white MacBook on top of glass-top table

5. illustration of business meeting of four people watching
on white MacBook on top of glass-top table

6. illustration of business meeting of eight people watching
on white MacBook on top of glass-top table
```

Each prompt generates its own four images as usual, which makes the output a little harder to see. We have selected one from each prompt to

upscale, and then put them together in a grid, shown as Figure 1-14. You'll notice that the model doesn't always get the correct number of people (generative AI models are surprisingly bad at math) but it has correctly inferred the general intention by adding more people to the photos on the right than the left.

The output is shown in Figure 1-14.

Figure 1-14. Prompt permutations grid

With models that have APIs like Stable Diffusion, you can more easily manipulate the photos and display them in a grid format for easy scanning. You can also manipulate the random seed of the image, in order to fix a