Text completion

Learn how to generate or manipulate text

Introduction

The completions endpoint can be used for a wide variety of tasks. It provides a simple but powerful interface to any of our models. You input some text as a prompt, and the model will generate a text completion that attempts to match whatever context or pattern you gave it. For example, if you give the API the prompt, "As Descartes said, I think, therefore", it will return the completion " I am" with high probability.

The best way to start exploring completions is through our Playground. It's simply a text box where you can submit a prompt to generate a completion. To try it yourself, open this example in Playground:

```
Write a tagline for an ice cream shop.
```

Once you submit, you'll see something like this:

```
Write a tagline for an ice cream shop.
We serve up smiles with every scoop!
```

The actual completion you see may differ because the API is non-deterministic by default. This means that you might get a slightly different completion every time you call it, even if your prompt stays the same. Setting temperature to 0 will make the outputs mostly deterministic, but a small amount of variability may remain.

This simple text-in, text-out interface means you can "program" the model by providing instructions or just a few examples of what you'd like it to do. Its success generally depends on the complexity of the task and quality of your prompt. A good rule of thumb is to think about how you would write a word problem for a middle-schooler to solve. A well-written prompt provides enough information for the model to know what you want and how it should respond.

This guide covers general prompt design best practices and examples. To learn more about working with code using our Codex models, visit our code guide.

Keep in mind that the default models' training data cuts off in 2021, so they may not have knowledge of current events. We plan to add more continuous training in the future.

Prompt design

Basics

Our models can do everything from generating original stories to performing complex text analysis. Because they can do so many things, you have to be explicit in describing what you want. Showing, not just telling, is often the secret to a good prompt.

There are three basic guidelines to creating prompts:

Show and tell. Make it clear what you want either through instructions, examples, or a combination of the two. If you want the model to rank a list of items in alphabetical order or to classify a paragraph by sentiment, show it that's what you want.

Provide quality data. If you're trying to build a classifier or get the model to follow a pattern, make sure that there are enough examples. Be sure to proofread your examples — the model is usually smart enough to see through basic spelling mistakes and give you a response, but it also might assume this is intentional and it can affect the response.

Check your settings. The temperature and top_p settings control how deterministic the model is in generating a response. If you're asking it for a response where there's only one right answer, then you'd want to set these lower. If you're looking for more diverse responses, then you might want to set them higher. The number one mistake people use with these settings is assuming that they're "cleverness" or "creativity" controls.

Troubleshooting

If you're having trouble getting the API to perform as expected, follow this checklist:

1. Is it clear what the intended generation should be?
2. Are there enough examples?
3. Did you check your examples for mistakes? (The API won't tell you directly)
4. Are you using temperature and top_p correctly?

Classification

To create a text classifier with the API, we provide a description of the task and a few examples. In this example, we show how to classify the sentiment of Tweets.

```
Decide whether a Tweet's sentiment is positive, neutral, or negative.

Tweet: I loved the new Batman movie!
Sentiment:
```

Open in Playground ↗

It's worth paying attention to several features in this example:

1. **Use plain language to describe your inputs and outputs.** We use plain language for the input "Tweet" and the expected output "Sentiment." As a best practice, start with plain language descriptions. While you can often use shorthand or keys to indicate the input and output, it's best to start by being as descriptive as possible and then working backwards to remove extra words and see if performance stays consistent.
2. **Show the API how to respond to any case.** In this example, we include the possible sentiment labels in our instruction. A neutral label is important because there will be many cases where even a human would have a hard time determining if something is positive or negative, and situations where it's neither.
3. **You need fewer examples for familiar tasks.** For this classifier, we don't provide any examples. This is because the API already has an understanding of sentiment and the concept of a Tweet. If you're building a classifier for something the API might not be familiar with, it might be necessary to provide more examples.

Improving the classifier's efficiency

Now that we have a grasp of how to build a classifier, let's take that example and make it even more efficient so that we can use it to get multiple results back from one API call.

```
Classify the sentiment in these tweets:

1. "I can't stand homework"
2. "This sucks. I'm bored 😠"
3. "I can't wait for Halloween!!!"
4. "My cat is adorable 💔💔"
5. "I hate chocolate"
```

```
Tweet sentiment ratings:
```

Open in Playground ↗

We provide a numbered list of Tweets so the API can rate five (and even more) Tweets in just one API call.

It's important to note that when you ask the API to create lists or evaluate text you need to pay extra attention to your probability settings (Top P or Temperature) to avoid drift.

1. Make sure your probability setting is calibrated correctly by running multiple tests.
2. Don't make your list too long or the API is likely to drift.

Generation

One of the most powerful yet simplest tasks you can accomplish with the API is generating new ideas or versions of input. You can ask for anything from story ideas, to business plans, to character descriptions and marketing slogans. In this example, we'll use the API to create ideas for using virtual reality in fitness.

```
Brainstorm some ideas combining VR and fitness:
```

Open in Playground ↗

If needed, you can improve the quality of the responses by including some examples in your prompt.

Conversation

The API is extremely adept at carrying on conversations with humans and even with itself. With just a few lines of instruction, we've seen the API perform as a customer service chatbot that intelligently answers questions without ever getting flustered or a wise-cracking conversation partner that makes jokes and puns. The key is to tell the API how it should behave and then provide a few examples.

Here's an example of the API playing the role of an AI answering questions:

```
The following is a conversation with an AI assistant. The assistant is
helpful, creative, clever, and very friendly.

Human: Hello, who are you?
AI: I am an AI created by OpenAI. How can I help you today?
Human:
```

Open in Playground ↗

This is all it takes to create a chatbot capable of carrying on a conversation. Underneath its simplicity, there are several things going on that are worth paying attention to:

1. **We tell the API the intent but we also tell it how to behave.** Just like the other prompts, we cue the API into what the example represents, but we also add another key detail: we give it explicit instructions on how to interact with the phrase "The assistant is helpful, creative, clever, and very friendly."

   Without that instruction the API might stray and mimic the human it's interacting with and become sarcastic or some other behavior we want to avoid.

2. **We give the API an identity.** At the start we have the API respond as an AI assistant. While the API has no intrinsic identity, this helps it respond in a way that's as close to the truth as possible. You can use identity in other ways to create other kinds of chatbots. If you tell the API to respond as a woman who works as a research scientist in biology, you'll get intelligent and thoughtful comments from the API similar to what you'd expect from someone with that background.

In this example we create a chatbot that is a bit sarcastic and reluctantly answers questions:

```
Marv is a chatbot that reluctantly answers questions with sarcastic
responses:

You: How many pounds are in a kilogram?
Marv: This again? There are 2.2 pounds in a kilogram. Please make a note
of this.
You: What does HTML stand for?
Marv: Was Google too busy? Hypertext Markup Language. The T is for try to
ask better questions in the future.
You: When did the first airplane fly?
Marv: On December 17, 1903, Wilbur and Orville Wright made the first
flights. I wish they'd come and take me away.
```

```
You: What is the meaning of life?
Marv: I'm not sure. I'll ask my friend Google.
You: Why is the sky blue?
```

Open in Playground ↗

To create an amusing and somewhat helpful chatbot, we provide a few examples of questions and answers showing the API how to reply. All it takes is just a few sarcastic responses, and the API is able to pick up the pattern and provide an endless number of snarky responses.

Transformation

The API is a language model that is familiar with a variety of ways that words and characters can be used to express information. This ranges from natural language text to code and languages other than English. The API is also able to understand content on a level that allows it to summarize, convert and express it in different ways.

Translation

In this example we show the API how to convert from English to French, Spanish, and Japanese:

```
Translate this into French, Spanish and Japanese:

What rooms do you have available?
```

Open in Playground ↗

This example works because the API already has a grasp of these languages, so there's no need to try to teach them.

If you want to translate from English to a language the API is unfamiliar with, you'd need to provide it with more examples or even fine-tune a model to do it fluently.

Conversion

In this example we convert the name of a movie into emoji. This shows the adaptability of the API to picking up patterns and working with other characters.

```
Convert movie titles into emoji.

Back to the Future: 👨👴🚗🕒
Batman: 🤵🦇
Transformers: 🚗🤖
Star Wars:
```

Open in Playground ↗

Summarization

The API is able to grasp the context of text and rephrase it in different ways. In this example, we create an explanation a child would understand from a longer, more sophisticated text passage. This illustrates that the API has a deep grasp of language.

```
Summarize this for a second-grade student:

Jupiter is the fifth planet from the Sun and the largest in the Solar
System. It is a gas giant with a mass one-thousandth that of the Sun, but
two-and-a-half times that of all the other planets in the Solar System
combined. Jupiter is one of the brightest objects visible to the naked
eye in the night sky, and has been known to ancient civilizations since
before recorded history. It is named after the Roman god Jupiter.[19]
When viewed from Earth, Jupiter can be bright enough for its reflected
light to cast visible shadows,[20] and is on average the third-brightest
natural object in the night sky after the Moon and Venus.
```

Open in Playground ↗

Completion

While all prompts result in completions, it can be helpful to think of text completion as its own task in instances where you want the API to pick up where you left off. For example, if given this prompt, the API will continue the train of thought about vertical farming. You can lower the temperature setting to keep the API more focused on the intent of the prompt or increase it to let it go off on a tangent.

```
Vertical farming provides a novel solution for producing food locally,
reducing transportation costs and
```

Open in Playground ↗

This next prompt shows how you can use completion to help write React components. We send some code to the API, and it's able to continue the rest because it has an understanding of the React library. We recommend using our Codex models for tasks that involve understanding or generating code. To learn more, visit our code guide.

```
import React from 'react';
const HeaderComponent = () => (
```

Open in Playground ↗

Factual responses

The API has a lot of knowledge that it's learned from the data that it was been trained on. It also has the ability to provide responses that sound very real but are in fact made up. There are two ways to limit the likelihood of the API making up an answer.

1. **Provide a ground truth for the API.** If you provide the API with a body of text to answer questions about (like a Wikipedia entry) it will be less likely to confabulate a response.
2. **Use a low probability and show the API how to say "I don't know".** If the API understands that in cases where it's less certain about a response that saying "I don't know" or some variation is appropriate, it will be less inclined to make up answers.

In this example we give the API examples of questions and answers it knows and then examples of things it wouldn't know and provide question marks. We also set the probability to zero so the API is more likely to respond with a "?" if there is any doubt.

```
Q: Who is Batman?
A: Batman is a fictional comic book character.

Q: What is torsalplexity?
A: ?

Q: What is Devz9?
```

```
A: ?

Q: Who is George Lucas?
A: George Lucas is American film director and producer famous for
creating Star Wars.

Q: What is the capital of California?
A: Sacramento.

Q: What orbits the Earth?
A: The Moon.

Q: Who is Fred Rickerson?
A: ?

Q: What is an atom?
A: An atom is a tiny particle that makes up everything.

Q: Who is Alvan Muntz?
A: ?

Q: What is Kozar-09?
A: ?

Q: How many moons does Mars have?
A: Two, Phobos and Deimos.

Q:
```

Open in Playground ↗

Inserting text Beta

The completions endpoint also supports inserting text within text by providing a suffix prompt in addition to the prefix prompt. This need naturally arises when writing long-form text, transitioning between paragraphs, following an outline, or guiding the model towards an ending. This also works on code, and can be used to insert in the middle of a function or file. Visit our code guide to learn more.

To illustrate how important suffix context is to our ability to predict, consider the prompt, "Today I decided to make a big change." There's many ways one could imagine completing the sentence. But if we now supply the ending of the story: "I've gotten many compliments on my new hair!", the intended completion becomes clear.

```
I went to college at Boston University. After getting my degree, I
decided to make a change. A big change!

I packed my bags and moved to the west coast of the United States.

Now, I can't get enough of the Pacific Ocean!
```

By providing the model with additional context, it can be much more steerable. However, this is a more constrained and challenging task for the model.

Best practices

Inserting text is a new feature in beta and you may have to modify the way you use the API for better results. Here are a few best practices:

Use max_tokens > 256. The model is better at inserting longer completions. With too small max_tokens, the model may be cut off before it's able to connect to the suffix. Note that you will only be charged for the number of tokens produced even when using larger max_tokens.

Prefer finish_reason == "stop". When the model reaches a natural stopping point or a user provided stop sequence, it will set finish_reason as "stop". This indicates that the model has managed to connect to the suffix well and is a good signal for the quality of a completion. This is especially relevant for choosing between a few completions when using n > 1 or resampling (see the next point).

Resample 3-5 times. While almost all completions connect to the prefix, the model may struggle to connect the suffix in harder cases. We find that resampling 3 or 5 times (or using best_of with k=3,5) and picking the samples with "stop" as their finish_reason can be an effective way in such cases. While resampling, you would typically want a higher temperatures to increase diversity.

Note: if all the returned samples have finish_reason == "length", it's likely that max_tokens is too small and model runs out of tokens before it manages to connect the prompt and the suffix naturally. Consider increasing max_tokens before resampling.

Try giving more clues. In some cases to better help the model's generation, you can provide clues by giving a few examples of patterns that the model can follow to decide a natural place

to stop.

```
How to make a delicious hot chocolate:
1. Boil water
2. Put hot chocolate in a cup
3. Add boiling water to the cup
4. Enjoy the hot chocolate
```

```
1. Dogs are loyal animals.
2. Lions are ferocious animals.
3. Dolphins are playful animals.
4. Horses are majestic animals.
```

Editing text Alpha

The edits endpoint can be used to edit text, rather than just completing it. You provide some text and an instruction for how to modify it, and the `text-davinci-edit-001` model will attempt to edit it accordingly. This is a natural interface for translating, editing, and tweaking text. This is also useful for refactoring and working with code. Visit our code guide to learn more. During this initial beta period, usage of the edits endpoint is free.

Examples

INPUT

```
GPT-3 is a very nice AI
That's pretty good at writing replies
When it's asked a question
It gives its suggestion
This is a poem it made that rhymes
```

OUTPUT

```
I am a very nice AI
I am pretty good at writing replies
When I am asked a question
I give my suggestion
This is a poem I made that rhymes
```

INSTRUCTIONS