Classes

Auto Steer

Auto Steering allows you to automatically generate feature interventions based on natural language descriptions of desired behaviors. This provides an easy way to steer model outputs without manually selecting features.

Basic Usage

The simplest way to use Auto Steering is with the `AutoSteer` method:

Code Output

```
# Create automatic feature edits for desired behavior
edits = client.features.AutoSteer(
    specification="be funny",  # Natural language description
    model=variant,  # Model variant to use
)

# Apply the edits to your variant
variant.set(edits)

# The model will now attempt to be funnier
response = client.chat.completions.create(
    messages=[{"role": "user", "content": "Tell me a story"}],
    model=variant
)
print(response.choices[0].message["content"])
```

How It Works

Under the hood, Auto Steer:

1. Generates contrastive examples of content with and without the desired behavior
2. Identifies the most relevant features that distinguish the behavior
3. Determines optimal feature values to encourage the desired behavior
4. Creates a set of feature edits that can be applied to the model

Advanced Usage

Combining with Manual Edits

Auto Steer edits can be combined with manual feature interventions:

```
# Generate automatic edits
auto_edits = client.features.AutoSteer(
    specification="be professional",
    model=variant
)

# Combine with manual feature edits
variant.set({
    **auto_edits,
    manual_feature: 0.8
})
```

Using with Conditionals

Auto Steer can be used within conditional statements:

```
# Generate edits for desired behavior
funny_edits = client.features.AutoSteer(
    specification="be funny",
    model=variant
)

# Apply edits conditionally
variant.set_when(context_feature > 0.5, funny_edits)
```

Best Practices

- Use clear, specific behavior descriptions
- Test generated edits to ensure desired results
- Consider combining multiple Auto Steer edits for complex behaviors
- Adjust number of features based on steering precision needed
- Use with conditionals for context-aware behavior

API Reference

AutoSteer

Generate automatic feature edits based on natural language description.

Parameters:

`specification` `str` `required`

Natural language description of desired behavior

`model` `Union[str, Variant]` `required`

Model or variant to use for generating edits

Returns:

`edits` `dict[Feature, float]`

Dictionary mapping features to their target values

Example:

Auto Steer

```
edits = client.features.AutoSteer(
    specification="be more creative",
```

```
    model=variant
)
```


‹ Conditionals

Classes

Chat Completions

Documentation for the Goodfire Chat API

The Chat API provides methods for interacting with Goodfire's language models in a chat format. The base chat interface is OpenAI-compatible. It supports both streaming and non-streaming completions, as well as logits computation.

Once you have a model variant, you can use it to create chat completions.

Examples

Basic Chat Completion

Code Output

```
# Initialize the client
client = goodfire.Client(
    '{YOUR_API_KEY}',
)

# Create a non-streaming completion
response =  client.chat.completions.create(
    messages=[
        {"role": "user", "content": "What is the capital of France?"}
    ],
    model="meta-llama/Llama-3.3-70B-Instruct"
)

print(response.choices[0].message["content"])
```

Streaming Chat Completion

Streaming Chat Completion | Output Poem


```python
# Stream the response tokens
for chunk in client.chat.completions.create(
    messages=[
        {"role": "user", "content": "Write a short poem"}
    ],
    model="meta-llama/Llama-3.3-70B-Instruct",
    stream=True
):
    print(chunk.choices[0].delta.content, end="")
```

Using with Model Variants

Using with Model Variants | Piratey Output

```python
# Create a variant with feature modifications
variant = goodfire.Variant("meta-llama/Llama-3.3-70B-Instruct")
pirate_features = client.features.search(
    "talk like a pirate",
    model=variant,
    top_k=1
)
variant.set(pirate_features[0], 0.5)

# Use the variant in chat completion
for token in client.chat.completions.create(
    messages=[
        {"role": "user", "content": "Tell me about the ocean."}
    ],
    model=variant,
    stream=True,
    max_completion_tokens=100,
):
    print(token.choices[0].delta.content, end="")
```

Computing Token Probabilities

Token Probabilities | Logit Output

```python
logits = client.chat.logits(
    messages=[
        {"role": "user", "content": "The capital of France is"}
    ],
    model="meta-llama/Llama-3.3-70B-Instruct",
    filter_vocabulary=["Paris", "London", "Berlin"]
)
print(logits.logits)
```

Methods

create()

Create a chat completion with the model.

Parameters:

`messages` `list[ChatMessage]` required

List of messages in the conversation. Each message should have `role` ("user", "assistant", or "system") and `content` fields.

`model` `Union[str, VariantInterface]` required

Model identifier or variant to use for completion

`stream` `bool` default: "False"

Whether to stream the response tokens

`max_completion_tokens` `Optional[int]` default: "2048"

Maximum number of tokens to generate

`top_p` `float` default: "0.9"

Nucleus sampling parameter

---

temperature `float` default: "0.6"

Sampling temperature

stop `Optional[Union[str, list[str]]]`

Sequences where the API will stop generating further tokens

seed `Optional[int]` default: "42"

Random seed for reproducible outputs

system_prompt `str`

System prompt to prepend to the conversation

Returns:

- If `stream=False`: `ChatCompletion` object
- If `stream=True`: Iterator of `StreamingChatCompletionChunk` objects

Examples:

Non-streaming completion:

Basic Chat Completion

```
response =  client.chat.completions.create(
    messages=[
        {"role": "user", "content": "What is the capital of France?"}
    ],
    model="meta-llama/Llama-3.3-70B-Instruct"
)
print(response.choices[0].message["content"])
```

Streaming completion:

Streaming Chat Completion

```python
for chunk in client.chat.completions.create(
    messages=[
        {"role": "user", "content": "Write a short poem"}
    ],
    model="meta-llama/Llama-3.3-70B-Instruct",
    stream=True
):
    print(chunk.choices[0].delta.content, end="")
```

logits()

Compute token probabilities for the next token in a conversation.

Parameters:

`messages` `list[ChatMessage]` `required`

List of messages in the conversation

`model` `Union[str, VariantInterface]` `required`

Model identifier or variant to use

`top_k` `Optional[int]`

Limit response to top K most likely tokens

`filter_vocabulary` `Optional[list[str]]`

List of tokens to compute probabilities for

Returns: `LogitsResponse` containing token probabilities

Example:

Token Probabilities

```
logits = client.chat.logits(
    messages=[
        {"role": "user", "content": "The capital of France is"}
    ],
    model="meta-llama/Llama-3.3-70B-Instruct",
    filter_vocabulary=["Paris", "London", "Berlin"]
)
print(logits.logits)
```

Response Objects

ChatCompletion

Response from a non-streaming chat completion.

Properties:

`id` str

Unique identifier for the completion

`object` str

Object type identifier

`created` Optional[int]

Unix timestamp of when the completion was created

`model` str

ID of the model used

`system_fingerprint` str

System fingerprint for the completion

`choices` list[ChatCompletionChoice]

List of completion choices

StreamingChatCompletionChunk

Individual chunk from a streaming chat completion.

Properties:

`id` `str`

Unique identifier for the completion

`object` `str`

Object type identifier

`created` `Optional[int]`

Unix timestamp of when the chunk was created

`model` `str`

ID of the model used

`system_fingerprint` `str`

System fingerprint for the completion

`choices` `list[StreamingChoice]`

List of completion choices in this chunk

LogitsResponse

Response from a logits computation request.

Properties:

`logits` `dict[str, float]`

Dictionary mapping tokens to their probabilities

‹ Client Reference

Variants ›

Classes

Client Reference

Client

The Goodfire SDK provides both synchronous and asynchronous clients for interacting with the Goodfire API.

The examples provided in this guide are for the synchronous client. The asynchronous client is identical but with `await` added to the function calls.

We've broken down the SDK into a few key concepts:

- Chat Completions - For interacting with language models
- Variants - For managing model edits & variants
- Features - For working with interpretable features
- Conditionals - For working with conditional feature interventions
- AutoSteering - For steering models with natural language

Initialization

Sync Async

```
from goodfire import Client

client = Client("your-api-key")
```

Classes

Conditionals

Documentation for working with conditional feature interventions

Conditionals allow you to define dynamic feature interventions that are applied based on the activation patterns of other features during model inference. This enables creating more sophisticated steering behaviors that respond to the content being generated.

Before using the Conditionals API, you'll need to find the features you want to intervene on, and a model variant

Examples

Basic Conditional Intervention

Apply pirate-themed features only when whale-related content is detected:

Basic Conditional Intervention | Output text

```
variant.reset()
# Find relevant features
whale_feature = client.features.search(
    "whales", model=variant, top_k=1
)

pirate_features = client.features.search(
    "talk like a pirate", model=variant, top_k=5
)

# Set up conditional intervention
variant.set_when(whale_feature > 0.75, {
    pirate_features[0]: 0.4
})
```

```python
# The model will now talk like a pirate when discussing whales
response = client.chat.completions.create(
    messages=[{"role": "user", "content": "Tell me about whales."}],
    model=variant
)

print(response.choices[0].message["content"])
```

Aborting Generation

Stop generation if certain content is detected:

Abort example   Abort output

```python
# Abort if whale features are too strong
variant.abort_when(whale_feature > 0.75)

try:
    response = client.chat.completions.create(
        messages=[{"role": "user", "content": "Tell me about whales."}],
        model=variant
    )
except goodfire.exceptions.InferenceAbortedException:
    print("Generation aborted due to whale content")
```

Auto-Generated Conditionals

Use natural language to automatically generate conditional statements:

Auto-Generated Conditionals example

```python
#create a variant
variant = goodfire.Variant("meta-llama/Llama-3.3-70B-Instruct")
# Generate conditional based on description - this will create conditions for
conditional= client.features.AutoConditional(
    "when the model talks about whales and penguins",
    model=variant
)
```

```python
# Get pirate  feature
pirate_feature = client.features.search(
    "talk like a pirate", model=variant, top_k=1
)
# Make the model talk like a pirate when it talks about both whales and penguin
variant.set_when(conditional, {
    pirate_feature[0]: 0.9
})

response = client.chat.completions.create(
    messages=[{"role": "user", "content": "Tell me about whales and penguins!"}
    model=variant
)

print(response.choices[0].message["content"])
```

Creating Conditionals

Comparison Operators

You can create conditionals by comparing features or feature groups with numeric values or other features using standard comparison operators. This creates a Conditional object that can be used in steering behaviors.

```python
# Compare feature to numeric value
condition = feature > 0.75

# Compare feature group to numeric value
condition = feature_group >= 0.5

# Compare features to each other
condition = feature1 < feature2
```

Supported operators:

- `==` (equal)
- `!=` (not equal)
- `<` (less than)

- `<=` (less than or equal)
- `>` (greater than)
- `>=` (greater than or equal)

Logical Operators

Multiple conditions can be combined using logical operators to create a ConditionalGroup:

```
# AND operator
condition = (feature1 > 0.5) & (feature2 < 0.3)

# OR operator
condition = (feature1 > 0.5) | (feature2 > 0.5)
```

Using Conditionals

set_when()

Apply feature interventions when a condition is met.

Parameters:

`condition` `ConditionalGroup` `required`

The ConditionalGroup that triggers the intervention

`values` `Union[FeatureEdits, dict[Union[Feature, FeatureGroup], float]]` `required`

Feature edits to apply when condition is met

Returns: None

Example:

```
# Set pirate features when whale features are detected
variant.set_when(whale_feature > 0.75, {
```

```
    pirate_features[0]: 0.5
})
```

Classes › Conditionals

abort_when()

Abort inference when a condition is met by raising an InferenceAbortedException.

Parameters:

condition `ConditionalGroup` required

The ConditionalGroup that triggers the abort

Returns: None

Example:

```
# Abort if whale features are too strong
variant.abort_when(whale_feature > 0.75)

try:
    response = client.chat.completions.create(
        messages=[{"role": "user", "content": "Tell me about whales."}],
        model=variant
    )
except goodfire.exceptions.InferenceAbortedException:
    print("Generation aborted due to whale content")
```

handle_when()

Register a custom handler function to be called when a condition is met.

Parameters:

condition `ConditionalGroup` required

The ConditionalGroup that triggers the handler

---

handler `Callable[[InferenceContext], None]` required

Function that takes an InferenceContext and returns None

Returns: None

Example:

```
def custom_handler(context: InferenceContext):
    # Custom handling logic
    pass

variant.handle_when(whale_feature > 0.5, custom_handler)
```

AutoConditional

The AutoConditional utility helps automatically generate conditional statements based on natural language descriptions.

Parameters:

specification `str` required

Natural language description of the desired condition

---

model `Union[str, Variant]` required

Model to use for generating conditions

Returns:

conditional `ConditionalGroup`

Generated ConditionalGroup

Example:

```
# Generate conditional based on description - this will create conditions   r
conditional= client.features.AutoConditional(
    "when the model talks about whales and penguins",
    model=variant
)
# Get pirate  feature
pirate_feature = client.features.search(
    "talk like a pirate", model=variant, top_k=1
)
# Make the model talk like a pirate when it talks about both whales and penguin
variant.set_when(conditional, {
    pirate_feature[0]: 0.9
})
```

Best Practices

- Use conditional interventions to create context-aware steering behaviors
- Combine multiple conditions with logical operators for more precise control
- Handle aborted inferences gracefully in your application
- Test conditions thoroughly to ensure desired behavior
- Consider using AutoConditional for quick prototyping

Classes

ConditionalGroup

A group of conditions combined with logical operators.

Show Properties

Conditional

A single conditional expression comparing features.

Show Properties

InferenceContext

Context object containing information about the current inference state.

Classes › Conditionals

Show Properties

‹ Features

Auto Steer ›

Notebooks

Decision Trees

Use features to label data and build decision trees

By inspecting feature activations on labelled datasets, we can build highly accurate and interpretable decision trees for classification tasks.

In this example, we classify financial news into positive and negative sentiments by building a decision tree.

Open in Colab

‹ Quickstart

Jailbreak Resistance ›

Notebooks

Dynamic Prompts

By using Feature Inspection and Contrasting we can build a model that dynamically changes its instructions based on the user's prompt.

In this example, we'll build a model that dynamically changes its instructions based on if the user is asking for code to be written or not.

Open in Colab

Classes

Features

Documentation for the Goodfire Features API

Accessing Features

Before using the Features API, you'll need a model variant:

```
variant = Variant("meta-llama/Llama-3.3-70B-Instruct")
```

You can then access features through the client's features interface. For example, to search for features:

Search for features | Features

```
# Search for features related to a concept
features = client.features.search(
    "angry",
    model=variant,
    top_k=5
)

# Print the found features
print(features)
```

Or inspect feature activations in text:

Inspect feature activations | Features activated in text

```
# Analyze how features activate in text
inspector = client.features.inspect(
```

```
    [
        {"role": "user", "content": "What do you think about pirates and whal
        {"role": "assistant", "content": "I think pirates are cool and whales a
    ],
    model=variant
)

# Get top activated features
for activation in inspector.top(k=5):
    print(f"{activation.feature.label}: {activation.activation}")
```

The Features API provides methods for working with interpretable features of language models. Features represent learned patterns in model behavior that can be analyzed and modified.

Methods

neighbors()

Get the nearest neighbors of a feature or group of features.

Parameters:

`features` `Feature | FeatureGroup` `required`

Feature or group of features to find neighbors for

`model` `str | VariantInterface` `required`

Model identifier or variant interface

`top_k` `int` `default: 10`

Number of neighbors to return

Returns: `FeatureGroup`

Example:

Getting neighbors of a feature

```python
# Get neighbors of a feature
neighbors = client.features.neighbors(
    feature,
    model="meta-llama/Llama-3.3-70B-Instruct",
    top_k=10
)

# Print neighbor labels
for neighbor in neighbors:
    print(neighbor.label)
```

search()

Search for features based on semantic similarity to a query string.

Parameters:

`query` `str` `required`

Search string to compare against feature labels

`model` `str | VariantInterface` `required`

Model identifier or variant interface

`top_k` `int` `default: 10`

Number of features to return

Returns: `FeatureGroup` - Collection of matching features

Example:

Search features

```python
# Search for features related to writing style
features = client.features.search(
```

```
    "formal writing style",
    model="meta-llama/Llama-3.3-70B-Instruct",
    top_k=10
)

# Print features
for feature in features:
    print(feature.label)
```

inspect()

Analyzes how features are activated across the input messages.

Parameters:

messages `list[ChatMessage]` required

Messages to analyze

---

model `str | VariantInterface` required

Model identifier or variant interface

---

features `Feature | FeatureGroup | None`

Optional specific features to analyze. If None, inspects all features.

---

aggregate_by `str` default: `"frequency"`

Method to aggregate feature activations across tokens: - “frequency”: Count of tokens where feature is active - “mean”: Mean activation value across tokens - “max”: Maximum activation value across tokens - “sum”: Sum of activation values across tokens

---

Returns: `ContextInspector` - An inspector object that provides methods for analyzing and visualizing how features are activated in the given context.

Example:

Inspect feature activations

```
# Analyze how features activate in text
inspector = client.features.inspect(
    [
        {"role": "user", "content": "What do you think about pirates and whales
        {"role": "assistant", "content": "I think pirates are cool and whales a
    ],
    model=variant
)

# Get top activated features
for activation in inspector.top(k=5):
    print(f"{activation.feature.label}: {activation.activation}")
```

contrast()

Identify features that differentiate between two conversation datasets.

Parameters:

dataset_1 `list[list[ChatMessage]]` `required`

First dataset of conversations

---

dataset_2 `list[list[ChatMessage]]` `required`

Second dataset of conversations

---

model `str | VariantInterface` `required`

Model identifier or variant interface

---

top_k `int` `default: 5`

Number of top features to return for each dataset

Returns: `tuple[FeatureGroup, FeatureGroup]` - Two FeatureGroups containing:

- Features steering towards dataset_1

Example:

Get constrast features

```
# Compare formal vs informal conversations
dataset_1 = [[
    {"role": "user", "content": "Hi how are you?"},
    {"role": "assistant", "content": "I'm doing well..."}
]]
dataset_2 = [[
    {"role": "user", "content": "Hi how are you?"},
    {"role": "assistant", "content": "Arr my spirits be high..."}
]]

formal_features, informal_features =  client.features.contrast(
    dataset_1=dataset_1,
    dataset_2=dataset_2,
    model="meta-llama/Llama-3.3-70B-Instruct",
    top_k=5
)
```

rerank()

Rerank a set of features based on a query.

Parameters:

`features` FeatureGroup required

Features to rerank

`query` str required

Query to rerank features by

`model` str | VariantInterface required

Model identifier or variant interface

`top_k` `int` `default: 10`

Number of top features to return

Returns: `FeatureGroup`

Example:

Rerank

```
# Rerank features based on relevance to "writing style"
reranked = client.features.rerank(
    features=formal_features,
    query="writing style",
    model="meta-llama/Llama-3.3-70B-Instruct",
    top_k=10
)
```

activations()

Retrieves feature activation values for each token in the input messages.

Parameters:

`messages` `list[ChatMessage]` `required`

Messages to analyze

`model` `str | VariantInterface` `required`

Model identifier or variant interface

`features` `Feature | FeatureGroup | None`

Optional specific features to analyze. If None, analyzes all features.

Returns: `NDArray[np.float64]` - Sparse activation matrix of shape [n_tokens, n_features] where each element represents the activation strength of a feature at a specific token. Most values are zero due to sparsity.

Example:

Get activation matrix

```
# Get activation matrix for a conversation
matrix =  client.features.activations(
    messages=[{"role": "user", "content": "Hello world"}],
    model="meta-llama/Llama-3.3-70B-Instruct"
)
```

lookup()

Retrieves details for a list of features by their indices.

Parameters:

`indices` `list[int]` `required`

List of feature indices to fetch

`model` `str | VariantInterface` `required`

Model identifier or variant interface

Returns: `dict[int, Feature]` - Mapping of feature index to Feature object

list()

Retrieves details for a list of features by their UUIDs.

Parameters:

`ids` `list[str]` `required`

List of feature UUIDs to fetch

Returns: `FeatureGroup` - Collection of Feature objects

Classes

Feature

A class representing a human-interpretable “feature” - a model’s conceptual neural unit. Features can be combined into groups and compared using standard operators.

Show Properties

FeatureGroup

A collection of Feature instances with group operations.

Show Properties

ConditionalGroup

Groups multiple conditions with logical operators.

Show Properties

FeatureActivation

Represents the activation of a feature.

Show Properties

ContextInspector

Analyzes feature activations in text.

Variants

Conditionals

Notebooks

Jailbreak Resistance

By using Feature Activations and Contrastive Search we can build a jailbreak resistant model.

Through this approach we were able to drastically lower the ability to jailbreak the model, using jailbreak prompts from the StrongREJECT dataset.

Open in Colab

Notebooks

On Demand RAG

By using conditional interventions we can build a “on demand RAG” system.

When we see the user is asking about something that might need more data, we can abort the request and get more data from an external source.

In this example, we’ll build a model that can insert brand deal data into the response when the user is asking about products.

For example, if the user asks about drinks, and we sponsor Coca Cola, we can stop the request, get RAG data on brand deals and pass it back into the model.

Open in Colab

Get Started

Quickstart

Get started with the Goodfire Ember SDK

Prerequisite: You'll need a Goodfire API key to follow this guide. Get one through our platform or contact support.

Open in Colab

Quickstart

Ember is a hosted API/SDK that lets you shape AI model behavior by directly controlling a model's internal units of computation, or "features". With Ember, you can modify features to precisely control model outputs, or use them as building blocks for tasks like classification.

In this quickstart, you'll learn how to:

- Find features that matter for your specific needs
- Edit features to create model variants
- Discover which features are active in your data
- Save and load your model variants

Code

```
pip install goodfire
```

You can get an API key through our platform

Initialize the SDK

Code

```
import goodfire
client = goodfire.Client(api_key=GOODFIRE_API_KEY)
# Instantiate a model variant.
variant = goodfire.Variant("meta-llama/Llama-3.3-70B-Instruct")
```

Our sampling API is OpenAI compatible, making it easy to integrate.

Generation Code | Generation Output

```
for token in client.chat.completions.create(
    [{"role": "user", "content": "Hi, how are you?"}],
    model=variant,
    stream=True,
    max_completion_tokens=100,
):
    print(token.choices[0].delta.content, end="")
```

Editing features to create model variants

How to find relevant features for edits

There are three ways to find features you may want to modify:

- Auto Steer: Simply describe what you want, and let the API automatically select and adjust feature weights
- Feature Search: Find features using semantic search

Auto Steer

Auto steering automatically finds and adjusts feature weights to achieve your desired behavior. Simply provide a short prompt describing what you want, and autosteering will:

- Find the relevant features
- Set appropriate feature weights
- Return a FeatureEdits object that you can set directly

Auto Steer Code | Auto Steer Output

```
edits = client.features.AutoSteer(
    specification="be funny",  # or your desired behavior
    model=variant,
)
variant.set(edits)
print(edits)
```

Now that we have a few funny edits, let's see how the model responds!

Auto Steer Generation Code | Auto Steer Generation Output

```
for token in client.chat.completions.create(
    [{"role": "user", "content": "Tell me about pirates"}],
    model=variant,
    stream=True,
    max_completion_tokens=120,
):
    print(token.choices[0].delta.content, end="")
```

The model automatically added puns/jokes, even though we didn't specify anything about comedy in our prompt.

```
variant.reset()
```

Feature search helps you explore and discover what capabilities your model has. It can be useful when you want to browse through available features.

Feature Search Code | Feature Search Output

```
funny_features = client.features.search(
    "funny",
    model=variant,
    top_k=10
)
print(funny_features)
```

When setting feature weights manually, start with 0.5 to enhance a feature and -0.3 to ablate a feature. When setting multiple features, you may need to tune down the weights.

Feature set and generation code | Feature set and generation output

```
variant.set(funny_features[0], 0.6)
for token in client.chat.completions.create(
      [
          {"role": "user", "content": "tell me about foxes"}
      ],
      model=variant,
      stream=True,
      max_completion_tokens=100,
  ):
      print(token.choices[0].delta.content, end="")
```

Feel free to play around with the weights and features to see how the model responds.

the group's centroid.

`neighbors()` helps you understand feature relationships beyond just their labels. It can reveal which features might work best for your intended model adjustments.

Nearest Neighbors Code | Nearest Neighbors Output

```
client.features.neighbors(
    funny_features[0],
    model=variant,
    top_k=5
)
```

Now, you can find more features that are similar to other features

Nearest Neighbors | Out

```
client.features.neighbors(
    funny_features[2],
    model=variant,
    top_k=5
)
```

Contrastive Search

Contrastive search lets you discover relevant features in a data-driven way.

Provide two datasets of chat examples:

- dataset_1: Examples of behavior you want to avoid
- dataset_2: Examples of behavior you want to encourage

Examples are paired such that the first example in dataset_1 contrasts the first example in dataset_2, and so on.

This two-step process ensures you get features that are both:

- Mechanistically useful (from contrastive search)
- Aligned with your goals (from reranking)

Let's specify two conversation datasets. The first has a typical helpful assistant response and the second assistant replies in jokes.

Contrastive Search Code | Contrastive Search Output

```
variant.reset()
default_conversation = [
    [
        {
            "role": "user",
            "content": "Hello how are you?"
        },
        {
            "role": "assistant",
            "content": "I am a helpful assistant. How can I help you?"
        }
    ]
]
joke_conversation = [
    [
        {
            "role": "user",
            "content": "Hello how are you?"
        },
        {
            "role": "assistant",
            "content": "What do you call an alligator in a vest? An investigat
        }
    ]
]
helpful_assistant_features, joke_features = client.features.contrast(
```

```
# Let's rerank to surface humor related features
joke_features = client.features.rerank(
    features=joke_features,
    query="funny",
    model=variant,
    top_k=5
)
joke_features
```

We now have a list of features to consider adding. Let's set some plausible-looking ones from `joke_features`.

Note that we could also explore removing some of the helpful_assistant features.

Contrastive Search Set Features Code | Contrastive Search Set Features Output

```
variant.reset()
variant.set(joke_features[0,1], 0.6)
for token in client.chat.completions.create(
    [
        {"role": "user", "content": "Hello. Tell me about whales."}
    ],
    model=variant,
    stream=True,
    max_completion_tokens=100,
):
    print(token.choices[0].delta.content, end="")
```

(Advanced) Conditional logic for feature edits

You can establish relationships between different features (or feature groups) using conditional interventions.

First, let's reset the variant and pick out the funny features.

Now, let's find a features where the model is talking like a pirate.

Pirate Features Code | Pirate Features Output

```
pirate_features = client.features.search(
    "talk like a pirate",
    model=variant,
    top_k=3
)
print(pirate_features)
```

Now, let's set up behaviour so that when the model is talking like a pirate, it will be funny.

Conditional Logic Set Features Code | Conditional Logic Set Features Output

```
variant.set_when(pirate_features[1] > 0.75, {
    funny_features[0]: 0.7,
})
# The model will now try to be funny when talking about pirates
response = client.chat.completions.create(
    messages=[{"role": "user", "content": "talk like a pirate and tell me about
    model=variant
)
print(response.choices[0].message["content"])
```

Say we decide the model isn't very good at pirate jokes. Let's set up behavior to stop generation altogether if the pirate features are too strong.

Abort when Pirate Features are too strong

```
# Abort if pirate features are too strong
variant.abort_when(pirate_features > 0.75)
try:
```

```
    print("Generation aborted due to too much pirate content")
```

If you aren't sure of the features you want to condition on, use AutoConditional with a specified prompt to get back an automatically generated condition.

Auto Conditional Code

```
# Generate auto conditional based on a description. This will automatically
# choose the relevant features and conditional weight
conditional = client.features.AutoConditional(
    "pirates",
    model="meta-llama/Llama-3.3-70B-Instruct",
)

# Apply feature edits when condition is met
variant.set_when(conditional, {
    joke_features[0]: 0.5,
    joke_features[1]: 0.5
})
```

Discover which features are active in your data

Working with a conversation context

You can inspect what features are activating in a given conversation with the `inspect` API, which returns a `context` object.

Say you want to understand what model features are important when the model tells a joke. You can pass in the same joke conversation dataset to the inspect endpoint.

Inspect Code | Inspect Output

```
)
context
```

From the context object, you can access a lookup object which can be used to look at the set of feature labels in the context.

Lookup Code | Lookup Output

```
lookup = context.lookup()
lookup
```

You can select the top `k` activating features in the context, ranked by activation strength. There are features related to jokes and tongue twisters, among other syntactical features.

Inspect Top Features Code | Inspect Top Features Output

```
top_features = context.top(k=10)
top_features
```

You can also inspect individual tokens level feature activations. Let's see what features are active at the punchline token.

Token Activations Code | Token Activations Output

```
print(context.tokens[-3])
token_acts = context.tokens[-3].inspect()
token_acts
```

(Advanced) Look at next token logits

Next Logits Code | Next Logits Output

```
logits.logits
```

Get feature activation vectors for machine learning tasks

To run a machine learning pipeline at the feature level (for instance, for humor detection) you can directly export features using `client.features.activations` to get a matrix or retrieve a sparse vector for a specific `FeatureGroup`.

Matrix Code | Matrix Output

```
activations = client.features.activations(
    messages=joke_conversation[0],
    model=variant,
)
activations
```

Vector Code | Vector Output

```
top_features.vector()
```

Inspecting specific features

There may be specific features whose activation patterns you're interested in exploring. In this case, you can specify features such as *humor_features* and pass that into the `features` argument of `inspect`.

Humor Features Code | Humor Features Output

```
humor_features = client.features.search("jokes and humor", model=variant, top_k
humor_features
```

```
context = client.features.inspect(
    messages=joke_conversation[0],
    model=variant,
    features=humor_features
)
context
```

Now you can retrieve the top k activating *humor features* in the `context`. This might be a more interesting set of features for downstream tasks.

Top Features Code | Top Features Output

```
humor_feature_acts = context.top(k=5)
humor_feature_acts
```

Save and load your model variants

You can serialize a variant to JSON format for saving.

Save Variant Code | Save Variant Output

```
variant.reset()
variant.set(pirate_features[1], 0.9)
variant_json = variant.json()
variant_json
```

And load a variant from JSON format.

Load Variant Code | Load Variant Output

```
loaded_variant = goodfire.Variant.from_json(variant_json)
loaded_variant
```

```
for token in client.chat.completions.create(
    [
        {"role": "user", "content": "tell me about whales"}
    ],
    model=loaded_variant,
    stream=True,
    max_completion_tokens=150,
):
    print(token.choices[0].delta.content, end="")
```

Using OpenAI SDK

You can also work directly with the OpenAI SDK for inference since our endpoint is fully compatible.

Install OpenAI Code

```
!pip install openai --quiet
```

OpenAI SDK Code

```
from openai import OpenAI

oai_client = OpenAI(
    api_key=GOODFIRE_API_KEY,
    base_url="https://api.goodfire.ai/api/inference/v1",
)

response = oai_client.chat.completions.create(
    messages=[
        {"role": "user", "content": "who is this"},
    ],
    model=variant.base_model,
    extra_body={"controller": variant.controller.json()},
```

For more advanced usage and detailed API reference, check out our SDK reference and example notebooks.

‹ Introduction

Decision Trees ›

Notebooks

Removing Knowledge

By creating a new Variant with certain features removed, we can remove knowledge from the model.

In this example, we'll create a variant which doesn't know about famous celebrities.

Open in Colab

Notebooks

Sorting by Features

By using Feature Activations we can sort data by the features that are most relevant to a given query.

In this example, we'll sort Elon Musk's tweets by the sarcasm feature.

Open in Colab

‹ Removing Knowledge

Classes

Variants

Documentation for working with model variants

Variants are edits to a model that allow you to modify model behavior by adjusting feature activations and defining conditional behaviors.

Creating Variants

Basic Usage

Create a variant by instantiating the `Variant` class with a base model:

Code Output

```
from goodfire import Variant

# Create a variant from a base model
variant = Variant("meta-llama/Llama-3.3-70B-Instruct")
print(variant)
```

Adding features to a variant

Adding features to a variant Edited variant

```
from goodfire import Variant

# Create a variant from a base model
variant = Variant("meta-llama/Llama-3.1-8B-Instruct")

# Search for a feature to modify
```

```
feature = client.features.search("formal writing style", model=variant, top_k=1

# Set feature modifications
variant.set(feature, 0.5)  # Value typically between -1 and 1
```

Conditional Controls

You can create variants that respond dynamically to feature activations:

Conditional Controls

```
# First get some features to work with
whale_feature = client.features.search("whales", model=variant, top_k=1)[0]
pirate_feature = client.features.search("pirate speech", model=variant, top_k=1

# Activate pirate features when whale features are detected
variant.set_when(whale_feature > 0.75, {
    pirate_feature: 0.5
})

# Abort generation if certain features are too strong
variant.abort_when(whale_feature > 0.9)

# Custom handler when condition is met
def my_handler(context):
    print(f"Whale feature activated with strength: {context.activations[whale_f
```

Methods

set()

Set feature modifications. This method is overloaded to handle different input types.

Signatures:

```
def set(self, feature: Feature | FeatureGroup, value: float)
```

```
def set(self, edits: dict[Feature, float] | FeatureEdits)
```


Parameters:

feature `Feature | FeatureGroup`

Single feature or feature group to modify. Required when using the first signature.

---

value `float`

Modification value (typically between -1 and 1). Required when using the first signature.

---

edits `dict[Feature, float] | FeatureEdits`

Dictionary of features and their values, or a FeatureEdits object. Required when using the second signature.

Examples:

```
# Single feature modification
formal_feature = client.features.search("formal writing style", model=variant,
variant.set(formal_feature, 0.5)

# Multiple features at once using dict
casual_feature = client.features.search("casual writing style", model=variant,
variant.set({
    formal_feature: 0.5,
    casual_feature: -0.3
})

# Using FeatureEdits object
edits = client.features.AutoSteer("be funny", model=variant)
variant.set(edits)
```

set_when()

Define conditional feature modifications.

Parameters:

condition `ConditionalGroup` required

Condition that triggers the modifications

---

values `dict[Feature, float] | FeatureEdits` required

Feature modifications to apply when condition is met

Example:

set_when example

```
whale_feature = client.features.search("whales", model=variant, top_k=1)[0]
pirate_feature = client.features.search("pirate speech", model=variant, top_k=1
# Activate pirate features when whale features are detected
variant.set_when(whale_feature > 0.3, {
    pirate_feature: 0.5
})
```

abort_when()

Abort generation when a condition is met.

Parameters:

condition `ConditionalGroup` required

Condition that triggers the abort

Example:

Abort example

```
inappropriate_feature = client.features.search("inappropriate content", model=v
# Abort if inappropriate content is detected
variant.abort_when(inappropriate_feature > 0.8)
```

```
try:
    response = client.chat.completions.create(
        messages=[{"role": "user", "content": "..."}],
        model=variant,
        stream=False
    )
except goodfire.exceptions.InferenceAbortedException:
    print("Aborted because of inappropriate content")

print(variant)
```

reset()

Remove all feature modifications.

Example:

Reset variant

```
variant.reset()
```

clear()

Remove modifications for specific features.

Parameters:

feature `Feature | FeatureGroup` required

Feature(s) to clear modifications for

Example:

Clear particular feature

```
variant.clear(feature)
```

Serialization

Variants can be serialized to and from JSON:

Storing and loading variants

```
# Save variant to JSON
variant_json = variant.json()

# Load variant from JSON
loaded_variant = Variant.from_json(variant_json)
```

Using with OpenAI SDK

Variants are compatible with the OpenAI SDK:

Using with OpenAI SDK

```
from openai import OpenAI

client = OpenAI(
    api_key="YOUR_GOODFIRE_API_KEY",
    base_url="https://api.goodfire.ai/api/inference/v1"
)

response = client.chat.completions.create(
    messages=[{"role": "user", "content": "Hello"}],
    model=variant.base_model,
    extra_body={"controller": variant.controller.json()}
)
```

Classes

VariantMetaData

Metadata about a model variant.

Properties:

`name` str

Name of the variant

`base_model` str

Base model identifier

`id` str

Unique identifier for the variant

‹ Chat Completions

Features ›