On this page

oai.completion

Completion Objects

```
class Completion(openai_Completion)
```

(openai<1) A class for OpenAI completion API.

It also supports: ChatCompletion, Azure OpenAI API.

set_cache

```
@classmethod
def set_cache(cls,
              seed: Optional[int] = 41,
              cache_path_root: Optional[str] = ".cache")
```

Set cache path.

Arguments:

- `seed` *int, Optional* - The integer identifier for the pseudo seed. Results corresponding to different seeds will be cached in different places.
- `cache_path` *str, Optional* - The root path for the cache. The complete cache path will be {cache_path_root}/{seed}.

clear_cache

```
@classmethod
def clear_cache(cls,
                seed: Optional[int] = None,
                cache_path_root: Optional[str] = ".cache")
```

Clear cache.

Arguments:

- `seed` *int, Optional* - The integer identifier for the pseudo seed. If omitted, all caches under cache_path_root will be cleared.
- `cache_path` *str, Optional* - The root path for the cache. The complete cache path will be {cache_path_root}/{seed}.

tune

```
@classmethod
def tune(cls,
         data: List[Dict],
         metric: str,
         mode: str,
         eval_func: Callable,
         log_file_name: Optional[str] = None,
         inference_budget: Optional[float] = None,
         optimization_budget: Optional[float] = None,
         num_samples: Optional[int] = 1,
         logging_level: Optional[int] = logging.WARNING,
         **config)
```

Tune the parameters for the OpenAI API call.

TODO: support parallel tuning with ray or spark. TODO: support agg_method as in test

Arguments:

- `data` *list* - The list of data points.

- `metric` *str* - The metric to optimize.
- `mode` *str* - The optimization mode, "min" or "max.
- `eval_func` *Callable* - The evaluation function for responses. The function should take a list of responses and a data point as input, and return a dict of metrics. For example,

```
def eval_func(responses, **data):
    solution = data["solution"]
    success_list = []
    n = len(responses)
    for i in range(n):
        response = responses[i]
        succeed = is_equiv_chain_of_thought(response, solution)
        success_list.append(succeed)
    return {
        "expected_success": 1 - pow(1 - sum(success_list) / n, n),
        "success": any(s for s in success_list),
    }
```

- `log_file_name` *str, optional* - The log file.
- `inference_budget` *float, optional* - The inference budget, dollar per instance.
- `optimization_budget` *float, optional* - The optimization budget, dollar in total.
- `num_samples` *int, optional* - The number of samples to evaluate. -1 means no hard restriction in the number of trials and the actual number is decided by optimization_budget. Defaults to 1.
- `logging_level` *optional* - logging level. Defaults to logging.WARNING.
- `metric`0 *dict* - The search space to update over the default search. For prompt, please provide a string/Callable or a list of strings/Callables.
  - If prompt is provided for chat models, it will be converted to messages under role "user".
  - Do not provide both prompt and messages for chat models, but provide either of them.
  - A string template will be used to generate a prompt for each data instance using `metric`1.
  - A callable template will be used to generate a prompt for each data instance using `metric`2. For stop, please provide a string, a list of strings, or a list of lists of strings. For messages (chat models only), please provide a list of messages (for a single chat prefix) or a list of lists of messages (for multiple choices of chat prefix to choose from). Each message should be a dict with keys "role" and "content". The value of "content" can be a string/Callable template.

Returns:

- `metric`3 - The optimized hyperparameter setting.
- `metric`4 - The tuning results.

create

```
@classmethod
def create(cls,
           context: Optional[Dict] = None,
           use_cache: Optional[bool] = True,
           config_list: Optional[List[Dict]] = None,
           filter_func: Optional[Callable[[Dict, Dict], bool]] = None,
           raise_on_ratelimit_or_timeout: Optional[bool] = True,
           allow_format_str_template: Optional[bool] = False,
           **config)
```

Make a completion for a given context.

Arguments:

- `context` *Dict, Optional* - The context to instantiate the prompt. It needs to contain keys that are used by the prompt template or the filter function. E.g., `prompt="Complete the following sentence: {prefix}, context={"prefix": "Today I feel"}`. The actual prompt will be: "Complete the following sentence: Today I feel". More examples can be found at templating.
- `use_cache` *bool, Optional* - Whether to use cached responses.
- `config_list` *List, Optional* - List of configurations for the completion to try. The first one that does not raise an error will be used. Only the differences from the default config need to be provided. E.g.,

```
response = oai.Completion.create(
    config_list=[
        {
            "model": "gpt-4",
            "api_key": os.environ.get("AZURE_OPENAI_API_KEY"),
            "api_type": "azure",
            "base_url": os.environ.get("AZURE_OPENAI_API_BASE"),
            "api_version": "2023-03-15-preview",
        },
        {
            "model": "gpt-3.5-turbo",
            "api_key": os.environ.get("OPENAI_API_KEY"),
            "api_type": "open_ai",
            "base_url": "https://api.openai.com/v1",
        },
        {
            "model": "llama-7B",
            "base_url": "http://127.0.0.1:8080",
            "api_type": "open_ai",
        }
    ],
    prompt="Hi",
)
```

- `filter_func` *Callable, Optional* - A function that takes in the context and the response and returns a boolean to indicate whether the response is valid. E.g.,

```
def yes_or_no_filter(context, config, response):
    return context.get("yes_or_no_choice", False) is False or any(
        text in ["Yes.", "No."] for text in oai.Completion.extract_text(response)
    )
```

- `raise_on_ratelimit_or_timeout` *bool, Optional* - Whether to raise RateLimitError or Timeout when all configs fail. When set to False, -1 will be returned when all configs fail.
- `allow_format_str_template` *bool, Optional* - Whether to allow format string template in the config.
- `**config` - Configuration for the openai API call. This is used as parameters for calling openai API. The "prompt" or "messages" parameter can contain a template (str or Callable) which will be instantiated with the context. Besides the parameters for the openai API call, it can also contain:
    - `prompt="Complete the following sentence: {prefix}, context={"prefix": "Today I feel"}`0 (int): the total time (in seconds) allowed for retrying failed requests.
    - `prompt="Complete the following sentence: {prefix}, context={"prefix": "Today I feel"}`1 (int): the time interval to wait (in seconds) before retrying a failed request.
    - `prompt="Complete the following sentence: {prefix}, context={"prefix": "Today I feel"}`2 (int) for the cache. This is useful when implementing "controlled randomness" for the completion.

Returns:

Responses from OpenAI API, with additional fields.

- `prompt="Complete the following sentence: {prefix}, context={"prefix": "Today I feel"}`3: the total cost. When `config_list` is provided, the response will contain a few more fields:
- `prompt="Complete the following sentence: {prefix}, context={"prefix": "Today I feel"}`5: the index of the config in the config_list that is used to generate the response.
- `prompt="Complete the following sentence: {prefix}, context={"prefix": "Today I feel"}`6: whether the response passes the filter function. None if no filter is provided.

test

```
@classmethod
def test(cls,
         data,
         eval_func=None,
         use_cache=True,
         agg_method="avg",
         return_responses_and_per_instance_result=False,
         logging_level=logging.WARNING,
         **config)
```

Evaluate the responses created with the config for the OpenAI API call.

Arguments:

- `data` *list* - The list of test data points.
- `eval_func` *Callable* - The evaluation function for responses per data instance. The function should take a list of responses and a data point as input, and return a dict of metrics. You need to either provide a valid callable eval_func; or do not provide one (set None) but call the test function after calling the tune function in which a eval_func is provided. In the latter case we will use the eval_func provided via tune function. Defaults to None.

```
def eval_func(responses, **data):
    solution = data["solution"]
    success_list = []
    n = len(responses)
    for i in range(n):
        response = responses[i]
        succeed = is_equiv_chain_of_thought(response, solution)
        success_list.append(succeed)
    return {
        "expected_success": 1 - pow(1 - sum(success_list) / n, n),
        "success": any(s for s in success_list),
    }
```

- `use_cache` *bool, Optional* - Whether to use cached responses. Defaults to True.
- `agg_method` *str, Callable or a dict of Callable* - Result aggregation method (across multiple instances) for each of the metrics. Defaults to 'avg'. An example agg_method in str:

```
agg_method = 'median'
```

An example agg_method in a Callable:

```
agg_method = np.median
```

An example agg_method in a dict of Callable:

```
agg_method={'median_success': np.median, 'avg_success': np.mean}
```

- `return_responses_and_per_instance_result` *bool* - Whether to also return responses and per instance results in addition to the aggregated results.
- `logging_level` *optional* - logging level. Defaults to logging.WARNING.
- `eval_func`0 *dict* - parameters passed to the openai api call `eval_func`1.

Returns:

None when no valid eval_func is provided in either test or tune; Otherwise, a dict of aggregated results, responses and per instance results if `return_responses_and_per_instance_result` is True; Otherwise, a dict of aggregated results (responses and per instance results are not returned).

cost

```
@classmethod
def cost(cls, response: dict)
```

Compute the cost of an API call.

Arguments:

- `response` *dict* - The response from OpenAI API.

Returns:

The cost in USD. 0 if the model is not supported.

extract_text

```
@classmethod
def extract_text(cls, response: dict) -> List[str]
```

Extract the text from a completion or chat response.

Arguments:

- `response` *dict* - The response from OpenAI API.

Returns:

A list of text in the responses.

extract_text_or_function_call

```
@classmethod
def extract_text_or_function_call(cls, response: dict) -> List[str]
```

Extract the text or function calls from a completion or chat response.

Arguments:

- `response` *dict* - The response from OpenAI API.

Returns:

A list of text or function calls in the responses.

logged_history

```
@classmethod
@property
def logged_history(cls) -> Dict
```

Return the book keeping dictionary.

print_usage_summary

```
@classmethod
def print_usage_summary(cls) -> Dict
```

Return the usage summary.

start_logging

```
@classmethod
def start_logging(cls,
                  history_dict: Optional[Dict] = None,
                  compact: Optional[bool] = True,
                  reset_counter: Optional[bool] = True)
```

Start book keeping.

Arguments:

- `history_dict` *Dict* - A dictionary for book keeping. If no provided, a new one will be created.
- `compact` *bool* - Whether to keep the history dictionary compact. Compact history contains one key per conversation, and the value is a dictionary like:

```
{
    "create_at": [0, 1],
    "cost": [0.1, 0.2],
}
```

where "created_at" is the index of API calls indicating the order of all the calls, and "cost" is the cost of each call. This example shows that the conversation is based on two API calls. The compact format is useful for condensing the history of a conversation. If compact is False, the history dictionary will contain all the API calls: the key is the index of the API call, and the value is a dictionary like:

```
{
    "request": request_dict,
    "response": response_dict,
}
```

where request_dict is the request sent to OpenAI API, and response_dict is the response. For a conversation containing two API calls, the non-compact history dictionary will be like:

```
{
    0: {
        "request": request_dict_0,
        "response": response_dict_0,
    },
    1: {
        "request": request_dict_1,
        "response": response_dict_1,
    },
```

The first request's messages plus the response is equal to the second request's messages. For a conversation with many turns, the non-compact history dictionary has a quadratic size while the compact history dict has a linear size.

- `reset_counter` *bool* - whether to reset the counter of the number of API calls.

stop_logging

```
@classmethod
def stop_logging(cls)
```

End book keeping.

ChatCompletion Objects

```
class ChatCompletion(Completion)
```

(openai<1) A class for OpenAI API ChatCompletion. Share the same API as Completion.

Edit this page

Previous
« client

Next
openai_utils »