```
import asyncio

import base64

import concurrent.futures

import time

from abc import abstractmethod

from concurrent.futures import ThreadPoolExecutor

from io import BytesIO

from typing import List, Optional, Tuple

import requests

from PIL import Image

from termcolor import colored



class BaseMultiModalModel:

    """

    Base class for multimodal models



    Args:

        model_name (Optional[str], optional): Model name. Defaults to None.

        temperature (Optional[int], optional): Temperature. Defaults to 0.5.

        max_tokens (Optional[int], optional): Max tokens. Defaults to 500.

        max_workers (Optional[int], optional): Max workers. Defaults to 10.

        top_p (Optional[int], optional): Top p. Defaults to 1.

        top_k (Optional[int], optional): Top k. Defaults to 50.

        beautify (Optional[bool], optional): Beautify. Defaults to False.
```

device (Optional[str], optional): Device. Defaults to "cuda".

max_new_tokens (Optional[int], optional): Max new tokens. Defaults to 500.

retries (Optional[int], optional): Retries. Defaults to 3.

Examples:

```
>>> from swarm_models.base_multimodal_model import BaseMultiModalModel
>>> model = BaseMultiModalModel()
>>> model.run("Generate a summary of this text")
>>> model.run("Generate a summary of this text", "https://www.google.com/images/branding/googlelogo/2x/googlelogo_color_272x92dp.png")
>>> model.run_batch(["Generate a summary of this text", "Generate a summary of this text"])
>>> model.run_batch([("Generate a summary of this text", "https://www.google.com/images/branding/googlelogo/2x/googlelogo_color_272x92dp.png"), ("Generate a summary of this text", "https://www.google.com/images/branding/googlelogo/2x/googlelogo_color_272x92dp.png")])
>>> model.run_batch_async(["Generate a summary of this text", "Generate a summary of this text"])
>>> model.run_batch_async([("Generate a summary of this text", "https://www.google.com/images/branding/googlelogo/2x/googlelogo_color_272x92dp.png"), ("Generate a summary of this text", "https://www.google.com/images/branding/googlelogo/2x/googlelogo_color_272x92dp.png")])
>>> model.run_batch_async_with_retries(["Generate a summary of this text", "Generate a summary of this text"])
>>> model.run_batch_async_with_retries([("Generate a summary of this text", "https://www.google.com/images/branding/googlelogo/2x/googlelogo_color_272x92dp.png"), ("Generate a summary of this text",
"https://www.google.com/images/branding/googlelogo/2x/googlelogo_color_272x92dp.png")])

        >>> model.generate_summary("Generate a summary of this text")

        >>> model.set_temperature(0.5)

        >>> model.set_max_tokens(500)

        >>> model.get_generation_time()

        >>> model.get_chat_history()

        >>> model.get_unique_chat_history()

        >>> model.get_chat_history_length()

        >>> model.get_unique_chat_history_length()

        >>> model.get_chat_history_tokens()

        >>> model.print_beautiful("Print this beautifully")

        >>> model.stream("Stream this")

        >>> model.unique_chat_history()

        >>> model.clear_chat_history()

        >>> model.get_img_from_web("https://www.google.com/images/branding/googlelogo/")


    """


    def __init__(
        self,
        model_name: Optional[str] = None,
        temperature: Optional[int] = 0.5,
        max_tokens: Optional[int] = 500,
        max_workers: Optional[int] = 10,
        top_p: Optional[int] = 1,
        top_k: Optional[int] = 50,
        beautify: Optional[bool] = False,
        device: Optional[str] = "cuda",
        max_new_tokens: Optional[int] = 500,
        retries: Optional[int] = 3,
        system_prompt: Optional[str] = None,
        meta_prompt: Optional[str] = None,
        *args,
        **kwargs,
    ):
        self.model_name = model_name
        self.temperature = temperature
        self.max_tokens = max_tokens
        self.max_workers = max_workers
        self.top_p = top_p
        self.top_k = top_k
        self.beautify = beautify
        self.device = device
        self.max_new_tokens = max_new_tokens
        self.retries = retries
        self.system_prompt = system_prompt
        self.meta_prompt = meta_prompt
        self.chat_history = []

    @abstractmethod
    def run(
        self,
        task: Optional[str] = None,
        img: Optional[str] = None,
        *args,
        **kwargs,
    ):
        """Run the model"""

    def __call__(
        self,
        task: Optional[str] = None,
        img: Optional[str] = None,
        *args,
        **kwargs,
    ):
        """Call the model

        Args:
            task (str): _description_
            img (str): _description_

        Returns:
            _type_: _description_
        """
        return self.run(task, img, *args, **kwargs)

    async def arun(self, task: str, img: str, *args, **kwargs):
        """Run the model asynchronously"""

    def get_img_from_web(self, img: str, *args, **kwargs):
        """Get the image from the web"""
        try:
            response = requests.get(img)
            response.raise_for_status()
            image_pil = Image.open(BytesIO(response.content))
            return image_pil
        except requests.RequestException as error:
            print(
                f"Error fetching image from {img} and error: {error}"
            )
            return None

    def encode_img(self, img: str):
        """Encode the image to base64"""
        with open(img, "rb") as image_file:
            return base64.b64encode(image_file.read()).decode("utf-8")

    def get_img(self, img: str):
        """Get the image from the path"""
        image_pil = Image.open(img)
        return image_pil

    def clear_chat_history(self):
        """Clear the chat history"""
        self.chat_history = []

    def run_many(
        self, tasks: List[str], imgs: List[str], *args, **kwargs
    ):
        """
        Run the model on multiple tasks and images all at once using concurrent

        Args:
            tasks (List[str]): List of tasks
            imgs (List[str]): List of image paths

        Returns:
            List[str]: List of responses


        """
        # Instantiate the thread pool executor
        with ThreadPoolExecutor(
            max_workers=self.max_workers
        ) as executor:
            results = executor.map(self.run, tasks, imgs)

        # Print the results for debugging
        for result in results:
            print(result)


    def run_batch(
        self, tasks_images: List[Tuple[str, str]]
    ) -> List[str]:
        """Process a batch of tasks and images"""
        with concurrent.futures.ThreadPoolExecutor() as executor:
            futures = [
                executor.submit(self.run, task, img)
                for task, img in tasks_images
            ]
            results = [future.result() for future in futures]
        return results


    async def run_batch_async(
        self, tasks_images: List[Tuple[str, str]]
    ) -> List[str]:
        """Process a batch of tasks and images asynchronously"""
        loop = asyncio.get_event_loop()
        futures = [
            loop.run_in_executor(None, self.run, task, img)
            for task, img in tasks_images
        ]
        return await asyncio.gather(*futures)


    async def run_batch_async_with_retries(
        self, tasks_images: List[Tuple[str, str]]
    ) -> List[str]:
        """Process a batch of tasks and images asynchronously with retries"""
        loop = asyncio.get_event_loop()
        futures = [
            loop.run_in_executor(
                None, self.run_with_retries, task, img
            )
            for task, img in tasks_images
        ]
        return await asyncio.gather(*futures)

    def unique_chat_history(self):
        """Get the unique chat history"""
        return list(set(self.chat_history))

    def run_with_retries(self, task: str, img: str):
        """Run the model with retries"""
        for i in range(self.retries):
            try:
                return self.run(task, img)
            except Exception as error:
                print(f"Error with the request {error}")
                continue

    def run_batch_with_retries(
        self, tasks_images: List[Tuple[str, str]]
    ):
        """Run the model with retries"""
        for i in range(self.retries):
            try:
                return self.run_batch(tasks_images)
            except Exception as error:
                print(f"Error with the request {error}")
                continue


    def _tokens_per_second(self) -> float:
        """Tokens per second"""
        elapsed_time = self.end_time - self.start_time
        if elapsed_time == 0:
            return float("inf")
        return self._num_tokens() / elapsed_time


    def _time_for_generation(self, task: str) -> float:
        """Time for Generation"""
        self.start_time = time.time()
        self.run(task)
        self.end_time = time.time()
        return self.end_time - self.start_time


    @abstractmethod
    def generate_summary(self, text: str) -> str:
        """Generate Summary"""


    def set_temperature(self, value: float):
        """Set Temperature"""
        self.temperature = value


    def set_max_tokens(self, value: int):
        """Set new max tokens"""
        self.max_tokens = value


    def get_generation_time(self) -> float:
        """Get generation time"""
        if self.start_time and self.end_time:
            return self.end_time - self.start_time
        return 0


    def get_chat_history(self):
        """Get the chat history"""
        return self.chat_history


    def get_unique_chat_history(self):
        """Get the unique chat history"""
        return list(set(self.chat_history))


    def get_chat_history_length(self):
        """Get the chat history length"""
        return len(self.chat_history)


    def get_unique_chat_history_length(self):
        """Get the unique chat history length"""
        return len(list(set(self.chat_history)))


    def get_chat_history_tokens(self):
        """Get the chat history tokens"""
        return self._num_tokens()


    def print_beautiful(self, content: str, color: str = "cyan"):
        """Print Beautifully with termcolor"""
        content = colored(content, color)
        print(content)


    def stream_response(self, text: str):
        """Stream the output


        Args:
            content (str): _description_
        """
        for chunk in text:
            print(chunk)


    def meta_prompt(self):
        """Meta Prompt
```

```
        Returns:
            _type_: _description_
        """
        META_PROMPT = """
        For any labels or markings on an image that you reference in your response, please
        enclose them in square brackets ([]) and list them explicitly. Do not use ranges; for
        example, instead of '1 - 4', list as '[1], [2], [3], [4]'. These labels could be
        numbers or letters and typically correspond to specific segments or parts of the image.
        """
        return META_PROMPT

    def set_device(self, device):
        """
        Changes the device used for inference.

        Parameters
        ----------
            device : str
                The new device to use for inference.
        """
        self.device = device
        self.model.to(self.device)

    def set_max_length(self, max_length):
        """Set max_length"""
```

```
self.max_length = max_length
```