```
import concurrent.futures
import re
from collections import Counter
from typing import Any, Callable, List, Optional

from swarms.structs.agent import Agent
from swarms.structs.conversation import Conversation
from swarms.utils.file_processing import create_file
from swarms.utils.loguru_logger import initialize_logger

logger = initialize_logger(log_folder="majority_voting")



def extract_last_python_code_block(text):
    """
    Extracts the last Python code block from the given text.

    Args:
        text (str): The text to search for Python code blocks.

    Returns:
        str or None: The last Python code block found in the text, or None if no code block is found.
    """
    # The regular expression pattern for Python code blocks
    pattern = r"```[pP]ython(.*?)```"
```

```
    # Find all matches in the text
    matches = re.findall(pattern, text, re.DOTALL)

    # If there are matches, return the last one
    if matches:
        return matches[-1].strip()
    else:
        return None


def parse_code_completion(agent_response, question):
    """
    Parses the code completion response from the agent and extracts the last Python code block.

    Args:
        agent_response (str): The response from the agent.
        question (str): The original question.

    Returns:
        tuple: A tuple containing the parsed Python code and a boolean indicating success.
    """
    python_code = extract_last_python_code_block(agent_response)
    if python_code is None:
        if agent_response.count("impl]") == 0:
            python_code = agent_response
        else:
            python_code_lines = agent_response.split("\n")
            python_code = ""
            in_func = False
            for line in python_code_lines:
                if in_func:
                    python_code += line + "\n"
                if "impl]" in line:
                    in_func = True
    if python_code.count("def") == 0:
        python_code = question + python_code
    return python_code, True



def most_frequent(
    clist: list,
    cmp_func: callable = None,
):
    """
    Finds the most frequent element in a list based on a comparison function.

    Args:
        clist (list): The list of elements to search.
          cmp_func (function, optional): The comparison function used to determine the frequency of
elements.
            If not provided, the default comparison function is used.
    Returns:
        tuple: A tuple containing the most frequent element and its frequency.
    """
    counter = 0
    num = clist[0]

    for i in clist:
        current_frequency = sum(cmp_func(i, item) for item in clist)
        if current_frequency > counter:
            counter = current_frequency
            num = i

    return num, counter


def majority_voting(answers: List[str]):
    """
    Performs majority voting on a list of answers and returns the most common answer.

    Args:
        answers (list): A list of answers.

    Returns:
        The most common answer in the list.
    """
    counter = Counter(answers)
    if counter:
        answer = counter.most_common(1)[0][0]
    else:
        answer = "I don't know"

    return answer



class MajorityVoting:
    """
    Class representing a majority voting system for agents.

    Args:
        agents (list): A list of agents to be used in the majority voting system.
        output_parser (function, optional): A function used to parse the output of the agents.
            If not provided, the default majority voting function is used.
        autosave (bool, optional): A boolean indicating whether to autosave the conversation to a file.
        verbose (bool, optional): A boolean indicating whether to enable verbose logging.
    Examples:
        >>> from swarms.structs.agent import Agent
        >>> from swarms.structs.majority_voting import MajorityVoting
        >>> agents = [
        ...     Agent("GPT-3"),
        ...     Agent("Codex"),
        ...     Agent("Tabnine"),
        ... ]
        >>> majority_voting = MajorityVoting(agents)

        >>> majority_voting.run("What is the capital of France?")

        'Paris'


    """


    def __init__(

        self,

        name: str = "MajorityVoting",

        description: str = "A majority voting system for agents",

        agents: List[Agent] = [],

        output_parser: Optional[Callable] = majority_voting,

        autosave: bool = False,

        verbose: bool = False,

        *args,

        **kwargs,

    ):

        self.agents = agents

        self.output_parser = output_parser

        self.autosave = autosave

        self.verbose = verbose


        self.conversation = Conversation(

            time_enabled=True, *args, **kwargs

        )
        # If autosave is enabled, save the conversation to a file
        if self.autosave:
            create_file(
                str(self.conversation), "majority_voting.json"
            )

        # Log the agents
        logger.info("Initializing majority voting system")
        # Length of agents
        logger.info(f"Number of agents: {len(self.agents)}")
        logger.info(
            "Agents:"
            f" {', '.join(agent.agent_name for agent in self.agents)}"
        )

    def run(self, task: str, *args, **kwargs) -> List[Any]:
        """
        Runs the majority voting system and returns the majority vote.

        Args:
            task (str): The task to be performed by the agents.
            *args: Variable length argument list.
            **kwargs: Arbitrary keyword arguments.

        Returns:
            List[Any]: The majority vote.
```

```
"""
# Route to each agent
with concurrent.futures.ThreadPoolExecutor() as executor:
    logger.info("Running agents concurrently")

    futures = [
        executor.submit(agent.run, task, *args)
        for agent in self.agents
    ]
    results = [
        future.result()
        for future in concurrent.futures.as_completed(futures)
    ]

# Add responses to conversation and log them
for agent, response in zip(self.agents, results):
    response = (
        response if isinstance(response, list) else [response]
    )
    self.conversation.add(agent.agent_name, response)
    logger.info(
        f"[Agent][Name: {agent.agent_name}][Response:"
        f" {response}]"
    )
```

```
# Perform majority voting on the conversation
responses = [
    message["content"]
    for message in self.conversation.conversation_history
    if message["role"] == "agent"
]

# If an output parser is provided, parse the responses
if self.output_parser is not None:
    majority_vote = self.output_parser(
        responses, *args, **kwargs
    )
else:
    majority_vote = majority_voting(responses)

# Return the majority vote
return majority_vote
```