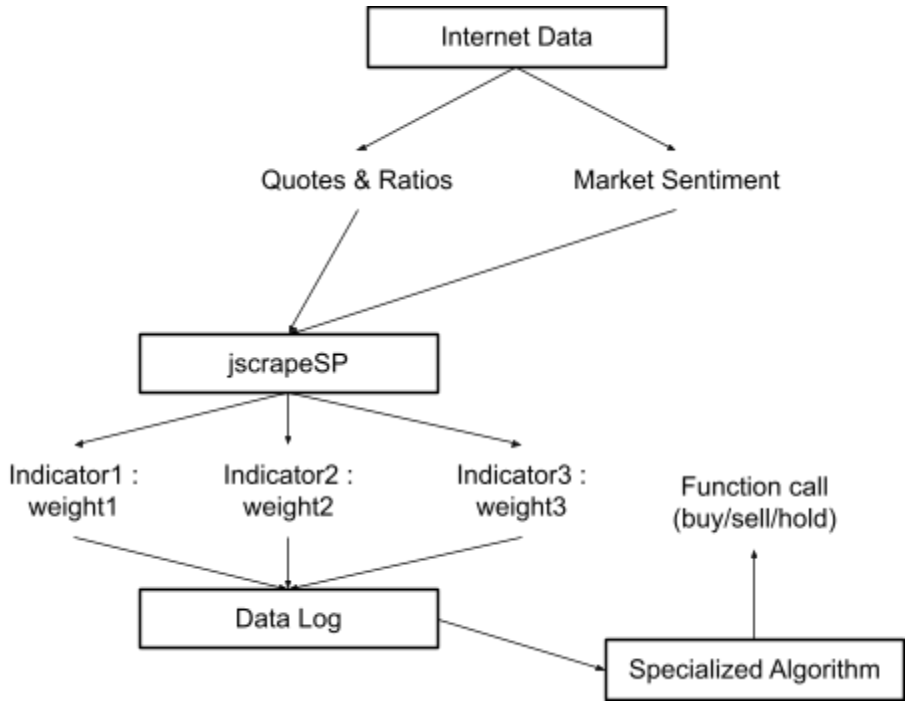
jscrapeSP.py

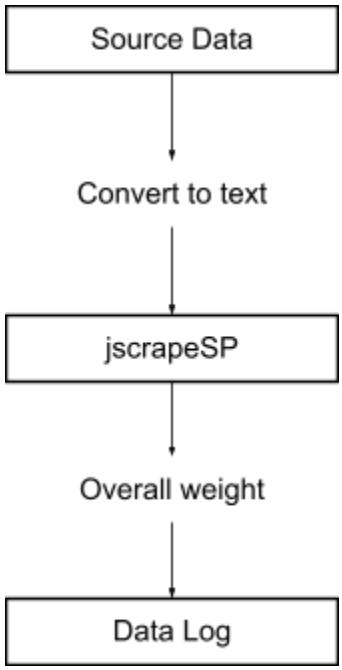
Anyu Chen

Objective:

The goal of jscrapeSP was to synthesize quantitative and qualitative stock data to weigh a stock's profit outlook using a series of indicators. The fundamental model leverages the pattern recognition of AI – by inheriting context through a set of indicators with assigned weight in an attempt to reduce the amount of input tokens utilized. Through fine-tuning with historical stock data and trading outcomes, the aim is for the model to accurately weigh indicators based on novel context information. In shifting the weight of individual indicators through quantitative and qualitative analysis, an informed trading decision can be called based on the aggregate weight of these traits (processed by another specialized program).



As jscrapeSP is still in the earliest stage of development, a specialized set of indicators and a fine-tuned version of the model has not been implemented. The current prototype serves as a rudimentary demonstration of the jscrapeSP functionality, with limitations imposed as monomodal data source, limited quantitative & qualitative reasoning capabilities, and the lack of an established indicator set to justify function call. The current model follows a simplified road map outlined below.



Input & Output fields:

Inputs: prompt, URL

The input fields consist of a prompt string, and a URL string. The prompt is used to directly prompt the LLM, while the URL is used to fetch context information using webscraping tools from BeautifulSoup and requests.

Output: analysis, action, weight

The output is consistently formatted using a structured output schematic. Returned in JSON, the output consists of a paragraph analysis of the information on the input URL, a function call of buy/sell/hold based on the analysis, and an overall weight valuation of the stock based on the analysis.

Libraries:

```
from google import genai
from google.genai.types import
GenerateContentConfig
from google.genai.types import
Tool

from bs4 import BeautifulSoup
import requests

import os

import datetime
```

<- used to send API requests to Gemini AI.

<- used to configure the model parameters

<- used to give Gemini a URL access tool.

<- used to navigate the HTML content scraped from sources.
<- used to fetch HTML content from the input URL.

<- used to retrieve the GEMINI_API_KEY as an environment variable.

<- used to provide a timestamp for each data entry in the logging process.

Code Explanation:

#WEBSCRAPING

scrapeLink(link)

The webscraping function scrapeLink takes an input URL to which an HTTP request is sent, then stores the content in the variable “response”. The content is then parsed using BeautifulSoup and returned as a navigable object. This function was utilized in the single-step Gemini call to scrape text data from input URLs for context.

```
def scrapeLink(link):
   response = requests.get(link)
   html_content = BeautifulSoup(response.text,
'html.parser')
   return html_content
```

<- send HTTP request, store content in response
<- parse content

<- return parsed HTML as BeautifulSoup object

soupFind(html_content,tag,target)

The function soupFind takes the output of scrapeLink, and using a provided tag and target, is able to return the target content of a specified tag in the HTML. The model utilises a try-except block to catch errors that may occur as results of typos in the input fields.

```
def soupFind(html_content,tag,target):
    soupmix = list()                                    <- instantiate list for target content
    for content in html_content.find_all(tag):          <- loop through specified tags in BeautifulSoup object
        try:
            if target == "text":
                soupmix.append(content.get_text())      <- append to list the element's contents
            else:
                soupmix.append(content.get(target))
        except Exception as e:                          <- catch exception if error occurs in the try block
            print(f"Error occurred: {e}")
    return soupmix                                      <- return list of target content
```

#GEMINI CALL

scaleGemini_v1(request,url)

The function scaleGemini_v1 utilizes a two-step process to output structured JSON responses with context from given URLs. In order to bypass Gemini API's inability to use url context and structured output schematics simultaneously, a nested function call is used to first prompt Gemini to retrieve as much contextual information as possible from the given sources, and then the output is appended to the initial prompt as the prompt of the second Gemini call. The second Gemini call is then able to structure the output response in JSON, using the information fetched by the initial function call as context.

**Due to the nature of the function, nesting is required to generate the final response:*

```
scaleGemini_v1(scaleGemini_v1("example prompt","https://www.example.com"),"")
```

```
def scaleGemini_v1(request,url):
    client = genai.Client(api_key =                     <- fetch API key stored as an environment variable
os.getenv("GEMINI_API_KEY"))


    prompt = request                                    <- set input request as the model prompt
```

```
   if len(url) > 0:
       try:
           prompt = "Retrieve as much relevant,
accurate, and unbiased information as possible from the
provided references."
           prompt += f" References: "
           for link in url.split(" "):
               prompt += link + ", "

           tools = [
               {"url_context": {}},
           ]

           response = client.models.generate_content(
               model = "gemini-2.5-flash-lite", contents
= prompt,
               config = GenerateContentConfig(
                   temperature=0,
                   max_output_tokens=1000,
                   tools = tools
               ),
           )

           return request + " Web context: " +
response.text

       except Exception as e:
           print(f"Error occurred: {e}")

   else:

       schema = {
           "type": "object",
           "properties":{
               "analysis": {"type": "string",
"description": "detailed analysis of stock"},
```

<- check for input URL, the initial call

<- redefine prompt for information retrieval

<- append URLs to the prompt

<- define tools to enable url access

<- generate URL context

<- model configurations

<- return original request with URL context appended

<- catch exception if an error occurs in the above code

<- the second call, with input being the output of the first call

<- define output schematic for structured responses

```
                "function": {"type": "string",
"description": "best course of action to take, according
to analysis", "enum": ["buy","sell","hold"]},
                "weight": {"type": "number",
"description": "confidence weight of function from 0 to
1, with 0 = definitive sell, 0.5 = hold, 1 = definitive
buy"}
            },
            "required": ["analysis", "function",
"weight"]
        }

        response = client.models.generate_content(
            model = "gemini-2.5-flash-lite", contents =
prompt,
            config = GenerateContentConfig(
                temperature=0,
                max_output_tokens=1000,
                response_mime_type="application/json",
                response_schema=schema,
            ),
        )

    return response.text
```

<- generate response

<- model configurations

<- return response as text

`scaleGemini_v2(request,url)`

The single step call scaleGemini_v2 utilizes the scrapeLink function to retrieve textual information as URL context. In order to minimize resource usage and potentially increase the accuracy of responses, text from the URLs are manually extracted and appended to the input prompt. Leveraging the over 1,000,000 input token limit of Gemini 2.5 models, the full URL contents can be considered, drastically increasing accuracy of the response.

```
def scaleGemini_v2(request,url):

    client = genai.Client(api_key =
os.getenv("GEMINI_API_KEY"))
```

<- fetch API key stored as an environment variable

```
    prompt = request

    for link in url.split(" "):
        prompt += "*** START OF SITE ***" +
scrapeLink(link).get_text() + "*** END OF SITE ***"

    schema = {
        "type": "object",
        "properties":{
            "analysis": {"type": "string", "description":
"detailed analysis of stock"},
            "function": {"type": "string", "description":
"best course of action to take, according to analysis",
"enum": ["buy","sell","hold"]},
            "weight": {"type": "number", "description":
"confidence weight of function from 0 to 1, with 0 =
definitive sell, 0.5 = hold, 1 = definitive buy"}
        },
        "required": ["analysis", "function", "weight"]
    }

    response = client.models.generate_content(
        model = "gemini-2.5-flash-lite", contents =
prompt,
        config = GenerateContentConfig(
            temperature=0,
            max_output_tokens=1000,
            response_mime_type="application/json",
            response_schema=schema,
        ),
    )

    return response.text
```

<- add the request as part of the model prompt

<- append textual information from each url to the model prompt

<- define output schematic for structured responses

<- generate response

<- model configurations

<- return response as text

#DATA LOGGING

log(fileName,data)

The log function takes a .txt file and data as input parameters, generates a timestamp, and appends the data to the end of the .txt file along with the timestamp.

```
def log(fileName,data):
    timeStamp = datetime.datetime.now()
    with open(fileName,"a") as file:
        file.write(f"{timeStamp}\n" + f" {data}\n")
```

<- generate timestamp

<- access .txt file

<- append data to file

log(fileName,data)

The read function takes a .txt file and prints the file contents in the terminal for viewing.

```
def read(fileName):
    with open(fileName,"r") as file:
        for line in file:
            print(line)
```

<- read.txt file

<- print text by line

References & Declarations:

1. Google. (n.d.). *Gemini API | google AI for developers*. Google. https://ai.google.dev/gemini-api/docs
2. *Beautiful Soup documentation*. Beautiful Soup Documentation - Beautiful Soup 4.4.0 documentation. (n.d.). https://beautiful-soup-4.readthedocs.io/en/latest/
3. *Requests*. PyPI. (n.d.). https://pypi.org/project/requests/

I hereby declare that Github Copilot and Gemini AI have been used in research and debugging, but not in the creation of the project.