# **Comprehensive Financial News Scraper, Financial Data Analysis, and Semantic Search Documentation**

This documentation provides an overview of a Python-based system that scrapes the latest market news from multiple sources, including MoneyControl, Economic Times, Livemint, and Business Standard. It integrates with OpenAI for intelligent analysis and responses, includes a function to answer specific queries about funds, and offers data management and analysis capabilities through the StockDataManager class. Additionally, the system incorporates a VectorStore class for managing embeddings and enabling semantic search over historical queries and responses.

## Table of Contents

1. [Overview](https://www.perplexity.ai/search/can-you-make-a-github-style-do-OWmlQ5JNQyuWgyyC9JucNQ#overview)
2. [Installation](https://www.perplexity.ai/search/can-you-make-a-github-style-do-OWmlQ5JNQyuWgyyC9JucNQ#installation)
3. [Usage](https://www.perplexity.ai/search/can-you-make-a-github-style-do-OWmlQ5JNQyuWgyyC9JucNQ#usage)
4. [Modules, Classes, Functions, and NLP Processing](https://www.perplexity.ai/search/can-you-make-a-github-style-do-OWmlQ5JNQyuWgyyC9JucNQ#modules-classes-functions-and-nlp-processing)
   * [NewsArticle](https://www.perplexity.ai/search/can-you-make-a-github-style-do-OWmlQ5JNQyuWgyyC9JucNQ#newsarticle)
   * [FinancialNewsScraper](https://www.perplexity.ai/search/can-you-make-a-github-style-do-OWmlQ5JNQyuWgyyC9JucNQ#financialnewsscraper)
   * [is\_related()](https://www.perplexity.ai/search/can-you-make-a-github-style-do-OWmlQ5JNQyuWgyyC9JucNQ#is_related)
   * [process\_article()](https://www.perplexity.ai/search/can-you-make-a-github-style-do-OWmlQ5JNQyuWgyyC9JucNQ#process_article)
   * [openai\_client.py](https://www.perplexity.ai/search/can-you-make-a-github-style-do-OWmlQ5JNQyuWgyyC9JucNQ#openai_clientpy)
   * [answer\_query()](https://www.perplexity.ai/search/can-you-make-a-github-style-do-OWmlQ5JNQyuWgyyC9JucNQ#answer_query)
   * [StockDataManager](https://www.perplexity.ai/search/can-you-make-a-github-style-do-OWmlQ5JNQyuWgyyC9JucNQ#stockdatamanager)
   * [VectorStore](https://www.perplexity.ai/search/can-you-make-a-github-style-do-OWmlQ5JNQyuWgyyC9JucNQ#vectorstore)
5. [Output Format](https://www.perplexity.ai/search/can-you-make-a-github-style-do-OWmlQ5JNQyuWgyyC9JucNQ#output-format)
6. [Error Handling](https://www.perplexity.ai/search/can-you-make-a-github-style-do-OWmlQ5JNQyuWgyyC9JucNQ#error-handling)

## Overview

The system collects financial news articles from the following sources:

* MoneyControl: Business and stock market news.
* Economic Times: Market news via RSS feeds.
* Livemint: Financial updates via RSS feeds.
* Business Standard: Market-related articles.

Each article includes:

* Title
* Content (or fallback to title if unavailable)
* URL
* Source
* Timestamp (if available)

The system uses NLP to:

* Summarize the article content.
* Extract named entities.
* Analyze the sentiment of the article.

Additionally, it integrates with OpenAI to provide intelligent, context-aware responses to financial questions, provides a direct query function to answer questions about funds, and offers data management and analysis capabilities through the StockDataManager class. Finally, the VectorStore class enables semantic search capabilities to identify similar historical queries and relevant entities.

## Installation

## **Prerequisites**

Ensure Python 3.7+ is installed along with the following libraries:

* requests
* beautifulsoup4
* feedparser
* fuzzywuzzy
* spacy
* transformers
* python-dotenv
* openai
* numpy
* faiss-cpu

Additionally, for the StockDataManager:

* json (built-in, but make sure it's working correctly)
* Create data directory to store .json files for stock data

## **Install Dependencies**

bash

pip install requests beautifulsoup4 feedparser fuzzywuzzy spacy transformers python-dotenv openai numpy faiss-cpu

python -m spacy download en\_core\_web\_sm

mkdir data

## **OpenAI API Key**

You'll need an OpenAI API key. Set it as an environment variable:

bash

export OPENAI\_API\_KEY="YOUR\_OPENAI\_API\_KEY"

## **Stock and Mutual Fund Data**

The StockDataManager relies on JSON files for stock and mutual fund data. Ensure these files (stock\_data.json, mutual\_funds\_data.json, and mf\_holdings\_data.json) are present in the /data directory. The formats of these files are explained in StockDataManager documentation.

## Modules, Classes, Functions, and NLP Processing

## **1. NewsArticle**

## **Description:**

A class representing a single news article.

## **Attributes:**

* title (str): The title of the article.
* content (str): The main content of the article.
* url (str): The URL of the article.
* source (str): The source of the article (e.g., MoneyControl).
* timestamp (str, optional): The publication timestamp.

## **Methods:**

* to\_dict() -> Dict: Converts the article object to a dictionary.

## **Example:**

Python

article = NewsArticle(

title="Market Update",

content="The Sensex rallied 300 points today...",

url="https://moneycontrol.com/article",

source="MoneyControl",

timestamp="2025-04-12 05:00:00"

)

print(article.to\_dict())

## **2. FinancialNewsScraper**

## **Description:**

A class that scrapes financial news from multiple sources concurrently.

## **Methods:**

## **\_get\_moneycontrol\_news() -> List[NewsArticle]**

Fetches news articles from MoneyControl.

## **\_get\_economic\_times\_news() -> List[NewsArticle]**

Fetches market news from Economic Times via RSS feeds.

## **\_get\_livemint\_news() -> List[NewsArticle]**

Fetches market news from Livemint via RSS feeds.

## **\_get\_business\_standard\_news() -> List[NewsArticle]**

Scrapes market-related articles from Business Standard.

## **get\_all\_news() -> List[Dict]**

Fetches news from all sources concurrently and returns a list of articles sorted by timestamp. Returns top 15 recent articles.

## **Example:**

Python

class FinancialNewsScraper:

def \_\_init\_\_(self):

self.headers = {

"User-Agent": "Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/91.0.4472.124 Safari/537.36"

}

def \_get\_moneycontrol\_news(self) -> List[NewsArticle]:

try:

urls = [

"https://www.moneycontrol.com/news/business/markets/",

"https://www.moneycontrol.com/news/business/stocks/"

]

articles = []

for url in urls:

response = requests.get(url, headers=self.headers, timeout=10)

soup = BeautifulSoup(response.text, 'html.parser')

for article in soup.select("li.clearfix"):

try:

title\_elem = article.find("h2")

if not title\_elem:

continue

title = title\_elem.text.strip()

link = title\_elem.find("a")["href"] if title\_elem.find("a") else None

if link:

# Get full article content

article\_response = requests.get(link, headers=self.headers, timeout=10)

article\_soup = BeautifulSoup(article\_response.text, 'html.parser')

content\_div = article\_soup.find("div", {"class": "content\_wrapper"})

content = " ".join([p.text.strip() for p in content\_div.find\_all("p")]) if content\_div else title

articles.append(NewsArticle(

title=title,

content=content,

url=link,

source="MoneyControl"

))

except Exception as e:

print(f"Error processing MoneyControl article: {str(e)}")

continue

return articles

except Exception as e:

print(f"Error fetching MoneyControl news: {str(e)}")

return []

def \_get\_economic\_times\_news(self) -> List[NewsArticle]:

try:

# ET RSS feed for markets

rss\_url = "https://economictimes.indiatimes.com/markets/rssfeeds/1977021501.cms"

feed = feedparser.parse(rss\_url)

articles = []

for entry in feed.entries[:10]: # Get top 10 articles

try:

# Get full article content

response = requests.get(entry.link, headers=self.headers, timeout=10)

soup = BeautifulSoup(response.text, 'html.parser')

content\_div = soup.find("div", {"class": "artText"})

content = " ".join([p.text.strip() for p in content\_div.find\_all("p")]) if content\_div else entry.summary

articles.append(NewsArticle(

title=entry.title,

content=content,

url=entry.link,

source="Economic Times",

timestamp=entry.published

))

except Exception as e:

print(f"Error processing ET article: {str(e)}")

continue

return articles

except Exception as e:

print(f"Error fetching Economic Times news: {str(e)}")

return []

def \_get\_livemint\_news(self) -> List[NewsArticle]:

try:

# Livemint RSS feed for markets

rss\_url = "https://www.livemint.com/rss/markets"

feed = feedparser.parse(rss\_url)

articles = []

for entry in feed.entries[:10]:

try:

response = requests.get(entry.link, headers=self.headers, timeout=10)

soup = BeautifulSoup(response.text, 'html.parser')

content\_div = soup.find("div", {"class": "mainArea"})

content = " ".join([p.text.strip() for p in content\_div.find\_all("p")]) if content\_div else entry.summary

articles.append(NewsArticle(

title=entry.title,

content=content,

url=entry.link,

source="Livemint",

timestamp=entry.published

))

except Exception as e:

print(f"Error processing Livemint article: {str(e)}")

continue

return articles

except Exception as e:

print(f"Error fetching Livemint news: {str(e)}")

return []

def \_get\_business\_standard\_news(self) -> List[NewsArticle]:

try:

url = "https://www.business-standard.com/markets"

response = requests.get(url, headers=self.headers, timeout=10)

soup = BeautifulSoup(response.text, 'html.parser')

articles = []

for article in soup.select(".article-list li"):

try:

title\_elem = article.find("h2")

if not title\_elem:

continue

title = title\_elem.text.strip()

link = "https://www.business-standard.com" + title\_elem.find("a")["href"] if title\_elem.find("a") else None

if link:

article\_response = requests.get(link, headers=self.headers, timeout=10)

article\_soup = BeautifulSoup(article\_response.text, 'html.parser')

content\_div = article\_soup.find("div", {"class": "article-content"})

content = " ".join([p.text.strip() for p in content\_div.find\_all("p")]) if content\_div else title

articles.append(NewsArticle(

title=title,

content=content,

url=link,

source="Business Standard"

))

except Exception as e:

print(f"Error processing Business Standard article: {str(e)}")

continue

return articles

except Exception as e:

print(f"Error fetching Business Standard news: {str(e)}")

return []

def get\_all\_news(self) -> List[Dict]:

"""Fetch news from all sources concurrently"""

with concurrent.futures.ThreadPoolExecutor(max\_workers=4) as executor:

# Submit all scraping tasks

future\_to\_source = {

executor.submit(self.\_get\_moneycontrol\_news): "MoneyControl",

executor.submit(self.\_get\_economic\_times\_news): "Economic Times",

executor.submit(self.\_get\_livemint\_news): "Livemint",

executor.submit(self.\_get\_business\_standard\_news): "Business Standard"

}

all\_articles = []

for future in concurrent.futures.as\_completed(future\_to\_source):

source = future\_to\_source[future]

try:

articles = future.result()

all\_articles.extend([article.to\_dict() for article in articles])

except Exception as e:

print(f"Error fetching news from {source}: {str(e)}")

# Sort articles by timestamp if available

all\_articles.sort(

key=lambda x: datetime.strptime(x["timestamp"], "%a, %d %b %Y %H:%M:%S %z") if x["timestamp"] else datetime.now(),

reverse=True

)

return all\_articles[:15] # Return top 15 most recent articles

def scrape\_moneycontrol():

"""Legacy function for compatibility"""

scraper = FinancialNewsScraper()

return scraper.get\_all\_news()

## **3. is\_related(news\_entities: List[str], fund\_name: str) -> bool**

## **Description:**

Uses fuzzy string matching to check if any entity in the list of news\_entities is related to the given fund\_name. A match is considered valid if the similarity score exceeds 80%.

## **Parameters:**

* news\_entities (List[str]): A list of entities (e.g., keywords or topics) extracted from a news article.
* fund\_name (str): The name of the fund or topic to check for relevance.

## **Returns:**

bool: True if any entity matches the fund name with a similarity score > 80, otherwise False.

## **Example:**

python

from scraper import is\_related

entities = ["Sensex", "Nifty", "Mutual Fund"]

fund = "Nifty 50 Index Fund"

result = is\_related(entities, fund)

print(result)

## **4. process\_article(article: Dict) -> Dict**

## **Description:**

Processes a news article to generate a summary, extract named entities, and analyze sentiment.

## **Parameters:**

* article (Dict): A dictionary representing a news article with a "content" key.

## **Returns:**

Dict: A dictionary containing the summary, entities, and sentiment analysis results.

## **Logic:**

1. Summarization:
   * Uses the transformers pipeline for text summarization, truncating the article content to the first 1000 characters.
2. Entity Recognition:
   * Uses spacy to extract named entities from the article content.
3. Sentiment Analysis:
   * Uses the transformers pipeline for sentiment analysis, truncating the article content to the first 512 characters.

## **Example:**

Python

import spacy

from transformers import pipeline

nlp = spacy.load("en\_core\_web\_sm")

summarizer = pipeline("summarization")

sentiment\_analyzer = pipeline("sentiment-analysis")

def process\_article(article):

summary = summarizer(article["content"][:1000])[0]["summary\_text"]

ents = [ent.text for ent in nlp(article["content"]).ents]

sentiment = sentiment\_analyzer(article["content"][:512])[0]

return {

"summary": summary,

"entities": ents,

"sentiment": sentiment

}

## **5. openai\_client.py**

This module handles communication with the OpenAI API.

## **Functions:**

## **ask\_openai(question: str, context: str) -> str**

Asks a question to OpenAI with a provided context and returns the response.

## **Parameters:**

* question (str): The financial question to ask.
* context (str): Context from recent news articles.

## **Returns:**

str: The response from OpenAI.

## **Logic:**

1. Prompt Construction:
   * Creates a detailed prompt for the OpenAI API, specifying the role, question, context, and desired response guidelines.
2. API Call:
   * Calls the OpenAI Chat Completions API to generate a response based on the prompt.
3. Error Handling:
   * Handles exceptions that may occur during the API call.

## **Example:**

Python

import os

from openai import OpenAI

from dotenv import load\_dotenv

*# Load environment variables*

load\_dotenv()

*# Initialize OpenAI client*

client = OpenAI(api\_key=os.getenv('OPENAI\_API\_KEY'))

def ask\_openai(question: str, context: str) -> str:

"""

Ask OpenAI a question with context and get a response

"""

try:

*# Create a more flexible prompt*

prompt = f"""You are a senior financial analyst with expertise in stocks, ETFs, and market trends.

Provide a concise, data-driven response to the following question:

Question: {question}

Context from recent news:

{context}

Guidelines for your response:

1. FIRST analyze what type of question this is (about a stock, ETF, sector, or general market concept)

2. For stock-specific questions:

- Focus on that stock's performance, key metrics, and news

- Only mention related funds if they're highly relevant

3. For fund/ETF questions:

- Provide current price, YTD performance, and key holdings

- Explain sector exposures and risk factors

4. Always include:

- Current market data (price, change, volume)

- Relevant timeframes (YTD, 1-month, etc.)

- Clear, actionable insights

5. Structure your response logically based on the question type

6. Be concise but include all key numbers and data points

Important: Do NOT force a rigid structure. Adapt to the question.

If asking about a specific stock, don't list unrelated funds.

"""

*# Call OpenAI API*

response = client.chat.completions.create(

model="gpt-4",

messages=[

{"role": "system", "content": "You are a top financial analyst who provides precise, data-rich responses tailored to each question. You adapt your style based on whether the question is about stocks, ETFs, sectors, or general concepts."},

{"role": "user", "content": prompt}

],

temperature=0.5, *# Lower for more factual responses*

max\_tokens=800

)

return response.choices[0].message.content.strip()

except Exception as e:

print(f"Error calling OpenAI API: {str(e)}")

return "I apologize, but I'm having trouble accessing the market data. Please try again shortly."

## **6. answer\_query(fund: str) -> Dict**

## **Description:**

Answers a specific question about a given fund by scraping related news, summarizing it, and using OpenAI to generate a response.

## **Parameters:**

* fund (str): The name of the fund to query about.

## **Returns:**

Dict: A dictionary containing the fund name, the context used for the query, and the OpenAI response, or a message if no related news is found.

## **Logic:**

1. Scrape News:
   * Scrapes news using the scrape\_moneycontrol() function.
2. Process and Filter News:
   * Processes each news article using process\_article() to get summaries, entities, and sentiment.
   * Filters articles based on relevance to the given fund using is\_related().
3. Create Context:
   * Creates a context string from the summaries of the top 5 related articles.
4. Query OpenAI:
   * Formulates a question about why the fund is down today and uses ask\_openai() to get an answer based on the context.
5. Return Response:
   * Returns a dictionary containing the fund name, the context used (top 3 summaries), and the OpenAI response. If no related news is found, it returns a message indicating this.

## **Example:**

Python

from app.openai\_client import ask\_openai

def answer\_query(fund):

news = scrape\_moneycontrol()

summaries = []

for article in news:

processed = process\_article(article)

save\_article\_to\_db(article, processed)

if is\_related(processed["entities"], fund):

summaries.append(f"- {processed['summary']} (Sentiment: {processed['sentiment']['label']})")

context = "\n".join(summaries[:5]) *# only top 5*

question = f"Why is {fund} down today?"

if context:

gpt\_answer = ask\_openai(question, context)

return {

"fund": fund,

"context\_used": summaries[:3],

"gpt\_response": gpt\_answer

}

else:

return {"fund": fund, "message": "No related news found."}

## **7. StockDataManager**

## **Description:**

A class for managing and analyzing stock and mutual fund data loaded from JSON files.

## **Attributes:**

* stock\_data\_path (str): Path to the stock data JSON file.
* mf\_data\_path (str): Path to the mutual fund data JSON file.
* mf\_holdings\_path (str): Path to the mutual fund holdings JSON file.
* stock\_data (Dict[str, Any]): Dictionary containing stock data.
* mf\_data (Dict[str, Any]): Dictionary containing mutual fund data.
* mf\_holdings (Dict[str, Any]): Dictionary containing mutual fund holdings.

## **Methods:**

## **\_\_init\_\_()**

Initializes the StockDataManager and loads data from JSON files.

## **\_load\_data() -> None**

Loads data from JSON files.

## **get\_stock\_metrics(symbol: str) -> Dict[str, Any]**

Gets metrics for a specific stock using lru\_cache for performance.

## **get\_sector\_performance(sector: str) -> Dict[str, float]**

Gets performance metrics for a specific sector using lru\_cache.

## **get\_relevant\_funds(symbol: str = None, sector: str = None) -> List[Dict[str, Any]]**

Gets relevant mutual funds based on a stock or sector.

## **get\_focused\_analysis(query: str) -> Dict[str, Any]**

Gets focused analysis based on a query, extracting symbols and sectors from the query.

## **\_get\_all\_sectors() -> List[str]**

Gets a list of all sectors using lru\_cache.

## **get\_market\_summary() -> Dict[str, Any]**

Gets an overall market summary.

## **\_get\_top\_movers(limit: int = 5, ascending: bool = True) -> List[Dict[str, Any]]**

Gets the top gaining or losing stocks.

## **get\_stock\_recommendation(symbol: str) -> Dict[str, Any]**

Gets a stock recommendation based on technical and fundamental analysis.

## **Example:**

python

import json

import os

from typing import Dict, Any, List, Optional

from datetime import datetime

from functools import lru\_cache

class StockDataManager:

def \_\_init\_\_(self):

self.stock\_data\_path = "data/stock\_data.json"

self.mf\_data\_path = "data/mutual\_funds\_data.json"

self.mf\_holdings\_path = "data/mf\_holdings\_data.json"

self.stock\_data: Dict[str, Any] = {}

self.mf\_data: Dict[str, Any] = {}

self.mf\_holdings: Dict[str, Any] = {}

self.\_load\_data()

def \_load\_data(self) -> None:

"""Load data from JSON files if they exist."""

try:

if os.path.exists(self.stock\_data\_path):

with open(self.stock\_data\_path, 'r') as f:

self.stock\_data = json.load(f)

if os.path.exists(self.mf\_data\_path):

with open(self.mf\_data\_path, 'r') as f:

self.mf\_data = json.load(f)

if os.path.exists(self.mf\_holdings\_path):

with open(self.mf\_holdings\_path, 'r') as f:

self.mf\_holdings = json.load(f)

except Exception as e:

print(f"Error loading data: {str(e)}")

@lru\_cache(maxsize=100)

def get\_stock\_metrics(self, symbol: str) -> Dict[str, Any]:

"""Get metrics for a specific stock."""

return self.stock\_data.get(symbol, {})

@lru\_cache(maxsize=50)

def get\_sector\_performance(self, sector: str) -> Dict[str, float]:

"""Get performance metrics for a specific sector."""

sector\_stocks = {

symbol: data for symbol, data in self.stock\_data.items()

if data.get('sector') == sector

}

if not sector\_stocks:

return {}

return {

'avg\_returns': sum(stock['returns'] for stock in sector\_stocks.values()) / len(sector\_stocks),

'market\_cap': sum(stock['market\_cap'] for stock in sector\_stocks.values()),

'volume': sum(stock['volume'] for stock in sector\_stocks.values())

}

def get\_relevant\_funds(self, symbol: str = None, sector: str = None) -> List[Dict[str, Any]]:

"""Get relevant mutual funds based on stock or sector."""

relevant\_funds = []

if symbol:

*# Find funds holding this stock*

relevant\_funds.extend([

{

'fund\_name': fund\_name,

'allocation': fund\_data['holdings'].get(symbol, 0),

'nav': fund\_data.get('nav', 0),

'returns': fund\_data.get('returns', 0)

}

for fund\_name, fund\_data in self.mf\_data.items()

if symbol in fund\_data.get('holdings', {})

])

if sector:

*# Find sector-focused funds*

relevant\_funds.extend([

{

'fund\_name': fund\_name,

'sector\_exposure': fund\_data.get('sector\_allocation', {}).get(sector, 0),

'nav': fund\_data.get('nav', 0),

'returns': fund\_data.get('returns', 0)

}

for fund\_name, fund\_data in self.mf\_data.items()

if fund\_data.get('sector\_allocation', {}).get(sector, 0) > 20 *# >20% exposure*

])

return sorted(relevant\_funds, key=lambda x: x.get('returns', 0), reverse=True)[:5]

def get\_focused\_analysis(self, query: str) -> Dict[str, Any]:

"""Get focused analysis based on the query."""

*# Extract symbols and sectors from query*

words = query.upper().split()

symbols = [word for word in words if word in self.stock\_data]

sectors = [word.title() for word in words if word.title() in self.\_get\_all\_sectors()]

analysis = {

'query\_focus': query,

'timestamp': datetime.now().isoformat()

}

if symbols:

symbol = symbols[0] *# Focus on the first mentioned symbol*

stock\_data = self.get\_stock\_metrics(symbol)

analysis.update({

'stock\_data': {

'symbol': symbol,

'current\_price': stock\_data.get('price'),

'change\_percent': stock\_data.get('returns'),

'volume': stock\_data.get('volume')

},

'related\_funds': self.get\_relevant\_funds(symbol=symbol)

})

if sectors:

sector = sectors[0] *# Focus on the first mentioned sector*

sector\_data = self.get\_sector\_performance(sector)

analysis.update({

'sector\_data': {

'name': sector,

'performance': sector\_data.get('avg\_returns'),

'market\_cap': sector\_data.get('market\_cap')

},

'sector\_funds': self.get\_relevant\_funds(sector=sector)

})

return analysis

@lru\_cache(maxsize=1)

def \_get\_all\_sectors(self) -> List[str]:

"""Get list of all sectors."""

return list(set(stock.get('sector') for stock in self.stock\_data.values() if stock.get('sector')))

def get\_market\_summary(self) -> Dict[str, Any]:

"""Get overall market summary."""

if not self.stock\_data:

return {}

total\_market\_cap = sum(stock['market\_cap'] for stock in self.stock\_data.values())

avg\_pe = sum(stock['pe\_ratio'] for stock in self.stock\_data.values()) / len(self.stock\_data)

return {

'total\_market\_cap': total\_market\_cap,

'average\_pe': avg\_pe,

'stock\_count': len(self.stock\_data),

'sectors': list(set(stock['sector'] for stock in self.stock\_data.values())),

'top\_gainers': self.\_get\_top\_movers(limit=5, ascending=False),

'top\_losers': self.\_get\_top\_movers(limit=5, ascending=True)

}

def \_get\_top\_movers(self, limit: int = 5, ascending: bool = True) -> List[Dict[str, Any]]:

"""Get top gaining or losing stocks."""

sorted\_stocks = sorted(

[{'symbol': k, \*\*v} for k, v in self.stock\_data.items()],

key=lambda x: x['returns'],

reverse=not ascending

)

return sorted\_stocks[:limit]

def get\_stock\_recommendation(self, symbol: str) -> Dict[str, Any]:

"""Get stock recommendation based on technical and fundamental analysis."""

stock = self.stock\_data.get(symbol)

if not stock:

return {}

*# Calculate basic metrics*

pe\_ratio = stock['pe\_ratio']

avg\_sector\_pe = sum(s['pe\_ratio'] for s in self.stock\_data.values()

if s['sector'] == stock['sector']) / len(self.stock\_data)

return {

'symbol': symbol,

'current\_price': stock['price'],

'pe\_ratio': pe\_ratio,

'sector\_avg\_pe': avg\_sector\_pe,

'market\_cap': stock['market\_cap'],

'recommendation': 'BUY' if pe\_ratio < avg\_sector\_pe else 'HOLD',

'confidence': min(100, max(0, int((avg\_sector\_pe - pe\_ratio) / avg\_sector\_pe \* 100))),

'technical\_indicators': {

'rsi': stock.get('rsi', 0),

'macd': stock.get('macd', 0),

'volume': stock.get('volume', 0)

}

}

*# Initialize the manager*

stock\_manager = StockDataManager()

## **8. VectorStore**

## **Description:**

A class for storing and retrieving embeddings of user queries and responses, enabling semantic search.

## **Attributes:**

* persist\_path (str): Path to the JSON file where embeddings are persisted.
* embeddings (Dict[str, np.ndarray]): Dictionary storing embeddings, with queries as keys.
* contexts (Dict[str, Any]): Dictionary storing original contexts, with queries as keys.
* max\_history (int): Maximum number of historical entries to store.
* index (faiss.IndexFlatL2): FAISS index for fast similarity search.
* dimension (int): Dimension of the OpenAI embeddings (1536).

## **Methods:**

## **\_\_init\_\_(persist\_path="data/vector\_store.json")**

Initializes the VectorStore, loads persisted data (if any), and initializes the FAISS index.

## **\_init\_faiss()**

Initializes the FAISS index for efficient similarity search.

## **\_load\_persisted\_data()**

Loads persisted data from the JSON file.

## **\_persist\_data()**

Saves data to the JSON file.

## **\_rebuild\_faiss\_index()**

Rebuilds the FAISS index from the current embeddings.

## **create\_embedding(text: str) -> List[float]**

Creates an embedding for a given text using the OpenAI API.

## **add\_context(query: str, response: str, entities: List[str] = None) -> None**

Adds a new context (query and response) with its embedding to the store.

## **\_remove\_context(query: str)**

Removes a context and its embedding from the store.

## **find\_similar\_contexts(query: str, top\_k: int = 3) -> List[Dict[str, Any]]**

Finds the most similar contexts to a given query using the FAISS index.

## **get\_relevant\_entities(query: str) -> List[str]**

Extracts relevant entities from similar contexts found using semantic search.

## **Example:**

Python

import numpy as np

from typing import List, Dict, Any

from openai import OpenAI

import os

import json

import time

import faiss

from dotenv import load\_dotenv

load\_dotenv()

client = OpenAI(api\_key=os.getenv('OPENAI\_API\_KEY'))

class VectorStore:

def \_\_init\_\_(self, persist\_path="data/vector\_store.json"):

self.persist\_path = persist\_path

self.embeddings = {} *# Store embeddings*

self.contexts = {} *# Store original context*

self.max\_history = 10 *# Maximum number of historical entries*

self.index = None *# FAISS index*

self.dimension = 1536 *# OpenAI embedding dimension*

*# Initialize FAISS index*

self.\_init\_faiss()

*# Load persisted data if exists*

self.\_load\_persisted\_data()

def \_init\_faiss(self):

"""Initialize FAISS index for efficient similarity search"""

self.index = faiss.IndexFlatL2(self.dimension)

self.index\_ids = [] *# Store mapping between FAISS indices and query strings*

def \_load\_persisted\_data(self):

"""Load persisted data from disk"""

if os.path.exists(self.persist\_path):

with open(self.persist\_path, 'r') as f:

data = json.load(f)

self.embeddings = {k: np.array(v) for k, v in data['embeddings'].items()}

self.contexts = data['contexts']

*# Rebuild FAISS index*

if self.embeddings:

self.\_rebuild\_faiss\_index()

def \_persist\_data(self):

"""Save data to disk"""

os.makedirs(os.path.dirname(self.persist\_path), exist\_ok=True)

with open(self.persist\_path, 'w') as f:

data = {

'embeddings': {k: v.tolist() for k, v in self.embeddings.items()},

'contexts': self.contexts

}

json.dump(data, f)

def \_rebuild\_faiss\_index(self):

"""Rebuild FAISS index from current embeddings"""

self.\_init\_faiss()

embeddings\_array = np.array(list(self.embeddings.values()))

self.index.add(embeddings\_array)

self.index\_ids = list(self.embeddings.keys())

def create\_embedding(self, text: str) -> List[float]:

"""Create embedding for a given text using OpenAI API"""

response = client.embeddings.create(

model="text-embedding-ada-002",

input=text

)

return response.data[0].embedding

def add\_context(self, query: str, response: str, entities: List[str] = None) -> None:

"""Add new context with its embedding"""

*# Create combined context*

context = {

'query': query,

'response': response,

'entities': entities or [],

'timestamp': time.time()

}

*# Create embedding for the query*

embedding = np.array(self.create\_embedding(query), dtype=np.float32)

*# Store in memory*

self.embeddings[query] = embedding

self.contexts[query] = context

*# Update FAISS index*

self.index.add(embedding.reshape(1, -1))

self.index\_ids.append(query)

*# Maintain history limit*

if len(self.embeddings) > self.max\_history:

oldest\_query = min(self.contexts.items(), key=lambda x: x[1]['timestamp'])[0]

self.\_remove\_context(oldest\_query)

*# Persist data*

self.\_persist\_data()

def \_remove\_context(self, query: str):

"""Remove context and its embedding"""

if query in self.embeddings:

del self.embeddings[query]

del self.contexts[query]

self.\_rebuild\_faiss\_index()

def find\_similar\_contexts(self, query: str, top\_k: int = 3) -> List[Dict[str, Any]]:

"""Find most similar contexts to the query using FAISS"""

query\_embedding = np.array(self.create\_embedding(query), dtype=np.float32)

*# Search in FAISS index*

distances, indices = self.index.search(query\_embedding.reshape(1, -1), top\_k)

*# Get similar contexts*

similar\_contexts = []

for idx in indices[0]:

if idx < len(self.index\_ids): *# Ensure index is valid*

query