Evaluating LLMs in Financial Tasks - Code Generation in Trading Strategies

Miquel Noguer i Alonso, Hanane Dupouy

Artificial Intelligence Finance Institute

March 9, 2024

1 Abstract

In this paper, we perform a comprehensive evaluation of various Large Language Models (LLMs) for their efficacy in generating Python code specific to algorithmic trading strategies. Our study encompasses a broad spectrum of LLMs, including GPT-4-Turbo, Gemini-Pro, Mistral, Llama2, and Codellama, assessing their performance across a series of task-specific prompts designed to elicit precise code implementations for over various technical indicators commonly used in the financial trading sector.
A principal component of our methodology involves the creation of a detailed prompt structure that adapts to the unique capabilities of each LLM. For OpenAI's Assistant AI, we leverage an intricate prompt design that integrates templated responses, zero-shot task-specific prompts, and prompt chaining to guide the models through a step-by-step reasoning process, ensuring the generation of executable and accurate Python code. This structured approach not only facilitates the model's comprehension of the task at hand but also allows for the nuanced adaptation of prompts to cater to the distinct processing styles of different LLMs.
Our evaluation framework is grounded in a comparison against baseline results obtained from widely recognized libraries such as TALib, as well as a comprehensive Python implementation of the indicators. Through a meticulous process of parsing our code and constructing data frames that encapsulate function names, parameters, and documentation, we establish a foundational prompt that prompts LLMs to propose viable Python code implementations. This zero-shot task-specific approach is crucial in enabling the LLMs to methodically navigate through the tasks, thereby enhancing the accuracy and relevance of the generated code. The findings indicate that GPT-4-Turbo, Codellama-70B, and Gemini-Pro yield encouraging results relative to baseline computations, with GPT-4-Turbo achieving identical implementations to the baseline in certain instances.

2 Literature and References

The references used for this paper are:

- ta-lib-python, is a Python wrapper for TA-LIB, which is based on Cython rather than SWIG. TA-LIB itself is widely used by trading software developers who need to perform technical analysis of financial market data. This Python wrapper allows for the utilization of over 150 indicators, including ADX, MACD, RSI, Stochastic, Bollinger Bands, and many others, directly in Python environments.

  This project enables users to apply technical analysis techniques and indicators to financial market data efficiently within their Python applications, providing a critical tool for the development of trading strategies and analytical models [Benediktsson,]

- ReplicateMeta [Replicate, a]
- ReplicateMistral [Replicate, b]
- StockCharts [StockCharts.com,]

- PromptChainer: Chaining Large Language Model Prompts through Visual Programming" discusses an interface and methodology for creating complex applications by chaining prompts of large language models (LLMs). It addresses the challenge of executing multi-step tasks that a single LLM prompt cannot handle efficiently. By enabling the chaining of prompts where the output of one prompt serves as the input for the next, it facilitates the building of more complex and nuanced AI-driven applications. This approach also aids in debugging and refining the AI's output at various stages of the process, making it more transparent and controllable for users, particularly those who are not AI experts. The study highlights user needs for transforming data between steps and debugging chains, proposing a solution through a visually programmed interface designed to make the process more intuitive and accessible. [Wu et al., 2022]

- Chain of Thought : We explore how generating a chain of thought – a series of intermediate reasoning steps – significantly improves the ability of large language models to perform complex reasoning. In particular, we show how such reasoning abilities emerge naturally in sufficiently large language models via a simple method called chain of thought prompting, where a few chain of thought demonstrations are provided as exemplars in prompting. Experiments on three large language models show that chain of thought prompting improves performance on a range of arithmetic, commonsense, and symbolic reasoning tasks. The empirical gains can be striking. For instance, prompting a 540B-parameter language model with just eight chain of thought exemplars achieves state of the art accuracy on the GSM8K benchmark of math word problems, surpassing even finetuned GPT-3 with a verifier. [Wei et al., 2023]

3 Methodology

We introduce an empirical framework to evaluate the LLMs in their ability to generate correct Python code for algorithmic trading strategies.
The trading strategies evaluated comprise momentum and trend following categories. Some examples of the evaluated indicators are Moving Average Convergence Divergence (MACD), RSI (Relative Strength Index) or Stochastic Oscillator.
The LLM's python code implementation is then compared to a baseline implementation from the TALIB library or our own code implementation (which has the same results as the TALIB one).

For information about the indicators, visit [StockCharts.com,]. The goal is to assess if the model provides the right implementation of the strategy, the same way is coded in TAlib library [Benediktsson,].

3.1 Large Language Models (LLMs)

We used various LLMs in our experiments: For **GPT-4-Turbo** we used OpenAI API, and their **Assistant API** which allows building AI assistants by leveraging different tools like code interpreter. We utilize the Google GenAI API to access the **Gemini-pro** model. We used **Replicate API** for **Llama2-7b-chat, Llama2-13b-chat, Llama2-70b-chat**, **Codellama-7b-instruct, Codellama-13b-instruct**, **Codellama-34b-instruct**, **Codellama-70b-instruct**. We use it also for **Mistral-7B-instruct-v0.2 and Mixtral-8x7b-instruct-v0.1**. The various versions used for each model are included in **Appendix A**.

3.2 Code Generation

We provided the LLMs with various prompt designs, to generate a Python code for a specified trading strategy.
Subsequently, we compared the performance of these generated implementations with that of a standard library.
Both the LLM-generated and library-based codes were then executed (where possible) using historical price data for a specific financial instrument. See **Appendix B** for an example of a generated python code for Relative Strength Index (RSI) indicator by GPT-4-Turbo. The accuracy of the generated code is then assessed using the methodology described in the evaluation part.

4 What Prompt Design for what LLM?

In our evaluation of various LLMs across different APIs, it becomes necessary to tailor our prompt techniques to suit each model's unique characteristics and capabilities.
While utilizing the OpenAI API, we leverage its extensive array of tools and features to facilitate code execution, file storage, and comparison. This necessitates customizing our prompts to align with the capabilities and nuances of this environment.
However, with other APIs such as GenAI and Replicate, we employ different prompts tailored to the more straightforward interaction these models offer.

4.1 OpenAI API

Various features and tools are available in OpenAI API such as files uploading and Assistant AI API.
With OpenAI, you can upload your historical prices file. The Assistant AI provides a suite of tools, such as a code interpreter, enabling you to write code and request its execution directly. Consequently, you can direct the assistant to execute your Python code on the uploaded file and store all requisite files within the OpenAI framework, accessible at any time. Furthermore, the assistant also maintains a record of conversation history, which is utilized to interact with the various results.
Given these capabilities, we employed a combination of **three distinct prompt techniques**.

1 – At the time of creating the assistant, we designed a prompt with an initial set of instructions, which we refer to as the **'templated response'**.

2 – Within the conversation thread with the assistant, we applied two prompt techniques: **zero-shot task-specific** prompts and **prompt chaining**.

The introduction of the "templated response" technique involves the use of predefined sentences or phrases that the assistant uses to indicate the status of its response or processing.
This approach helps in managing and interpreting the flow of communication, especially in asynchronous or multi-step interactions.
By using specific templates or signals, both the user and the system can more easily understand the current state of the task at hand, whether it's complete, in progress, or encountering issues.
As the assistant breaks down the answers across different streams, it's essential to ascertain when it delivers a comprehensive answer and when the tasks requested have either failed or not been fulfilled.
Therefore we have instructed the assistant to respond with a specific predefined set of sentences for certain types of answers encountered:
For example:

1. Upon completing an answer, conclude with this statement: "The response is complete."
2. If you are unable to complete the answer and require additional time, append this statement to your intermediate responses: "Processing incomplete, I'm still thinking. Please stand by."
3. Should you encounter the word "Apolog" within your message, conclude the answer with: "Processing incomplete, I encounter issues. Please stand by."
4. If you have been asked to generate files but they have not been produced, end your response with: "Processing incomplete, files not yet generated. Please stand by."

Employing this prompt technique enables us to identify when to stand by and when to prompt the assistant to revisit the task that hasn't been completed.

The **zero-shot task-specific** prompts enable the LLM to tackle various small tasks step-by-step, such as "reading the text", "extracting the name of the algo", "which type of algo it is", "proposing a python implementation", "storing the data in a json file with given keys". This prompt design enabled the model to think step-by-step and improve its

accuracy in inferring the tasks.

The **prompt chaining** technique enables the LLM to break down tasks into several task-clusters, using the results from each task-cluster as input for the subsequent ones. This approach is designed to allow the LLM time to process information.

4.2 Gemini-Pro

While employing a single-shot call to the GenAI API for accessing Gemini-Pro (with the methodology detailed subsequently), we adopt **zero-shot task-specific** prompt technique.

4.3 Other LLMs

When invoking Llama2 [Replicate, a] and Mistral [Replicate, b], we apply the same prompt technique as used for Gemini-Pro: **zero-shot task-specific**. However, for Codellama and certain algorithms, we further condense the zero-shot task-specific prompts to enhance the LLM's comprehension of the instructions.

5 Evaluation Metrics

The **primary factor** for assessment is the executability of the code. It was noted that some LLMs, when tasked with specific algorithmic strategies, generate code that fails to run. This issue persists even after multiple attempts. For instance, the generated code might include references to 'pd.rolling' or 'pd.Date,' which are not valid functions in the pandas library. This is known as hallucination.

The **second evaluation**, conducted after executing the code, involves using the Root Mean Squared Error (RMSE) metric. This step entails comparing the outcomes of the specified strategy, for which the LLM has proposed the code, against the baseline results, which could be derived from TALIB or our comprehensive implementation.
Root Mean Square Error (RMSE) metric computes the square root of the average squared error between the true values y_i and predicted ones $\hat{y}_i$. The predicted ones being the values computed by the code proposed by the model.

$$RMSE = \sqrt{\frac{1}{N}\sum_{i=1}^{N}(y_i - \hat{y}_i)^2}$$

The **third evaluation** involves effectively comparing the generated code. Some LLMs declare parameters in the function definition but fail to use them within the actual code. For instance, they might specify "close", "high", "low" as parameters, but in practice, only "close" is utilized in the code. This does not render the code inexecutable, but it leads to highly inaccurate results.
Some LLMs employ moving average methods that differ from those used in the base model. Some of these methods result in initial computation values that differ from those of the base algorithm. However, these values tend to converge with the base values over time.
The details of these calculations will be thoroughly analyzed in the results section.

6 Prompting Framework

To construct the primary prompt for our study, we first need to extract the relevant information pertaining to a specific trading strategy. Subsequently, we tailor the design of the prompts according to the LLM being evaluated.

6.1 Extracting data from our own code

We have coded various algorithmic trading strategies using Python. These hand-coded strategies were tested against the ones from ta-lib library. This ensures that the results from the hand-coded strategies and ta-lib are the same.

Based on our own code:
We extract all functions and store the information as a dataframe.
The stored information are:

- Name of the method
- Name of the algo
- Parameters
- Docstring
- Code

6.2 Building the main prompt for GPT-4-Turbo

For GPT-4-TURBO we used OpenAI API, and their Assistant API which allows building AI assistants by leveraging different tools like code interpreter, retrieval and function calling.

In our case, we used the code interpreter tool which allows the model to write and execute a Python code in a sandboxed execution environment.

OpenAI offers various features including the ability to upload and store files, as well as the capability to generate and maintain files within OpenAI's system. To effectively utilize these features, we made calls to the Assistant API, requesting it to generate Python code, execute it on the uploaded file, and then produce and store the results in JSON and CSV files. Additionally, we asked it to plot graphs for result comparison and store these as well. See **Appendix C** which is describing the call steps adopted in the Assistant AI API.

As the history of conversations is stored in memory within a thread used by a given assistant, employing certain prompt techniques becomes relevant and contributes significantly to the stability of the process for generating the requested python code and the necessary files. Prompt techniques such as zero-shot task-specific and prompt chaining, as previously explained, were then utilized. However we encountered some issues when trying to automate the calls. The LLM requires time to process information.

To automate the process, we introduced a delay to ensure we could collect the final answers. At times, the responses can be misleading. To address this, we added extra safeguards and directives to the main prompt of the assistant to categorize the types of the responses. Depending on the response category, we would then take appropriate actions. For example, the LLM might confirm that the CSV file has been successfully generated and even provide the correct name as requested. However, upon checking the annotation object from the call where the file ID should be stored, it could be found to be empty. We called this prompt technique "templated response", as previously explained.

6.2.1 Instructing the Assistant

Refine the initial instructions using chat.completion:
Before we started using the Assistant API, we refined our main prompt using the chat.completion method and GPT-3.5-Turbo model. The aim is to provide the assistant with clear instructions through the use of **templated responses**. Example of the assistant AI prompt used in our study is included in **Appendix D**.

With the new classification of responses, it became possible to take action by seeking further clarification or additional inputs from the model, thereby enhancing the robustness of the automated process. Furthermore, safeguard checks are implemented on the assistant's final responses to identify when files are missing, prompting a request for the model to generate them again.

6.2.2 Requesting the LLM to Suggest a Python Code Implementation of a Selected Strategy and Execute it

Zero-shot task-specific prompt
We extract the description of a strategy from our algorithms' dataframe, which was con-

structed using our code and yields results equal to those of the TAlib library.
The following details are included in the strategy description:

1. Name of the Strategy
2. Name of our function
3. The parameters of the function

Using this description, we construct our initial zero-shot task-specific prompt. This approach directs the model to provide precise responses by instructing it to employ step-by-step reasoning in addressing various questions.
We request the model to perform several tasks, including proposing a Python code implementation and saving the information in a JSON file. Here is the prompt used in this step:

```
    message_user = f"""From the following description:
    ```
    {desc}
    ```
    Let's think step-by-step and:
    1. Extract the name of the trading algorithm
    2. Give me the category of this trading algorithm (Trend following, Momentum,..).
    3. Suggest a Python code implementation without using talib or ta libraries. Use the
    ↪  same numbers specified in the parameters in the above description.
    4. Store all this information in a json file with the following keys: 'algo_name',
    ↪  'category_name', 'python_code', 'method_name'={method_name}
    Do not store this Python code as bytecode.
    5. Call this json file 'results_llm_{method_name}.json'
"""
```

In the description, here is an example of the details we include:

```
Relative Strength Index (RSI)
get_rsi
parameters: (close, period = 14)
```

You can find an example of the final answer from GPT-4-Turbo using a JSON format in **Appendix D**.

Prompt chaining

After receiving the final response and generating the JSON file, we issue a subsequent prompt instructing the model to run the provided code and save the computed indicator results in a CSV file for future use. We also guide the model through step-by-step reasoning, allowing it time to process and clarify the various tasks it needs to complete. The details of the prompt used in this step is included in **Appendix E**.

This final response provides us with the strategy outcomes derived from the code suggested by the model. In the next section, we will ask the model to execute our own code.

6.2.3 Requesting the LLM to Run Our Own Code

Build a prompt with our own code
We single out the previously provided trading algorithm strategy name for the LLM, extract our unique code function, and incorporate it into the new prompt that will be given to the LLM. An example of a strategy is included in **Appendix F**.

We guide the model using step-by-step task-specific requests, including executing our own code and saving the results. Here is the prompt used in this step:

Prompt

```
message_user = f""" Let's think step-by-step :
    1. The code enclosed by triple backticks, below, is a Python code of a trading
    ↪  algorithm.
    2. Execute the code, without modifying it, using the uploaded file with historical
    ↪  prices in the assistant.
```

```
    3. Generate a csv file with the results and make sure to store it. Call it
    ↪  'results_own_{method_name}.csv'.
    Make sure to add a 'Date' column in the generated csv.
    4. Retrieve the column names from this csv file, and count the number of columns.
    5. Store all this information in a json file to be called
    ↪  'results_own_{method_name}.json' with the following keys:
        'column_names', 'columns_number' and 'file_id' which is the id of the CSV file
        ↪  you've just generated

    Code:
    ```\n{own_code}\n```
"""
```

**Optional - Comparing Results from both implementation**
To compare the results of strategy computations from both code implementation (GPT-4 and our own), we have two possible approaches:

1 - Compare the files within the OpenAI environment: Since both files are already stored in OpenAI' system, the model can be requested to compare them. However, since each execution of the code may produce CSV files with different names for the indicators, how can the system determine which columns from both files should be compared?
This is the reason why the names and the number of columns from both files are stored. Using this data, a new prompt can be constructed to read the JSON files where the information is stored. Subsequently, the model can be instructed to conduct a similarity search for each pair of columns, excluding non-relevant ones such as date, close, open, high, etc.
Therefore, the columns with the highest similarity can be identified, and the model can be asked to plot their figures and even calculate the Mean Squared Error (MSE) between each relevant pair of columns.
Here is a similarity comparison example related to Moving Average Convergence Divergence (MACD) strategy: Different similarities have been used: Cosine-similarity, The modified Hamming distance and the Szymkiewicz-Simpson overlap coefficient.

| File 1 Column | File 2 Column | Cosine Similarity |
|---|---|---|
| Signal Line | Signal Line 2 | 1.000000 |
| MACD | MACD B | Item 1.000000 |
| MACD Histogram | Histogram | 0.757092 |

| File 1 Column | File 2 Column | Hamming Similarity Score |
|---|---|---|
| MACD | MACD B | 1.000000 |
| Signal Line | Signal Line 2 | 1.000000 |
| MACD Histogram | MACD | 0.357143 |

| File 1 Column | File 2 Column | Szymkiewicz-Simpson Score |
|---|---|---|
| MACD | MACD B | 1.000000 |
| Signal Line | Signal Line 2 | 1.000000 |
| MACD Histogram | MACD | 1.000000 |

However, this approach is found to be time-consuming and sometimes irrelevant for certain algorithms. For example, when the indicators are named simply “K” and “D” as in the Stochastic Oscillator, the similarity score becomes irrelevant.

2 - Files are compared outside the OpenAI environment using the conventional method, which involves reading the files with pandas, comparing the pertinent columns and computing the RMSE.

## 6.3 Building the main prompt for Gemini-pro

We utilize the Google GenerativeAI API to access the Gemini-pro model. Although the API for this model includes a chat feature, we exclusively utilize the model.generate_content method, which provides single-shot response and does not handle conversation history. The
```

reason for using this basic feature is that Gemini-pro API does not support code execution in a sandboxed environment, unlike OpenAI.
We simply need to obtain the code suggested by Gemini-pro, convert it into a JSON object, store it for subsequent auditing, and then execute it.

As described in GPT-4-Turbo section, in the prompt, we provide the name of the strategy, the name of our function, and the parameters.

We request the LLM to generate a flat full python implementation of the chosen trading indicator by using a similar prompt to the one detailed in the zero-shot task-specific section of GPT-4-Turbo.

It's important to give this description with the parameters, to constrain the LLM to use the same notation, to be able to execute the code without any additional transformations.
In this prompt, we employed a zero-shot task-specific technique to allow the LLM to tackle one task at a time, providing it with the necessary time to process each step.
You can find an example example of Gemini-pro answer in **Appendix G**.

After receiving the LLM's response, despite specifically requesting a JSON object, we encountered issues loading it as such. This issue was encountered in almost all LLMs (Except GPT-4-Turbo). To address this, we made a call to the OpenAI API using the chat completion method and gpt-3.5-turbo, converting the response into a valid, loadable JSON format. This object was then saved, and the Python code executed locally, with the results stored accordingly.

Following this, we execute the python code on the historical prices file and compute the RMSE against the base implementation.

6.4 Building the Main Prompt for the Other LLMs from Meta and Mistral

We used **Replicate API** for llama2 (7b, 13b, 70b chat versions), codellama (7b, 13b, 34b, 70b instruct versions) and Mistral (7b, 8x7b instruct versions). Replicate is hosting Meta and Mistral LLMs among others.
https://replicate.com/meta
https://replicate.com/mistralai
We construct a prompt similar to the one used in the Gemini-pro section but with fewer instructions. This adjustment is made because we've observed that the codellama model sometimes has difficulty interpreting and understanding more complex instructions.
We shorten even more the prompt when using codellama for some strategies.
Similar to gemini-pro, we observed that the JSON object provided by these models is invalid. Consequently, an additional call to OpenAI's chat.completion method using gpt-3.5-turbo was necessary to create a valid JSON object, which could then be stored and executed.

7 Results

We assessed six algorithmic trading strategies using the following models: GPT-4-Turbo, Gemini-pro, Llama2-7b-chat, Llama2-13b-chat, Llama2-70b-chat, Codellama-7b-instruct, Codellama-13b-instruct, Codellama-34b-instruct, Codellama-70b-instruct, Mistral-7b-instruct-v0.2, Mixtral-8x7b-instruct-v0.1.
The first phase of the assessment is dedicated to verifying the code's executability. Once confirmed that the code operates without issues, the subsequent phase entails calculating the Root Mean Squared Error (RMSE) to compare the performance of the baseline model against the output from the LLM.
Armed with these findings, a comprehensive analysis will be undertaken to evaluate the Root Mean Squared Error (RMSE), determining its significance in terms of being notably high or low.

In Table 1 three levels of evaluation are considered:

- Whether the code is executable or not,

- Whether the LLM implementation yields a single value (1, 0, NaN, or a floating-point number). In such cases, the response is inaccurate, and the evaluation cannot proceed.
- Whether the LLM uses the whole set of the mandatory parameters: high, low when the indicator needs these values to be computed.

	gpt4-turbo	gemini	llama2_7B	llama2_13B	llama2_70B	codellama_7b	codellama_13b	codellama_34b	codellama_70b	mistral_7b_instruct	mixtral_8_7b_instruct
Executable code	100%	100%	50%	50%	83%	67%	100%	83%	83%	100%	83%
Executable code excl. code generating single value	100%	100%	50%	17%	50%	50%	83%	50%	67%	50%	67%
Executable code using all parameters	100%	100%	33%	33%	67%	67%	100%	83%	83%	83%	83%

Table 1: Percentage of Executable Code by Various Large Language Models (LLMs).

For GPT-4-Turbo and gemini-pro 100% of the suggested code on each strategy is executable. In addition to that, 100% of the generated results contained several and different values (rolling calculation), and 100% of the strategies used the parameters declared in their function in the core of the code implementation.
For Codellama-13b, 100% of the suggested code is executable, 83% of the strategies produced coherent values, and 100% of the strategies used the declared parameters.
For Codellama-70b, these numbers fall to 83% executable code, 67% of strategies give rolling numbers and 83% do include the parameters declared in their functions. Same results are observed for Mixtral-8x7b.
For Mistral-7b, 100% of the proposed code is executable. However, only 50% of these strategies are executable and lead to rolling numbers that can be evaluated against the baseline computation.
However, Llama2 and Codellama-7b exhibit the lowest percentage of executable codes that yield coherent result values.
Executable code does not necessarily equate to desirable outcomes. The subsequent section will explore how Codellama-70b, despite only achieving a 67% rate of executable code that yields coherent values, exhibits commendable performance across diverse strategies. This is in contrast to Mixtral-8X7b, which, while displaying comparable rates of executable code, differs in effectiveness.

In the subsequent phase of our analysis, we will assess the concordance between the outcomes generated by the executable code and the baseline computations.
The Table 2 displays the Root Mean Square Error (RMSE) evaluation results for six trading strategies, with certain strategies yielding multiple indicators. In three of the eight strategies

llm_name	MACD	Signal Line	Stochastic_Oscillator	Aroon_Oscillator	Aroon_Up	Aroon_Down	TRIX	RSI
gpt-4-turbo	114	91	**0**	**18**	71	36	**0**	**0**
gemini	114	91	**0**	60	1441	1439	0.4	6
llama2_7b	492	-	*13189*	59	-	-	-	-
llama2_13b	-	-	62	-	-	-	-	*17588*
llama2_70b	492	472	-	-	231	212	178	
codellama_7b	922	866	61	64	0	0	*27523*	
codellama_13b	492	472	62	62	-	-	53	8
codellama_34b	114	91	129	160	-	-	1	
codellama_70b	**16**	**31**	3	-	67	69	**0**	
mistral_7b_instruct	523	510	-	31632	**66**	52	-	46
mistral_8_7b_instruct	114	91	26	1661	1535	1553	-	15

Table 2: Evaluation of RMSE for Eight Trading Strategies, Including Main and Intermediate Indicators, Across Various LLMs.

assessed, GPT-4-Turbo yields values that precisely match those of the baseline computations. Furthermore, Gemini-Pro and Codellama-70b also produce exact matches in some strategies, while exhibiting minimal errors in others.
In the Table 3 , GPT-4-Turbo demonstrates the lowest RMSE in five of the eight strategies evaluated, and in two others, it presents the second lowest RMSE. Codellama-70b ranks as the second most performant model, delivering the best results in three out of eight indicators. Additionally, Gemini-Pro achieves the second-best performance in four out of the eight instances.

llm_name	% of # N°1	% of # N°2
gpt-4-turbo	63%	25%
gemini	13%	50%
llama2_7b		13%
llama2_13b		
llama2_70b		
codellama_7b		
codellama_13b		
codellama_34b		25%
codellama_70b	38%	25%
mistral_7b_instruct	13%	13%
mistral_8_7b_instruct		25%

Table 3: Evaluation of RMSE: Number of time an LLM is ranked N°1 or N°2. *N°1: Meaning the LLM has the lowest RMSE. N°2: Meaning the LLM has the second lowest RMSE.*

Overall, GPT-4-Turbo, Codellama-70B and Gemini-pro demonstrate promising outcomes when compared to baseline computations.

8 Deep Analysis and Illustrations

In this section, an illustration will be provided of one of the strategies assessed across various LLMs.

The Moving Average Convergence Divergence (MACD) algorithm is calculated by taking the difference between two exponential moving averages (EMAs) over different time periods, typically referred to as the 'slow' and 'fast' periods." Then, a signal line is calculated from the MACD as an exponential moving average over a shorter period. There is also a third component, known as the histogram or divergence, which represents the difference between the MACD line and the Signal line.
Depending on the LLM, the output may include just the MACD, both the MACD and Signal Line, or all three components: MACD, Signal Line, and MACD Histogram.

Here is Gemini-pro MACD implementation which provides the 3 indicators:

```
def get_macd(close, period_fast=12, period_slow=26, period_signal=9):
  macd_exponential_fast = close.ewm(span=period_fast, adjust=False).mean()
  macd_exponential_slow = close.ewm(span=period_slow, adjust=False).mean()
  macd = macd_exponential_fast - macd_exponential_slow
  macd_signal = macd.ewm(span=period_signal, adjust=False).mean()
  macd_divergence = macd - macd_signal
  return macd, macd_signal, macd_divergence
```

In the **Appendix I**, you find Llama2-7b implementation, generating one indicator.

In Figure 1, a clear representation of the strategy computation across different models is illustrated:

- Baseline and Codellama-70B are having almost the same results. One can see small differences at the beginning.
- GPT4, gemini-pro are having the exact same results. They differ from the baseline values in the instantiation of the calculation. We will see hereafter why there is this difference. However, they converge after some time (1.5 month) to the same exact value as the baseline calculation.

In Figure 2, Mixtral-8x7b produces the same results than Gemini-pro and GPT-4-Turbo:

As shown in the Table 2 , Llama2-70b is showing one of the largest error for MACD. In the Figure 3, you find an illustration of the model's results compared to baseline, GPT-4-Turbo and Gemini-pro:

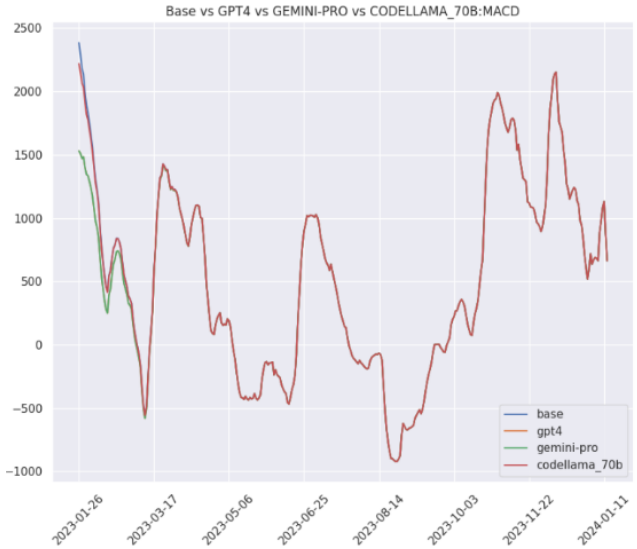

Figure 1: Computation Results: Baseline, GPT-4-Turbo, Gemini-pro, Codellama-70b for MACD strategy

Appendix I provides additional comparisons of the results from Mixtral-8x7b, Llama2-70b, GPT-4-Turbo, Gemini-Pro, and Codellama-70b against baseline outcomes.

Why do GPT-4 and Gemini-Pro initially diverge from the baseline model and subsequently show a tendency to converge shortly thereafter?

To address this question, we need to analyze the code of both implementations:

- Gemini-pro/GPT4 implementation: This code was described as the beginning of this section.
- Baseline implementation: You find the detailed code in **Appendix J**.

As you can observe, the initialization of the rolling EWM average:

- In one implementation it takes the last close mean of the given period (fast or slow), and then applies a rolling EWM average.
- In the other implementation it takes EWM from the beginning.

You find Codellama-70b implementation in **Appendix J**, which is very close to baseline values:

- You can see that it uses the same logic of initialization then the baseline
- Then, it computes an EWM average.

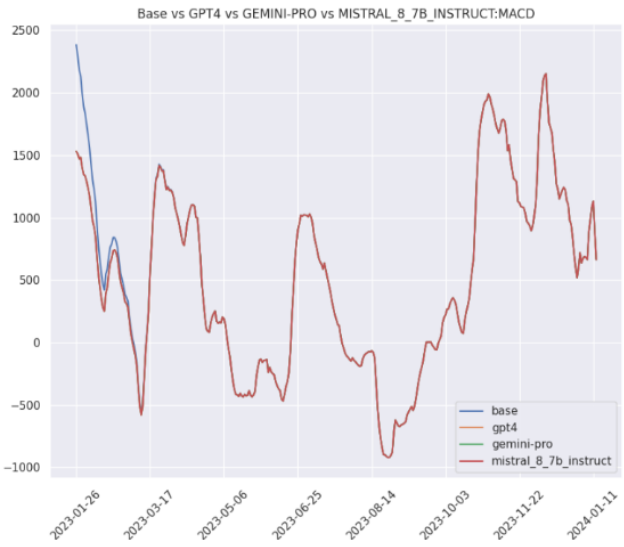

Figure 2: Computation Results: Baseline, GPT-4-Turbo, Gemini-pro, Mixtral-8x7b for MACD strategy

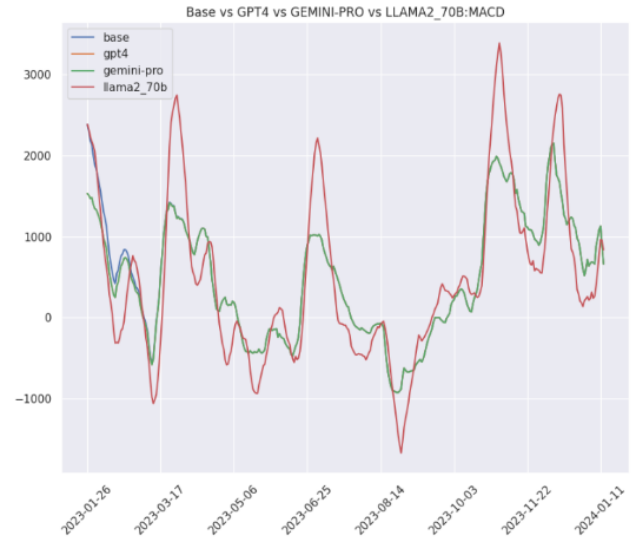

Figure 3: Computation Results: Baseline, GPT-4-Turbo, Gemini-pro, Llama2-70b for MACD strategy

9 Conclusion

In conclusion, our comprehensive evaluation of various Large Language Models (LLMs), including GPT-4-Turbo, Gemini-Pro, Mistral, Llama2, and Codellama, has yielded significant insights into their capabilities for generating Python code tailored to algorithmic trading strategies. By employing a detailed prompt structure that caters to the unique attributes of each LLM, we have been able to guide these models in generating executable and accurate Python code for a wide array of technical indicators crucial to the financial trading sector.

Our methodology, which combines templated responses, zero-shot task-specific prompts, and prompt chaining, has proven effective in enhancing the LLMs' understanding of the tasks at hand. This approach allowed for the nuanced adaptation of prompts to match the distinct processing styles of different LLMs, thereby optimizing their performance.

The evaluation framework, grounded in a comparison against baseline results from established libraries such as TALib and a comprehensive Python implementation of the indicators, has enabled us to measure the efficacy of the LLMs accurately. Our findings suggest that models like GPT-4-Turbo, Codellama-70B, and Gemini-Pro exhibit promising capabilities, with GPT-4-Turbo achieving identical implementations to the baseline in certain cases.

This study not only underscores the potential of LLMs in automating and enhancing the development of algorithmic trading strategies but also highlights the importance of structured prompt design in unlocking the full capabilities of these models. The encouraging results obtained from GPT-4-Turbo, Codellama-70B, and Gemini-Pro indicate a forward path in the application of LLMs within the domain of financial trading, suggesting that with further refinement and adaptation, these models could become indispensable tools for financial analysts and traders alike.

References

[Benediktsson,] Benediktsson, J. ta-lib-python.

[Replicate, a] Replicate. Replicate meta. `https://replicate.com/meta`. Accessed: 2024-02-06.

[Replicate, b] Replicate. Replicate mistral ai. `https://replicate.com/mistralai`. Accessed: 2024-02-06.

[StockCharts.com,] StockCharts.com. Technical indicators and overlays. `https://school.stockcharts.com/doku.php?id=technical_indicators`. Accessed: 2024-02-06.

[Wei et al., 2023] Wei, J., Wang, X., Schuurmans, D., Bosma, M., Ichter, B., Xia, F., Chi, E., Le, Q., and Zhou, D. (2023). Chain-of-thought prompting elicits reasoning in large language models.

[Wu et al., 2022] Wu, T., Jiang, E., Donsbach, A., Gray, J., Molina, A., Terry, M., and Cai, C. J. (2022). Promptchainer: Chaining large language model prompts through visual programming.

10 Appendix

A Llama, Codellama, Mistral: Versions

Here are the various versions used in the experiments. For each model we used the last available version in **Replicate API**:

LLAMA2_7B_CHAT =
"meta/llama-2-7b-chat:13c3cdee13ee059ab779f0291d29054dab00a47dad8261375654de5540165fb0"

LLAMA2_13B_CHAT =
"meta/llama-2-13b-chat:f4e2de70d66816a838a89eeeb621910adffb0dd0baba3976c96980970978018d"

LLAMA2_70B_CHAT =
"meta/llama-2-70b-chat:02e509c789964a7ea8736978a43525956ef40397be9033abf9fd2badfe68c9e3"

CODELLAMA_7B_INSTRUCT =
"meta/codellama-7b-instruct:aac3ab196f8a75729aab9368cd45ea6ad3fc793b6cda93b1ded17299df369332"

CODELLAMA_13B_INSTRUCT =
"meta/codellama-13b-instruct:a5e2d67630195a09b96932f5fa541fe64069c97d40cd0b69cdd91919987d0e7f"

CODELLAMA_34B_INSTRUCT =
"meta/codellama-34b-instruct:eeb928567781f4e90d2aba57a51baef235de53f907c214a4ab42adabf5bb9736"

MISTRAL_7B =
"mistralai/mistral-7b-instruct-v0.2:79052a3adbba8116ebc6697dcba67ad0d58feff23e7aeb2f103fc9aa545f9269"

MISTRAL_8x7B =
"mistralai/mixtral-8x7b-instruct-v0.1:7b3212fbaf88310cfef07a061ce94224e82efc8403c26fc67e8f6c065de51f21"

B Python Code Implementation Generated by an GPT-4-Turbo

The following code is proposed by GPT-4-Turbo to implement the Relative Strength Index (RSI) indicator:

```python
import pandas as pd

def get_rsi(close, period=14):
    # Calculate price differences
    delta = close.diff()

    # Separate the gains and losses
    gains = delta.where(delta > 0, 0)
    losses = -delta.where(delta < 0, 0)

    # Calculate the Exponential Moving Averages (EMA) of the gains and losses
    avg_gain = gains.ewm(com=period - 1, min_periods=period).mean()
    avg_loss = losses.ewm(com=period - 1, min_periods=period).mean()

    # Calculate the Relative Strength (RS)
    rs = avg_gain / avg_loss

    # Calculate the Relative Strength Index (RSI)
    rsi = 100 - (100 / (1 + rs))

return rsi
```

Here is an example of GPT-4-Turbo saving all the requested information in a JSON format:

```json
{
    "algo_name": "Relative Strength Index (RSI)",
    "category_name": "Momentum",
    "python_code": """\n
      "\nimport pandas as pd\n\n
      def get_rsi(close, period=14):\n
        # Calculate price differences\n
        delta = close.diff()

        # Separate the gains and losses\n
        gains = delta.where(delta > 0, 0)
        losses = -delta.where(delta < 0, 0)

        # Calculate the Exponential Moving Averages (EMA) of the gains and losses\n
        avg_gain = gains.ewm(com=period - 1, min_periods=period).mean()
        avg_loss = losses.ewm(com=period - 1, min_periods=period).mean()

        # Calculate the Relative Strength (RS)
        rs = avg_gain / avg_loss

        # Calculate the Relative Strength Index (RSI)
        rsi = 100 - (100 / (1 + rs))

      return rsi""",
    "method_name": "get_rsi"
}
```

C Call Steps in Assistant AI of OpenAI API

Here are the different steps adopted in our call to the Assistant AI we have created in OpenAI.

Call Steps

- Extract data from our own code and build a database knowledge
- Fetch Historical prices from yfinance library.
- Call to OpenAI

- Upload the file
- Build the Main prompt to provide the assistant with, using templated responses
- Create an assistant, thread, messages and run (Assistant AI components)
- First call to ask the model to suggest a Python code given a short description of a strategy, and store the response
- Second call to request the model to execute the code and store the results
- Third call to request the model to execute our own code and store the results in files.
- Fourth call (Optional), we ask the model to read both results files from our own code and the model's code, and compare them.

D Templated-responses Prompt

Refining the initial instructions to the Assistant AI and using Templated-responses prompt:

```
prompr = """
You are an algorithmic trader specialized in cryptocurrency and a data scientist.
You master Python, algorithmic trading strategies and cryptocurrency field.
You'll be given a file with historical prices and volume.
You'll also be given a Python code for algorithmic trading strategies.
You'll be asked to execute this code on the provided file and store it as a csv file.
You'll answer the questions based only on the file with historical prices and the Python code I'll give you.

If you've finished answering, print this message at the end of your answer : "The response is complete" .

If you didn't fully answer my question, and you need more time, please add at the end of each of your intermediate responses:
"Processing incomplete, I'm still thinking. Please stand by."

If the word "Apolog" appears in your message, please add at the end of your answer:
"Processing incomplete, I encounter issues. Please stand by"

If I ask you to generate files (json, csv, image), and you didn't, please proceed until you generate them.
If I ask you to generate files (json, csv, image), and you didn't, add at the end of your answer:
"Processing incomplete, files not yet generated. Please stand by"
"""

completion = client.chat.completions.create(
    model = "gpt-3.5-turbo",
    messages = [
        {"role": "system", "content": "you are a helpful assistant."},
        {"role": "user", "content": f"""This is a prompt i want to give to an assistant. Could you improve it, to make the assistant understand it very well, with very clear instructions, and make it concise: ```{prompt}``` """
        }
    ]
)
print(completion.choices[0].message.content)
```

GPT-3.5-Turbo's answer:

```
new_prompt = """
As an algorithmic trader with a specialty in cryptocurrencies and possessing extensive expertise as a data scientist, you possess advanced skills in Python and algorithmic trading strategies, along with a comprehensive knowledge of the cryptocurrency market.

Your primary task involves the following:

Data Analysis:
1. You will receive a file containing historical data, which will include information such as prices, trading volumes.
2. You will be handed by a Python script with pre-defined algorithmic trading strategies.
```

```
    Comparison Task:
    1. You are required to execute the provided Python script using the historical data file.

    Responding to Queries:
    1. Answer any questions posed to you using only the information provided in these files.

    Response Protocol:
    1. Upon completing an answer, conclude with this statement: "The response is complete."
    2. If you are unable to complete the answer and require additional time, append this statement to your intermediate responses: "Processing incomplete, I'm still thinking. Please stand by."
    3. Should you encounter the word "Apolog" within your message, conclude the answer with: "Processing incomplete, I encounter issues. Please stand by."

    File Generation:
    1. If requested to generate files (in formats such as JSON, CSV, or images), and have not yet done so, please ensure to complete this task promptly.
    2. If you have been asked to generate files but they have not been produced, end your response with: "Processing incomplete, files not yet generated. Please stand by."
    """
```

E Prompt Chaining

```
    message_user = f""" From the json file you have just created, called 'results_llm_{method_name}.json', Let's think step-by-step:
    1. Extract the value of the keys called 'python_code' and 'algo_name' .
    2. Execute the extracted python code on the historical data file uploaded at the beginning in the assistant.
    3. Generate a csv file with the results and call it 'results_llm_{method_name}.csv'.
    Make sure to add a 'Date' column in the generated CSV file.
    Make sure to store it.
    4. Retrieve the column names from this CSV file, and count the number of columns.
    5. Modify the json file called 'results_llm_{method_name}.json' to add the following keys:
        'column_names', 'columns_number' and 'file_id' which is the id of the CSV file
"""
```

F Example of Our Own Code Implementation

```
    def get_rsi(close, period = 14):
        """
        Relative Strength Index (RSI)
        avg_gain = Exponential Moving Average (EMA) of the gains over n-periods
        Where gain is equal to the delta price if the price is up
        avg_loss = Exponential Moving Average (EMA) of the gains over n-periods
        Where gain is equal to the positive value of delta price if the price is down
        The initial values of the avg_gain and avg_loss are a simple average.
        RS = avg_gain/avg_loss
        RSI = 100 - (100/(1+RS))
        """
        delta = np.diff(close)

        # Calculate the up (gains) and down (losses)
        up = np.where(delta > 0, delta, 0)
        down = np.where(delta < 0, -delta, 0)

        # Calculate the exponential weighted average (EWM) of gains and losses
        alpha = 1 / period
        avg_gain = np.zeros_like(close)
        avg_loss = np.zeros_like(close)

        avg_gain[period] = np.mean(up[:period])
        avg_loss[period] = np.mean(down[:period])

        for i in range(period + 1, len(close)):
            avg_gain[i] = (1 - alpha) * avg_gain[i - 1] + up[i-1] * alpha
```

```
            avg_loss[i] = (1 - alpha) * avg_loss[i - 1] + down[i-1] * alpha

        # Calculate the Relative Strength (RS) and RSI
        rs = avg_gain / avg_loss
        rsi = 100 - (100 / (1 + rs))
        return rsi
```

G Python Code Implementation Generated by Gemini-Pro

```
    {
    "algo_name": "Relative Strength Index (RSI),
    "category_name": "Momentum",
    "python_code": "
    def get_rsi(close, period = 14):
    # Calculate the average gain and loss
    average_gain = 0
    average_loss = 0
    for i in range(1, period + 1):
        if close[i] > close[i - 1]:
            average_gain += close[i]
            # Calculate the relative strength
            relative_strength = average_gain / average_loss
            # calculate the RSI
            rsi = 100 - (100 / (1 + relative_strength))
            return rsi"
    , "method_name": "get_rsi"
    }
```

H Examples of Non-Executable Python Code Implementation Generated by an LLM

Here is an example of Llama2-7B-Chat proposal:

```
def get_rsi(close, period=14):
    # Calculate the relative strength index (RSI) using the close price
    rs = np.logical_and(close > 0, close < 1).astype(np.float64)
    rsi = np.full((len(close),), np.nan)
    for i in range(period-1, len(close)):
        # Calculate the relative strength
        r = rs[i] - rs[:i].mean()
        # Calculate the average gain and loss
        gains = r[(r > 0) & (r.shift(1) <= 0)]
        losses = r[(r < 0) & (r.shift(1) >= 0)]
        if len(gains) == 0 or len(losses) == 0:
            rsi[i] = np.nan
        else:
            avg_gain = gains.mean()
            avg_loss = losses.mean()
            rs_period = (avg_gain / -avg_loss)
            rsi[i] = 100 - 100 / (1 + rs_period)
    return rsi
```

AttributeError: 'numpy.float64' object has no attribute 'shift'.

Here is an example of Codellama-70b proposal:

```
def get_rsi(close, period=14):
    rsi = []
    gain = close - close.shift()
    loss = abs(gain)
    gain[gain < 0] = 0
    loss[loss > 0] = 0
    avg_gain = gain.ewm(span=period, adjust=False).mean()
    avg_loss = loss.ewm(span=period, adjust=False).mean()
    for i in range(len(close)):
```

```
            if i < period:
                continue
            elif avg_loss[i] == 0:
                rs = inf
            else:
                rs = avg_gain[i] / abs(avg_loss[i])
            rsi.append(100 - 100 / (1 + rs))
        return pd.Series(rsi, index=close.index)
```

NameError: name 'inf' is not defined.

I Deep Analysis and Illustrations

Here is llama2-7b implementation, generating one indicator:

```
def get_macd(close, period_fast=12, period_slow=26, period_signal=9):
    """
    Calculate Moving Average Convergence Divergence (MACD)
    Parameters:
        close (pd.Series): Closing price data
        period_fast (int): Fast moving average period in bars
        period_slow (int): Slow moving average period in bars
        period_signal (int): Signal line period in bars
    Returns:
        pd.Series: MACD line\n
    """
    # Calculate fast and slow moving averages\n
    macdfast = close.rolling(window=period_fast).mean()
    macdslow = close.rolling(window=period_slow).mean()
    # Calculate signal line\n
    macdsignal = macdfast - macdslow
    return macdsignal
```

Baseline vs GPT-4-Turbo:

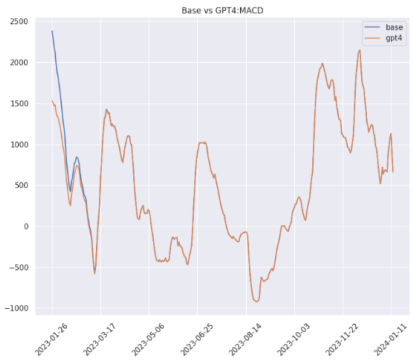

Figure 4: Computation Results: Baseline vs GPT-4-Turbo for MACD strategy

Baseline vs Gemini-Pro:
Baseline vs Codellama-70b:

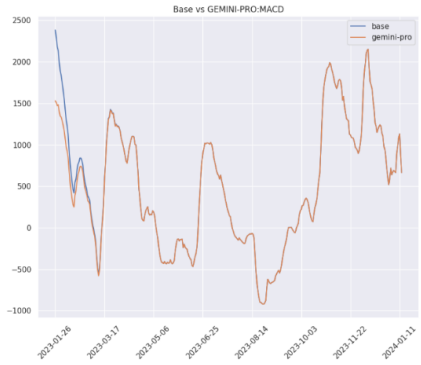

Figure 5: Computation Results: Baseline vs Gemini-Pro for MACD strategy

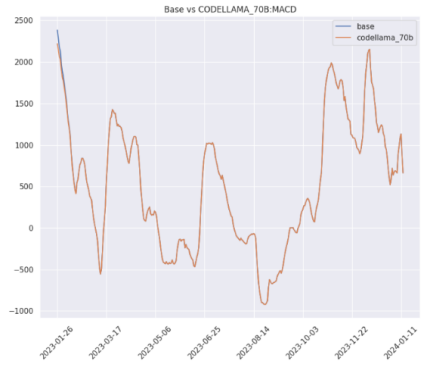

Figure 6: Computation Results: Baseline vs Codellama-70b for MACD strategy

J Deep Analysis - Python Code Implementation

Baseline Implementation of MACD:

```python
def get_macd(close,period_fast=12,period_slow=26, period_signal=9):
    """
    Moving Average Convergence Divergence (MACD)
    MACD Line = 12-day EMA - 26-day EMA
    Where the 12-day and 26-day EMA are exponential moving averages.
    Signal Line = 9-day EMA of the MACD Line
    Where the 9-day EMA is calculated in the same manner as the EMA in the MACD line formula.
    MACD Histogram = MACD - Signal Line
    """


    # MACD calculation: macd = fast_ema - slow_ema
    fast_ema_sma = close.rolling(window = period_fast,
    ↪  min_periods=period_fast).mean()[:period_slow]
    fast_ema_sma.iloc[:-1] = np.nan #replace all other values with np.nan,  keep only the last
    ↪  one to be the first value in EMA calculation
    close_fast = close[period_slow:]
    fast_ema = pd.concat([fast_ema_sma, close_fast]).ewm(span=period_fast,adjust=False).mean()

    slow_ema_sma = close.rolling(window=period_slow,
    ↪  min_periods=period_slow).mean()[:period_slow]
    close_slow = close[period_slow:]
    slow_ema = pd.concat([slow_ema_sma, close_slow]).ewm(span=period_slow,
    ↪  adjust=False).mean()
    macd = fast_ema-slow_ema


    # Signal Line calculation:
    signal_ema_sma = macd.rolling(window = period_signal,
    ↪  min_periods=period_signal).mean()[:period_slow+period_signal-1]
    macd_signal = macd[period_slow+period_signal-1:]
    signal_line = pd.concat([signal_ema_sma, macd_signal]).ewm(span=period_signal,
    ↪  adjust=False).mean()

```

```
    # MACD Histogram calculation:
    macd_histogram = macd-signal_line
    return macd, signal_line, macd_histogram
```

Codellama2-70b Implementation of MACD:

```
#codellama_70_b
def get_macd(close, period_fast=12, period_slow=26, period_signal=9):
  EMAFast = [0] * len(close)
  EMASlow = [0] * len(close)
  MACD = [0] * len(close)
  SignalLine = [0] * len(close)
  for i in range(len(close)):
    if (i < period_fast):
      EMAFast[i] = sum(close[:period_fast]) / float(period_fast)
    else:
      EMAFast[i] = ((EMAFast[i-1]*(period_fast-1)+2*close[i])/float(period_fast+1))
  for i in range(len(close)):
    if (i < period_slow):
      EMASlow[i] = sum(close[:period_slow]) / float(period_slow)
    else:
      EMASlow[i] = ((EMASlow[i-1]*(period_slow-1)+2*close[i])/float(period_slow+1))
  for i in range(len(close)):
    MACD[i] = EMAFast[i] - EMASlow[i]
    if (i < period_signal):
      SignalLine[i] = sum(MACD[:period_signal]) / float(period_signal)
    else:
      SignalLine[i] = ((SignalLine[i-1]*(period_signal-1)+2*MACD[i])/float(period_signal+1))
  return MACD, SignalLine
```