Optimization Requirements

GoQuant Internship Assignment

1. Memory Management

Implemented Techniques

```
import gc
gc.set_threshold(700, 10, 10)  # garbage collection threshold in
    ↪ Streamlit
...
gc.collect()  # Memory management - trigger collection at end of
    ↪ main loop
```

- **Custom garbage collection thresholds** are set to control the frequency of automatic garbage collection. This helps in reducing latency due to memory cleanup during execution.
- **Explicit garbage collection** is invoked after the main loop using `gc.collect()` to ensure any unused memory is freed efficiently.
- **Minimal state retention**: The application avoids memory bloat by not storing historical orderbook snapshots. It maintains only the **latest orderbook data**:

```
orderbook_data = {
    "bids": data.get("bids", []),
    "asks": data.get("asks", [])
}
```

- This dictionary is updated with each L2 snapshot, ensuring minimal memory footprint and no growth over time.

2. Network Communication

Implemented Techniques

```
lock = threading.Lock()
data = json.loads(message)
```

- **Threaded WebSocket client**: Runs in a separate daemon thread to keep the UI responsive and prevent blocking.
- **Efficient parsing**: Only extracts required fields `("bids", "asks")` from the incoming L2 snapshot, minimizing memory usage and deserialization cost.
- The use of:
  - `threading.Lock()` ensures safe access to shared variables like `orderbook_data`.
  - `json.loads()` directly parses only the essential fields.
- Overall, the WebSocket client is optimized to:
  - Receive only lightweight L2 updates.
  - Minimize bandwidth by not subscribing to full market depth or trade streams.
  - Handle data updates without blocking the UI.

3. Data Structure Selection

Implemented Techniques

```
idx = np.searchsorted(bins, shares)
return np.array(inventory_path), np.array(trajectory)

orderbook_data = {
    "bids": data.get("bids", []),
    "asks": data.get("asks", [])
}
```

- **NumPy arrays** are used for efficient numerical operations and vectorization. This significantly reduces CPU time in simulation and path optimization.
- **Dictionaries** are used for storing and retrieving orderbook data and slippage metrics:
  - Fast key-based access.
  - Thread-safe sharing with `Lock()`.
- **Efficient usage of list/dict combinations** ensures fast parsing and updates of incoming data without unnecessary overhead.
- **Only relevant data fields** are stored, and structures are reused instead of duplicated across calls.

4. Thread Management

Implemented Techniques

```
threading.Thread(target=run, daemon=True).start()
time.sleep(1.5)  # thread management optimization
```

- **Daemon threads**: Automatically exit when the main program ends. Ideal for background WebSocket handling.
- **Controlled sleep intervals** prevent aggressive reconnection or CPU overuse.
- **Minimal shared state**: Only critical shared structures like `orderbook_data` and `last_slippage` are exposed to the thread, guarded with locks.
- **Avoid thread bloat**: Only a single thread handles all real-time data. No unnecessary parallel threads are spawned.

5. Regression Model Efficiency

Implemented Techniques

- **Preload model and scaler** at Streamlit startup:

  ```
  model = joblib.load("logistic_model_regression.pkl")
  scaler = joblib.load("logistic_scaler.pkl")
  ```

- The model is loaded once and stored in memory to avoid repeated I/O and latency per prediction.
- **No retraining or file reads** occur during runtime. This makes inference efficient and responsive.
- Inputs are scaled with a preloaded `StandardScaler`, ensuring consistency and fast transformation.
- **Vectorized prediction pipeline**: Inputs are processed in NumPy array form, enabling batch prediction if needed.