Backtesting Framework for a Smart Order Router (SOR)

Salim Aissaoui

January 5, 2025

Contents

1 Part 1: Methodology and Framework

1.1 1. Introduction

A Smart Order Router (SOR) aims to optimally execute trades across multiple venues, minimizing market impact, slippage, and costs. A robust backtesting framework ensures that such strategies can be tested against historical or synthetic market data prior to production deployment. Key benefits include:

- Evaluating algorithmic performance under realistic conditions.
- Comparing different strategies (TWAP, VWAP, etc.) on equal footing.
- Identifying system weaknesses in order routing logic.

1.2 2. Data Pipeline

Data Sources and Processing

- **Synthetic Data:** Generated via random walk models for price and randomized volumes.
- **Handling Missing Data:** Either imputation or removing incomplete rows.
- **Synchronization:** Indexing by timestamps ensures consistent time-series alignment.

Storage and Scalability

- For small-scale, in-memory `pandas` DataFrames suffice.
- For large-scale or multi-venue data, consider databases or distributed computing.

1.3 3. Execution Strategies

TWAP (Time-Weighted Average Price) Splits orders evenly over predetermined time intervals, providing a consistent, time-based approach.

VWAP (Volume-Weighted Average Price) Allocates slices based on market volumes. A dynamic variant (as in this framework) continually re-estimates volumes to allocate future slices.

RL-Based (Conceptual Skeleton) Demonstrates an example RL-inspired policy that buys whenever the current price is below a running average. In real-world systems, a more sophisticated approach with comprehensive state/action/reward definitions is necessary.

1.4 4. Performance Metrics

- **Execution Cost** vs. Benchmark (e.g., difference from market VWAP).
- **Slippage**: Difference between a reference (arrival) price and fill prices.
- **Fill Rate**: (Optionally) fraction of the target quantity successfully executed.

1.5 5. Simulation Logic

Multi-Venue (Future Extension)

- Distinct order books for each venue.
- SOR logic deciding where and when to place slices.

Partial Fills

- If an order is larger than the available liquidity at a price level, partial fills occur.

Transaction Costs

- Exchange fees, maker-taker rebates, or crossing fees can be incorporated to measure net performance.

1.6 6. Extensibility

- **Complex Orders:** Bracket orders, conditional orders, stop-limit, etc.
- **Advanced ML/RL Approaches:** Multi-agent systems, LOB-based RL, etc.
- **Scenario Testing:** Stress tests with high volatility, low liquidity, or correlated instruments.

1.7 7. Conclusion

This framework provides a foundation for backtesting various SOR strategies. By integrating additional features like multi-venue routing and advanced RL models, it can be expanded into a comprehensive research environment for order execution strategies.

2 Part 2: Code Implementation and One-Page Report

2.1 Python Script with Documentation

```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt

# ----------------------------
# 1. Generate Synthetic Market Data
# ----------------------------
def generate_synthetic_data(num_points=100, initial_price=100.0, volatility=0.02):
    """
    Generate synthetic price data using a simple random walk model.

    :param num_points: Number of data points (e.g., 100)
    :param initial_price: Starting price
    :param volatility: Random walk volatility factor
    :return: DataFrame indexed by timestamp with columns 'price' and 'volume'
    """
    np.random.seed(42)
    timestamps = pd.date_range(start="2023-01-01", periods=num_points, freq="T")

    prices = [initial_price]
    volumes = []

    # Build the price series via random walk
    for i in range(1, num_points):
        price_change = np.random.randn() * volatility
        new_price = prices[-1] * (1 + price_change)
        prices.append(max(1, new_price))
        volumes.append(np.random.randint(10, 1000))

    prices = np.array(prices)
    volumes = np.array([np.random.randint(10, 1000) for _ in range(num_points)])

    df = pd.DataFrame({
        "timestamp": timestamps,
        "price": prices,
        "volume": volumes
    })
    df.set_index("timestamp", inplace=True)
    return df

# ----------------------------
# 2. TWAP Strategy
# ----------------------------
def twap_execution(data, total_quantity, start_time, end_time, freq="5T"):
    """
```

```python
    Executes a TWAP strategy by slicing orders equally over the time period.

    :param data: DataFrame with 'price' column indexed by timestamp
    :param total_quantity: total quantity to be traded
    :param start_time: start of trading (datetime)
    :param end_time: end of trading (datetime)
    :param freq: frequency of order slices, e.g., "5T" for 5 minutes
    :return: DataFrame of executions [exec_timestamp, exec_price, exec_quantity]
    """
    trade_data = data.loc[start_time:end_time]
    if len(trade_data) == 0:
        raise ValueError("No data in the specified trading window.")

    slice_times = pd.date_range(start=start_time, end=end_time, freq=freq)
    num_slices = len(slice_times)
    slice_quantity = total_quantity / num_slices if num_slices > 0 else total_quantity

    executions = []
    for t in slice_times:
        # If there's no exact match, pick the nearest price
        if t in trade_data.index:
            current_price = trade_data.loc[t, "price"]
        else:
            current_price = trade_data.iloc[trade_data.index.get_loc(t, method='nearest')]["price"]

        executions.append({
            "exec_timestamp": t,
            "exec_price": current_price,
            "exec_quantity": slice_quantity
        })

    exec_df = pd.DataFrame(executions)
    return exec_df

# ---------------------------
# 3. VWAP Strategy with Dynamic Rebalancing
# ---------------------------
def vwap_execution_dynamic(data, total_quantity, start_time, end_time, rebalance_freq="5T"):
    """
    Executes a VWAP-like strategy with dynamic rebalancing intervals.
    At each rebalance interval, the strategy:
      - Observes the realized volume so far
      - Estimates the remaining volume for the rest of the window
      - Reallocates the remaining order quantity accordingly

    :param data: DataFrame with 'price' and 'volume' indexed by timestamp
    :param total_quantity: total quantity to be traded
    :param start_time: start of trading (datetime)
    :param end_time: end of trading (datetime)
```

```
    :param rebalance_freq: frequency at which the strategy recalculates slice sizes
    :return: DataFrame of executions [exec_timestamp, exec_price, exec_quantity]
    """
    trade_data = data.loc[start_time:end_time]
    if len(trade_data) == 0:
        raise ValueError("No data in the specified trading window.")

    rebalance_times = pd.date_range(start=start_time, end=end_time, freq=rebalance_freq)
    exec_records = []

    remaining_quantity = total_quantity
    previous_time = start_time

    for i, current_time in enumerate(rebalance_times):
        if i == 0:
            # Skip the first interval, no data prior to this
            continue

        interval_data = trade_data.loc[previous_time:current_time]
        if len(interval_data) == 0:
            continue

        # Observed volume in this interval
        observed_interval_volume = interval_data["volume"].sum()

        # Remaining intervals
        intervals_left = len(rebalance_times) - i

        # Estimate future volume with naive approach: average so far * intervals left
        if i == 1:
            average_volume_so_far = observed_interval_volume
        else:
            observed_data_so_far = trade_data.loc[start_time:current_time]
            average_volume_so_far = observed_data_so_far["volume"].sum() / i

        estimated_future_volume = average_volume_so_far * intervals_left
        total_estimated_volume = observed_interval_volume + estimated_future_volume

        if total_estimated_volume <= 0:
            continue

        # Fraction of this interval's volume relative to total est. volume
        fraction_of_interval = observed_interval_volume / total_estimated_volume

        # Allocate that fraction of the remaining quantity
        quantity_to_execute = remaining_quantity * fraction_of_interval

```

```
        # Assume execution at average price within this interval
        avg_price_interval = np.average(interval_data["price"], weights=
            interval_data["volume"])

        exec_records.append({
            "exec_timestamp": current_time,
            "exec_price": avg_price_interval,
            "exec_quantity": quantity_to_execute
        })

        remaining_quantity -= quantity_to_execute
        previous_time = current_time

        if remaining_quantity <= 0:
            break

    # Allocate leftover quantity (if any) at the end_time
    if remaining_quantity > 0:
        last_time = rebalance_times[-1]
        final_data = trade_data.loc[last_time:end_time]
        if not final_data.empty:
            final_price = np.average(final_data["price"], weights=
                final_data["volume"])
        else:
            final_price = trade_data.iloc[-1]["price"]
        exec_records.append({
            "exec_timestamp": end_time,
            "exec_price": final_price,
            "exec_quantity": remaining_quantity
        })

    exec_df = pd.DataFrame(exec_records)
    return exec_df

# ----------------------------
# 4. Reinforcement-Learning-Based Routing (Conceptual Example)
# ----------------------------
def rl_smart_routing(data, total_quantity, start_time, end_time):
    """
    A minimal conceptual example of using RL to decide whether to execute
        or hold at each time step.
    This is NOT a production RL solution; it simply buys when
        current_price < running avg.

    :param data: DataFrame with 'price' column indexed by timestamp
    :param total_quantity: total quantity to be traded
    :param start_time: start of trading (datetime)
    :param end_time: end of trading (datetime)
    :return: DataFrame with RL-based execution records.
    """
    trade_data = data.loc[start_time:end_time]
    if len(trade_data) == 0:
        raise ValueError("No data in the specified trading window.")

```

```python
    timestamps = trade_data.index
    exec_records = []
    remaining_quantity = total_quantity

    running_avg_price = trade_data["price"].expanding().mean()

    for i, t in enumerate(timestamps):
        if remaining_quantity <= 0:
            break

        current_price = trade_data.loc[t, "price"]
        avg_price_so_far = running_avg_price.iloc[i]

        # If current price < running average => buy 10% of what's left
        if current_price < avg_price_so_far:
            slice_quantity = remaining_quantity * 0.1
            exec_records.append({
                "exec_timestamp": t,
                "exec_price": current_price,
                "exec_quantity": slice_quantity
            })
            remaining_quantity -= slice_quantity

    # Final fill if any quantity remains
    if remaining_quantity > 0:
        final_price = trade_data.iloc[-1]["price"]
        exec_records.append({
            "exec_timestamp": end_time,
            "exec_price": final_price,
            "exec_quantity": remaining_quantity
        })
        remaining_quantity = 0

    exec_df = pd.DataFrame(exec_records)
    return exec_df

# ----------------------------
# 5. Calculate Metrics (Execution Cost, Slippage)
# ----------------------------
def calculate_metrics(exec_df, market_data, benchmark="VWAP"):
    """
    Calculate performance metrics:
      - Execution Cost relative to a benchmark (VWAP)
      - Slippage relative to the arrival price (first execution)

    :param exec_df: DataFrame of executions
    :param market_data: Full market data to compute VWAP
    :param benchmark: Currently supports only "VWAP"
    :return: Dictionary of metrics
    """
    if len(exec_df) == 0:
        print("No executions found. Cannot calculate metrics.")
        return {}

```

```
    if benchmark == "VWAP":
        total_volume = (market_data["price"] * market_data["volume"]).sum()
        total_shares = market_data["volume"].sum()
        overall_vwap = total_volume / total_shares if total_shares > 0 else market_data["price"].mean()

        avg_exec_price = np.average(exec_df["exec_price"], weights=exec_df["exec_quantity"])
        execution_cost = avg_exec_price - overall_vwap
        expected_price = exec_df.loc[0, "exec_price"] if not exec_df.empty else overall_vwap
        slippage = avg_exec_price - expected_price

        return {
            "Benchmark": benchmark,
            "Overall_VWAP": overall_vwap,
            "Avg_Exec_Price": avg_exec_price,
            "Execution_Cost": execution_cost,
            "Slippage": slippage
        }
    else:
        return {}

# ----------------------------
# 6. Main Simulation
# ----------------------------
if __name__ == "__main__":
    # Generate synthetic data
    df_market = generate_synthetic_data(num_points=200, initial_price=100, volatility=0.01)

    # Parameters
    total_quantity = 1000
    start_time = df_market.index[0]
    end_time = df_market.index[-1]

    # A. Basic TWAP Execution
    exec_twap = twap_execution(df_market, total_quantity, start_time, end_time, freq="5T")
    metrics_twap = calculate_metrics(exec_twap, df_market, benchmark="VWAP")
    print("=== TWAP Strategy Results ===")
    for k, v in metrics_twap.items():
        print(f"{k}: {v:.4f}" if isinstance(v, float) else f"{k}: {v}")

    # B. VWAP with Dynamic Rebalancing
    exec_vwap_dynamic = vwap_execution_dynamic(df_market, total_quantity, start_time, end_time, rebalance_freq="10T")
    metrics_vwap_dynamic = calculate_metrics(exec_vwap_dynamic, df_market, benchmark="VWAP")
    print("\n=== VWAP (Dynamic) Strategy Results ===")
    for k, v in metrics_vwap_dynamic.items():
        print(f"{k}: {v:.4f}" if isinstance(v, float) else f"{k}: {v}")
```

```python

    # C. RL-Based Strategy (Conceptual Example)
    exec_rl = rl_smart_routing(df_market, total_quantity, start_time,
        end_time)
    metrics_rl = calculate_metrics(exec_rl, df_market, benchmark="VWAP")
    print("\n=== RL-Based Strategy Results (Conceptual) ===")
    for k, v in metrics_rl.items():
        print(f"{k}: {v:.4f}" if isinstance(v, float) else f"{k}: {v}")

    # Visualization
    fig, axes = plt.subplots(3, 1, figsize=(10, 12), sharex=True)

    # Plot: TWAP
    axes[0].plot(df_market.index, df_market["price"], label="Price")
    axes[0].scatter(exec_twap["exec_timestamp"], exec_twap["exec_price"],
                    color="red", marker="x", label="TWAP Exec")
    axes[0].set_title("TWAP Executions")
    axes[0].legend()

    # Plot: VWAP Dynamic
    axes[1].plot(df_market.index, df_market["price"], label="Price")
    axes[1].scatter(exec_vwap_dynamic["exec_timestamp"], exec_vwap_dynamic
        ["exec_price"],
                    color="purple", marker="o", label="VWAP (Dynamic) Exec
                        ")
    axes[1].set_title("VWAP (Dynamic) Executions")
    axes[1].legend()

    # Plot: RL-based
    axes[2].plot(df_market.index, df_market["price"], label="Price")
    axes[2].scatter(exec_rl["exec_timestamp"], exec_rl["exec_price"],
                    color="green", marker="D", label="RL Exec")
    axes[2].set_title("RL-Based Executions (Conceptual)")
    axes[2].legend()

    plt.xlabel("Time")
    plt.ylabel("Price")
    plt.tight_layout()
    plt.show()
```

Listing 1: Backtesting Implementation with Docstrings and Comments

2.2 One-Page Report

Implementation Approach

Data Generation We use a random walk model for the price series and random integers for volumes, indexing by minute-level timestamps. This simulates a simplified but realistic market feed.

Execution Strategies

- **TWAP**: Slices the total quantity equally across fixed intervals.
- **VWAP (Dynamic)**: Allocates shares based on observed and forecasted volume, recalculated at each interval.
- **RL-Based (Conceptual)**: Buys when current price is below a running average (a simplified placeholder policy).

Metrics Calculation

- **Overall VWAP**: Weighted average market price.
- **Avg Execution Price**: Weighted average of the actual fill prices.
- **Execution Cost**: Difference between avg execution price and VWAP (lower is better).
- **Slippage**: Difference between the arrival (first) price and the final average execution price.

Results (Example)

```
=== TWAP Strategy Results ===
Benchmark: VWAP
Overall_VWAP: 95.8914
Avg_Exec_Price: 95.7621
Execution_Cost: -0.1293
Slippage: -4.2379

=== VWAP (Dynamic) Strategy Results ===
Benchmark: VWAP
Overall_VWAP: 95.8914
Avg_Exec_Price: 95.5276
Execution_Cost: -0.3638
Slippage: -5.0162

=== RL-Based Strategy Results (Conceptual) ===
Benchmark: VWAP
Overall_VWAP: 95.8914
Avg_Exec_Price: 98.5772
Execution_Cost: 2.6858
Slippage: -0.6647
```

Interpretation

- **VWAP (Dynamic)**: Outperformed TWAP in this run by getting a slightly lower average price than VWAP.
- **RL Strategy**: Underperformed (higher execution cost), indicating that a naive approach can yield subpar results without further optimization.

References

1. **Combining Deep Learning on Order Books with Reinforcement Learning for Profitable Trading** - Focus on temporal-difference learning for return forecasting.

2. **Multi-Agent Reinforcement Learning in a Realistic Limit Order Book Market Simulation** - Focus on agent-based simulation using Double Deep Q-Learning.

3. **Interpretable ML for High-Frequency Execution** - Focus on modeling fill probability with state dependence for execution backtesting.

4. **Deep Reinforcement Learning for Market Making Under a Hawkes Process-Based Limit Order Book Model** - Focus on a DRL-based controller for optimizing order execution under stochastic conditions.