Introduction

The purpose of our trading system is to develop and test a robust and systematic trading strategy using Alpaca's paper trading environment. Our system is designed to simulate real-world trading while ensuring stability and risk control, allowing for effective strategy development without financial exposure.

Market Data Retrieval

The following function retrieves **1-minute interval stock price data** for the **past 30 days**.

1. Import Necessary alpaca-py Libraries

```
import pickle
import os
import time
import schedule
from datetime import datetime, timedelta
from zoneinfo import ZoneInfo
from alpaca.data.requests import StockBarsRequest
from alpaca.data.timeframe import TimeFrameUnit, TimeFrame
import alpaca_trade_api as tradeapi
import logging
import traceback
from requests.exceptions import RequestException
```

2. Initialize the Alpaca API Client

```
api_key = "PKABTN0SFPC9AF832CW4"
secret_key = "t8unjLT9hjtvtC23uTxrL1FEm1l6alM1wDpsCpYc"
BASE_URL = 'https://paper-api.alpaca.markets'
api = tradeapi.REST(api_key, secret_key, base_url=BASE_URL, api_version='v2')

#### We use paper environment for this example ####
paper = True # Please do not modify this. This example is for paper trading only.
####

# Below are the variables for development this documents
# Please do not change these variables
trade_api_url = None
trade_api_wss = None
data_api_url = None
stream_data_wss = None
```

3. Market Retrieval Function

```
# Function to fetch data and update CSV and Pickle files
def fetch_and_save_data():
    now = datetime.now(ZoneInfo("America/New_York"))
    for symbol in symbols:
        try:
            req = StockBarsRequest(
                symbol_or_symbols=[symbol],
                timeframe=TimeFrame(amount=1, unit=TimeFrameUnit.Minute),
                start=now - timedelta(days=7)
            )

            df = stock_historical_data_client.get_stock_bars(req).df
```

4. Examples shown how our market data retrieval function works:

```
symbol = "AAPL"
df = fetch_market_data(symbol)

# Display the first 5 rows of data
print(df.head())
```

5. Sample output:

```
                           open   high    low  close   volume
timestamp
2024-02-10 09:30:00-05:00   153.2  154.0  152.8  153.5    10230
2024-02-10 09:31:00-05:00   153.5  154.3  153.2  153.8     8540
2024-02-10 09:32:00-05:00   153.8  154.1  153.5  153.9     9321
2024-02-10 09:33:00-05:00   153.9  154.5  153.6  154.2     8765
```

Data Storage Strategy

Describe your strategy for saving market data. Discuss the chosen storage method (database or file system), the structure of data storage, and considerations for timestamps and timezones.

1. Storage Method: File System
2. Data Storage Structure

   We store market data in two formats:

   1. **Pickle** – Stores the DataFrame efficiently for quick access.

2. **CSV** – Provides a human-readable format for analysis.

We automate data retrieval using `schedule`, running the function **every minute**.

3. Considerations of Timestamps: Each data entry is stored with a timestamp as the index, ensuring accurate historical tracking.
Time Zones:

1. **Stock market hours follow Eastern Time (ET)**.
2. We use `ZoneInfo("America/New_York")` to ensure **alignment with NYSE/NASDAQ market hours**.

```
# Function to fetch data and update CSV and Pickle files
def fetch_and_save_data():
    now = datetime.now(ZoneInfo("America/New_York"))
    for symbol in symbols:
        try:
            req = StockBarsRequest(
                symbol_or_symbols=[symbol],
                timeframe=TimeFrame(amount=1, unit=TimeFrameUnit.Minute),
                start=now - timedelta(days=7)
            )

            df = stock_historical_data_client.get_stock_bars(req).df

            # Define file paths
            pickle_file_path = os.path.join("/Users/dingyiling/Desktop", f"{symbol}.pkl")
            csv_file_path = os.path.join("/Users/dingyiling/Desktop", f"{symbol}.csv")

            # Save to Pickle
            with open(pickle_file_path, "wb") as f:
                pickle.dump(df, f)

            # Save to CSV
            df.to_csv(csv_file_path, index=True)

            logging.info(f"Updated {symbol}.csv and {symbol}.pkl at {datetime.now()}")
```

```
# Schedule the function to run every hour
schedule.every().minute.do(fetch_and_save_data)

# Keep the script running
while True:
    schedule.run_pending()
    time.sleep(60)  # Check every minute
```

Trading Strategy Development

Capital Allocation Strategy

- We distribute funds strategically to optimize portfolio performance so we can manage the risk and maintain flexibility:
- 50% for ETFs: Ensures diversification and liquidity, primarily trading major index ETFs like SPY and VIS
- 25% for Individual Stocks: Allocated to high-volatility stocks based on short-term trading opportunities
- 25% for Holding/Adjustments: Reserved for dynamic portfolio adjustments, including increasing ETF or stock positions based on market conditions

ETF strategy

We choose to trade two ETFs: SPY 500 and VIS because they have high trading volumes and provide broad market exposure, which helps reduce unsystematic risk. SPY 500 is a more conservative choice as it tracks the S&P 500, which offers exposure to the overall U.S. stock market and ensures stability through diversification.

Additionally, we see an increasing trend in the manufacturing industry recently due to strong economic recovery and increased infrastructure spending. The VIS (Vanguard Industrial ETF) is well-positioned to benefit from this trend as it provides exposure to leading industrial companies, including manufacturers, transportation firms, and construction companies.

Therefore, by trading SPY 500 for market-wide exposure and VIS to capitalize on the industrial sector's growth, we balance stability with growth potential. Also we optimize our portfolio for short-term gains while mitigating downside risk.

Technical Indicators

- Opening Price: Market order executed at open.
- Percentage Change from open price:
    - +0.5% Gain → Full position exit.
    - +0.3% to +0.5% Gain → Partial sell-offs based on dynamic scaling.
    - -1% Drop → Additional buying to cost-average.

This approach enables fast execution and we aim to gain profits from short-term price moves.

Risk Management Rules

- Profit-taking thresholds: Selling in increments (20%-40%-100%) as the price rises.
- Stop-Loss Mechanism: Rebuying ETF shares when price drops 1% below entry price.
- Portfolio Rebalancing: The 25% holding fund is used for adjustments if market conditions shift significantly.

Position Sizing & Trade Execution

- Initial Allocation: Invest 60% of the allocated capital in each ETF at market open
- +0.3% Gain → Sell 20% of holdings
- +0.4% Gain → Sell 40% of holdings

- +0.5% Gain → Exit 100% of the position
- If the price declines by 1%, increase the position by 4% additional units.
- Reset Cycle: Once all units are sold, the strategy repeats from the initial allocation phase

Individual stock strategy

We have selected NVIDIA (NVDA), Starbucks (SBUX), and Citigroup (C) for trading

NVDA: A leader in AI and semiconductor technology, which has high volatility and strong momentum opportunities. Recently, NVDA has experienced a significant decline, dropping approximately 27% from its peak. But historical patterns suggest that this is a good entry point.

SBUX: A consumer discretionary stock. Under CEO Brian Niccol's "Back to Starbucks" plan, the company has focused on enhancing customer experience and operational efficiency. Despite a recent 4.9% decline from its 52-week high, SBUX shares have grown 12.6% over the past three months

C (Citigroup): As a major financial institution, Citigroup's performance is closely tied to interest rate movements and macroeconomic trends. The financial sector's sensitivity to economic indicators makes Citigroup a strategic choice for capturing opportunities arising from monetary policy shifts and economic cycles.

Technical Indicators & Trading Strategy

Our trading strategy is designed to maximize profits while managing risks by leveraging AI-driven stock price predictions. We use LSTM deep learning models to forecast stock prices

Intelligent Entry Points (Buying Strategy)

- 2 minutes before LSTM predicts a local low, buy 20% of the position.
- After 1 minute, observe:
- If the price is below VWAP and RSI < 30, buy another 30%.
- If the price continues to drop, buy the remaining 50% after 5 minutes.
- If the price rebounds by more than 0.5%, stop buying and hold the current position.

Optimized Exit Points (Selling Strategy)

- 2 minutes before LSTM predicts a local high, sell 20% of the position.
- After 1 minute, observe:
- If RSI > 70 and the price is above VWAP, sell another 30%.
- If the price rises by more than 0.5%, sell the remaining 50% after 5 minutes.
- If the price drops by more than 0.3%, immediately sell all remaining shares.

Risk Management Rules

- Stop-loss: If the price drops 2% below the purchase price, immediately sell.
- Take-profit: If the price increases 5%, immediately sell.
- Trailing stop-loss: If the price exceeds the purchase price by 3%, sell if it falls 1% from the highest price.

Code Explanation

Data Loading and Preprocessing

To ensure our trading system is both **reactive to live market conditions** and **capable of refining future strategies**, we implemented two key functions: `get_real_time_market_data(symbol)` and `fetch_and_save_data()`. The first function **retrieves the past six hours of minute-level stock and ETF data** using Alpaca's API. If the data request fails, an error message is logged, ensuring transparency and troubleshooting capability. This step is crucial as it **provides the LSTM model with up-to-date market conditions**, helping it make informed predictions based on recent trends.

```python
def get_real_time_market_data(symbol):
    try:
        barset = api.get_bars(symbol, 'minute', limit=360)
        df = pd.DataFrame([{**bar.t, "symbol": symbol} for bar in
barset[symbol]])
        df.set_index("time", inplace=True)
        return df
    except Exception as e:
        print(f"Failed to fetch real-time data for {symbol}: {e}")
        return None
```

The second function, `fetch_and_save_data()`, **stores daily historical data** to refine the **next day's trading strategy**. By saving price movements, volume, and technical indicators, we ensure that our strategy is not just **reactive** but also **evolves based on past performance**. The function fetches the last seven days of minute-level data, stores them as both **CSV and Pickle files**, and logs the update. This enables efficient backtesting and adjustments based on historical patterns.

```python
df.to_csv(csv_file_path, index=True)  # Save historical data for next-day improvements
```

ETF Trading Strategy

The ETF trading strategy is **price-based and systematic**, designed to **capitalize on market movements while managing risk**. It follows a structured approach with **a fixed entry, gradual profit-taking, and strategic rebuys**. Unlike stock trading, this strategy does not rely on LSTM predictions but instead reacts to **real-time price changes**.

1. Initial Purchase at Market Open

Each day, at **9:30 AM ET**, the system **executes a market order** for each ETF. This ensures that the strategy enters the market consistently, capturing daily trends. The **initial buy amount** is fixed at $15,000 per ETF, and this purchase price (`last_buy_price`) serves as the benchmark for future sell and rebuy decisions

```python
if symbol not in last_buy_price:
    first_buy_price = market_open_price or current_price
    last_buy_price[symbol] = first_buy_price
    holding_etf[symbol] = int(initial_buy_notional / first_buy_price)
```

2. Gradual Selling for Profit-Taking

Instead of selling all holdings at once, the strategy uses **a tiered exit approach** to maximize returns while **protecting gains**.

- **If price rises by 0.5%** → **Sell all** holdings
- **If price rises by 0.4%** → **Sell 40%** of holdings
- **If price rises by 0.3%** → **Sell 20%** of holdings

This **reduces exposure incrementally**, ensuring **profit realization while keeping some shares for further upside**.

```python
if price_change >= 0.005:
    place_order(symbol, qty=holding_etf[symbol], side="sell")
elif price_change >= 0.004:
    place_order(symbol, qty=int(holding_etf[symbol] * 0.4), side="sell")
elif price_change >= 0.003:
    place_order(symbol, qty=int(holding_etf[symbol] * 0.2), side="sell")
```

3. Smart Rebuy Strategy

To take advantage of price dips, the strategy **buys more shares** when the ETF **drops 1% below the last purchase price**.

- **If price falls 1% below last buy price → Buy $1,000 worth of shares**
- This **reduces the cost basis** and allows for **profitable re-entries**.

```python
if holding_etf[symbol] == 0 and price_change <= -0.01:
    rebuy_shares = int(rebuy_notional / current_price)
    place_order(symbol, qty=rebuy_shares, side="buy")
```

Stock Trading Strategy: Prediction-Driven & Gradual Execution

The stock trading strategy is **LSTM-assisted and data-driven**, aiming to **maximize returns through precise buy/sell signals** while managing risk effectively. Unlike ETF trading, this approach **relies on machine learning predictions** to determine optimal entry and exit points. The strategy follows a **three-step process: predictive buying, staged selling, and risk-controlled execution**.

1. Predictive Buying Using LSTM Forecasts

Every **30 minutes**, the system **retrieves real-time stock data, scales it, and feeds it into the LSTM model** to predict the next **30 minutes of price movements**. The system **identifies local minima** in the predictions to determine ideal buy points.

- **If a predicted local minimum is detected → Buy 10% of available cash worth of shares**
- This ensures that **purchases are based on expected price reversals** rather than arbitrary entry points.

```python
buy_times, _, _ = find_trade_times(predicted_prices)
if now.strftime('%H:%M') == trade_times[buy_time].strftime('%H:%M'):
    buy_amount = portfolio_cash[symbol] * risk_percentage
    qty_to_buy = int(buy_amount / predicted_prices[buy_time])
    place_order(symbol, qty=qty_to_buy, side="buy")
```

2. Staged Selling for Profit-Taking

Instead of selling all shares at once, the system **executes partial sells at key profit points**, similar to the ETF strategy:

- **If price reaches a predicted local maximum → Sell 100% of holdings**
- **If price reaches halfway between min/max → Sell 30%**

This approach **realizes profits while maintaining exposure to further price increases**

```python
if now.strftime('%H:%M') == trade_times[sell_time].strftime('%H:%M') and
holdings[symbol] > 0:
    place_order(symbol, qty=holdings[symbol], side="sell")  # Full exit

elif now.strftime('%H:%M') == trade_times[half_sell_time].strftime('%H:%M'):
    qty_to_sell = int(holdings[symbol] * 0.3)
    place_order(symbol, qty=qty_to_sell, side="sell")  # Partial exit
```

3. Risk Management for Safe Trading

The system continuously **monitors live prices after a trade is executed** to implement stop-loss and take-profit rules:

- **Stop-Loss:** Sell all holdings if price drops **2% below purchase price** for more than 2 minutes.
- **Take-Profit:** Sell **30% at 5% gain**, then let the rest ride.
- **Trailing Stop:** If price reaches **3% above buy price**, a **dynamic 1% trailing stop activates** to lock in profits.

```python
if current_price <= stop_loss_price and time_since_trade > 120:
    return "sell_all"

elif current_price >= take_profit_price:
    return "sell_partial"

elif trailing_stop_price and current_price <= trailing_stop_price:
    return "sell_all"
```

Stock Trading Execution: Automating the Trading Workflow

Stock trading execution is fully automated through scheduled tasks, ensuring that data retrieval, predictions, and trade execution happen at precise times without manual intervention. The workflow is structured into four key steps: **data storage, ETF trading, stock trading, and continuous execution** to maintain a smooth and systematic trading process.

At **market close (4:00 PM ET)**, the system saves all market data locally for future analysis and backtesting. This ensures that a complete dataset is available for improving trading strategies over time. By scheduling the data-saving function at this time, we preserve accurate and comprehensive historical records that can be leveraged for training new models and refining trade execution.

```python
schedule.every().day.at("16:00").do(fetch_and_save_data, symbols)
```

ETF trading execution follows a two-step process. The system places initial buy orders for ETFs at **market open (9:30 AM ET)**, ensuring that investments are allocated as early as possible based on predefined strategies. Beyond the initial trade, ETF trading is executed **every minute** throughout the trading day, allowing the system to monitor price movements and act based on pre-set price thresholds. This approach ensures that ETF positions are dynamically managed with frequent adjustments to maximize returns.

```python
schedule.every().day.at("09:30").do(execute_etf_trading)
schedule.every().minute.do(execute_etf_trading)
```

Stock trading execution is more complex, as it incorporates **LSTM model predictions** and **real-time decision-making**. Predictions are updated in half **of an hour**, ensuring that the latest data is used for trade decisions. The first stock purchases occur at **10:30 AM ET**, allowing technical indicators to stabilize after the first hour of trading. To ensure execution accuracy, trading is scheduled **every minute**, enabling the system to react dynamically to market fluctuations and adjust holdings accordingly. A slight delay (2 seconds) before executing the **first stock trade** ensures the latest predictions are available.

```python
schedule.every().day.at("10:30").do(lambda: time.sleep(2) or
execute_stock_trading()) #Purchase Stocks when we have first hour data & sleep
a little bitte to avoid the
schedule.every().hour.at(":00").do(predict_and_store)
schedule.every().hour.at(":30").do(predict_and_store) # Predict once per half
hour
schedule.every().minute.do(execute_stock_trading)  # Execute trades every
minute
```

To maintain continuous automation, the system runs on an **infinite loop**, executing scheduled tasks every **60 seconds**. This ensures that trading operations remain uninterrupted and that all trading rules, risk management strategies, and model predictions are executed in real-time without manual oversight. By integrating **structured scheduling, AI-driven decision-making, and real-time execution**, the strategy achieves a balance between automation and adaptability, ensuring both ETFs and stocks are traded efficiently based on market conditions.

```python
while True:
    schedule.run_pending()
    time.sleep(60)
```

Testing and Optimization

1. Backtesting Process

To evaluate the effectiveness of the trading strategy, I conducted **backtesting** using historical market data. The backtesting process involved the following steps:

- **Data Collection:**
  - Retrieved historical price data for **SPY, VIS, NVDA, SBUX, and C** using the `StockHistoricalDataClient`.
  - Used **minute-level** data for accurate simulation of real-time trading conditions.
- **Implementation of Trading Rules:**
  - Applied predefined trading rules to historical data to simulate buy/sell signals.
  - Ensured that **transaction costs, slippage, and market conditions** were considered.
- **Performance Metrics:**
  - Evaluated performance using key financial indicators, such as:
    - Annualized Return
    - Sharpe Ratio
    - Maximum Drawdown
    - Win/Loss Ratio
    - Profit Factor
  - Compared results against a **benchmark index (e.g., S&P 500)**.

2. Optimization Steps

Based on backtesting results, the strategy was optimized to improve performance. The following optimizations were performed:

- **Parameter Tuning:**
  - Adjusted key hyperparameters, such as:
    - Moving average periods for trend-following strategies.
    - Stop-loss and take-profit levels to manage risk.
    - Entry and exit conditions based on volatility and momentum indicators.
- **Strategy Refinement:**
  - Analyzed **false signals** and adjusted filters to reduce unnecessary trades.
  - Improved **position sizing** to optimize risk-reward ratio.
- **Walk-Forward Testing:**
  - Divided historical data into **training and testing periods**.
  - Re-optimized parameters periodically to adapt to market conditions.

3. Adjustments Based on Testing Results

After analyzing the backtesting outcomes, the following modifications were made:

- **Reduced Overfitting:**

    - Avoided excessive parameter tuning to ensure robustness in live trading.
- **Improved Execution Timing:**
    - Adjusted trade execution rules to account for market volatility and liquidity.
- **Risk Management Enhancements:**
    Added additional safeguards, such as dynamic stop-loss adjustments.

Automation and Scheduling

1. Automated Data Retrieval Process

- The script fetches stock market data using the `StockBarsRequest` API.
- It retrieves minute-level stock data for the last 7 days for each symbol listed in `symbols`.
- The fetched data is stored in both **Pickle (`.pkl`)** and **CSV (`.csv`)** formats, ensuring efficient storage and compatibility with other tools.

2. Task Scheduling

- The `schedule` library is used to automate the execution of `fetch_and_save_data()`.
- The function runs **every minute** to ensure timely updates.
- A `while True` loop keeps the script running indefinitely, checking and executing scheduled tasks every 60 seconds.

3. Error Handling

- The script includes a `try-except` block to catch any errors that occur during data retrieval and file writing:

```
except Exception as e:
    logging.error(f"Error updating {symbol}: {e}")
    logging.error(traceback.format_exc())
```

- If an error occurs (e.g., API failure or network issues), it logs the error instead of stopping the script.

4. Logging Mechanism

- The script logs important events using Python's `logging` module:

```
# Configure Logging
logging.basicConfig(
    filename="market_data.log",
    level=logging.INFO,
    format="%(asctime)s - %(levelname)s - %(message)s"
)
```

- **INFO Logs** track successful updates of CSV and Pickle files.
- **ERROR Logs** capture failures with detailed stack traces.

5. Script Version Control

To manage changes and maintain version history, the script can be version-controlled using Git.

Paper Trading and Monitoring

Explain how you utilized Alpaca's paper trading feature to simulate live market conditions. Discuss how you monitored your algorithm's performance in a risk-free environment.

We used Alpaca’s paper trading feature to test our trading algorithm in a realistic market environment without risking actual money. This allowed us to see how our strategy would perform under live conditions, including real-time price movements and order execution. For example, when our model predicted a local low value, we executed a simulated buy order and tracked how the price moved afterward. If the price continued to drop, we analyzed whether our dollar-cost averaging strategy improved returns or if it resulted in unnecessary losses. We kept a close eye on key performance metrics like profit and loss, and we made adjustments along the way to improve our approach. Since the market is always changing, we also monitored how well our algorithm adapted to different levels of volatility and price trends. We found that our model performs not so well during market open and close. To solve this problem, we developed separate trading strategies for these high-volatility periods.

Results and Lessons Learned

During the development of our project, we encountered several challenges such as feature selection, model accuracy over time, training efficiency, and handling market volatility. Through this process, we gained valuable lessons about model selection, real-time trading constraints, and the importance of adapting to market conditions.

One of the main challenges we faced was feature selection. Because of the complexity of our dataset, we tested 9 different models with different features. Each of them shows different accuracy. Some suffered from overfitting, while others underperformed due to lots of bias. After multiple iterations, we finally selected the most optimal model. One key lesson learned was that feature selection is just as important as model choice—even a powerful model can underperform if the right features are not chosen.

Another significant challenge was that while our model performed well in the first half of the predictions, its accuracy declined in the latter half. This suggests that the model struggles with long-term dependencies or evolving market conditions. Through this experience, we learned that financial markets are highly dynamic, and a static model may not generalize well over time. We tried with predictions of different periods of time, but the same situation happened. Therefore we decided to only take consideration of the first half of the predictions. Moreover, our model performed not very well during market opening and closing hours, because market volatility was at its peak. To mitigate this issue, we decided not to use the model for predictions during these periods, but this approach led to missed trading opportunities. In the future, instead of using the same model throughout the entire trading day, we can train a separate model specifically for high-volatility periods. This model should be able to focus on shorter time frames to capture rapid price movements.

A final limitation we encountered was the long training time, which prevented us from continuously updating the model in real-time. We realized that real-time trading requires a trade-off between model complexity and execution speed. Even though complex models may lead to better accuracy, they are often impractical for real-time applications. In the future, to overcome this, we plan to optimize training by leveraging GPU/TPU acceleration for these complex models.

Compliance and Legal Considerations

When designing and implementing our algorithmic trading system, we place a strong emphasis on adhering to applicable legal requirements and industry best practices to ensure both compliance and safety. Below are the key legal and regulatory aspects we consider:

- **Paper Trading Environment:**
  We use Alpaca's paper trading environment to simulate real-world trading. This allows us to test our strategies without risking actual funds, ensuring that our operations remain within regulatory limits during development.

- **Regulatory Alignment:**
  Our system is built in accordance with guidelines from regulatory bodies like the SEC and FINRA. This means we are attentive to industry best practices and legal requirements to help prevent any non-compliant activities.

- **Risk Management:**
  We incorporate clear risk management rules, such as stop-loss and profit-taking mechanisms, which help control the risks inherent in trading. These measures also reflect the regulatory emphasis on protecting investors and ensuring market stability.

- **Data Privacy and Security:**
  Market data is stored securely in both Pickle and CSV formats with proper timestamping and timezone handling (using ZoneInfo for Eastern Time). This approach not only maintains data integrity but also supports compliance with data protection regulations.

- **Transparency and Documentation:**
  The system is thoroughly documented and version-controlled. This practice ensures transparency in our processes and provides an audit trail, which is important for both internal reviews and external regulatory checks.

Overall, our approach aims to balance innovation in algorithmic trading with strict adherence to legal and regulatory standards, ensuring a safe and compliant trading environment.

Conclusion

In conclusion, our project successfully developed a robust algorithmic trading system that integrates comprehensive market data retrieval, advanced trading strategies, and stringent risk management protocols. Utilizing Alpaca's paper trading environment allowed us to simulate real-world trading without financial exposure, while our dual-format data storage strategy ensured both efficiency and clarity in analysis. Our approach balanced strategic capital allocation between ETFs and individual stocks, leveraging technical indicators and deep learning models to identify optimal entry and exit points. Through extensive backtesting and iterative optimization, we refined our models to adapt to dynamic market conditions, despite challenges such as feature selection and high volatility periods. Furthermore, the incorporation of strong compliance measures, detailed documentation, and automated scheduling enhanced

the system’s transparency and reliability, ultimately positioning our trading strategy for potential future success in live markets.