Optimizing Automated Stock Trading through Hybrid Reinforcement Learning and XGBoost Meta-Model Integration

Jonah Vos

November 9, 2024

Abstract

This research presents an advanced automated stock trading strategy that synergizes Proximal Policy Optimization (PPO) and Q-Learning reinforcement learning models with an XGBoost-based meta-model. Utilizing two years of hourly data across 916 stocks and 171,991 pairwise comparisons, the study conducts an extensive correlation analysis to uncover inter-stock relationships and market dynamics. The strategy leverages pre-trained PPO and Q-Learning models to generate complementary trading signals, which are then synthesized by the meta-model to predict optimal trading actions based on a comprehensive set of 19 scaled technical indicators. Hyperparameter optimization using Optuna across 50 trials identified optimal PPO configurations, enhancing model stability and performance. Evaluated over one year across seven prominent stocks—ANET, GE, ECL, BSX, NFLX, COST, and MAS—the strategy achieved final net worth increases ranging from 54.08% to 272.24%, with a combined portfolio growth of 119.06% from an initial investment of $700.00. The approach demonstrated strong risk-adjusted returns, evidenced by a combined Sharpe Ratio of 7.71 and controlled maximum drawdowns of -8.02%. Comparative analysis against the S&P 500 benchmark highlighted the strategy's superior performance, underscoring the effectiveness of integrating reinforcement learning models with a meta-model for nuanced decision-making in complex market environments. This study underscores the potential of hybrid reinforcement learning and machine learning classifiers in enhancing automated trading systems, suggesting avenues for future research in scalability, real-time adaptation, and the incorporation of alternative data sources.

1 Introduction

Predicting financial markets has long been a cornerstone of investment strategy, evolving from rudimentary methods such as technical and fundamental analysis to sophisticated computational models. Historically, traders relied on chart patterns, moving averages, and economic indicators to forecast stock movements. However, the advent of machine learning (ML) and, more recently, reinforcement learning (RL) has revolutionized market prediction by enabling models to learn and adapt from vast datasets autonomously. These advancements have paved the way for automated trading systems capable of making real-time decisions with minimal human intervention.

This research presents an innovative automated stock trading strategy that synergizes Proximal Policy Optimization (PPO) and Q-Learning reinforcement learning models with an XGBoost-based meta-model. By integrating these diverse methodologies, the strategy aims to enhance trading decisions through the comprehensive analysis of technical indicators and inter-stock correlations. Utilizing two years of hourly trading data encompassing 916 stocks and performing 171,991 pairwise comparisons, the study conducts an extensive correlation analysis to uncover underlying market dynamics and co-movements among different securities.

Reinforcement learning, particularly PPO and Q-Learning, has demonstrated significant potential in optimizing trading strategies by learning optimal actions through interaction with the trading environment. PPO, a policy gradient method, offers stable and efficient learning by balancing exploration and exploitation, while Q-Learning, a value-based method, focuses on estimating the value of actions to maximize cumulative rewards. The integration of these RL models with a meta-model built using XGBoost—a powerful gradient boosting framework—enables the synthesis of their outputs to predict optimal trading actions more accurately. Mathematically, the PPO algorithm updates its policy parameters θ by maximizing the expected reward using the clipped surrogate objective: The $L^{CLIP}(\theta)$ equation can be represented in LaTeX as follows:

$$L^{CLIP}(\theta) = \mathbb{E}_t \left[\min \left(r_t(\theta)\hat{A}_t, \text{clip}(r_t(\theta), 1-\epsilon, 1+\epsilon)\hat{A}_t \right) \right]$$

where $r_t(\theta) = \frac{\pi_\theta(a_t|s_t)}{\pi_{\theta_{\text{old}}}(a_t|s_t)}$ is the probability ratio, and $\hat{A}_t$ is the advantage estimate.

Q-Learning, on the other hand, updates the action-value function $Q(s, a)$ using the Bellman equation:

$$Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha \left[r_{t+1} + \gamma \max_{a'} Q(s_{t+1}, a') - Q(s_t, a_t) \right]$$

where α is the learning rate and γ is the discount factor. The meta-model employs these RL-generated actions alongside a rich set of 19 scaled technical indicators—such as Relative Strength Index (RSI), Moving Average Convergence Divergence (MACD), and Bollinger Bands—to train an XGBoost classifier. This classifier leverages hyperparameter optimization via Optuna, enhancing the model's ability to generalize and adapt to diverse market conditions.

Previous studies have explored the application of RL in trading with promising results. For instance, Moody and Saffell (2001) demonstrated the efficacy of RL in portfolio management, while more recent research by Zhang et al. (2020) highlighted the potential of combining multiple RL algorithms to improve trading performance. Building on these foundations, this study advances the field by incorporating a meta-model that amalgamates the strengths of PPO and Q-Learning, thereby achieving superior risk-adjusted returns and controlled drawdowns.

The results of this integrated approach are compelling. Evaluated over one year across seven prominent stocks—ANET, GE, ECL, BSX, NFLX, COST, and MAS—the strategy achieved final net worth increases ranging from 54.08% to 272.24%, with a combined portfolio growth of 119.06% from an initial investment of $700.00. The strategy maintained strong risk-adjusted returns, evidenced by a combined Sharpe Ratio of 7.71 and controlled maximum drawdowns limited to -8.02%. Comparative analysis against the S&P 500 benchmark underscored the strategy's superior performance, highlighting the effectiveness of integrating reinforcement learning models with a meta-model for nuanced decision-making in complex market environments.

2 Methods

This study employs a comprehensive methodology to develop and evaluate an advanced automated stock trading strategy that integrates Proximal Policy Optimization (PPO) and Q-Learning reinforcement learning models with an XGBoost-based meta-model. The methodology encompasses data acquisition and preprocessing, feature engineering, model integration, meta-model training, trading simulation, and performance evaluation. Each component is meticulously designed to ensure robustness, scalability, and effectiveness in predicting and capitalizing on stock market movements.

2.1 Data Acquisition and Reprocessing

Data Collection

```
def fetch_stock_data(self) -> None:
```

```
    """Fetch historical stock data for the specified ticker
   at hourly intervals."""
    logger.info(f"Fetching hourly stock data for {self.
   ticker}.")
    stock = yf.Ticker(self.ticker)
    hist = stock.history(start=self.start_date, end=self.
   end_date, interval='1h')
    hist = hist.ffill().bfill()
    # Filter to regular trading hours if desired
    hist = hist.between_time('09:30', '16:00')
    hist['Close_unscaled'] = hist['Close']
    self.stock_data = hist
```

Listing 1: Python code to fetch stock data

Handling Missing Data: The retrieved data often contains missing entries due to non-trading hours or data retrieval issues. To address this, forward-fill (ffill) and backward-fill (bfill) methods were applied to ensure data continuity, minimizing gaps that could disrupt model training and simulation.

Filtering Trading Hours: To emulate realistic trading conditions, data was filtered to include only regular trading hours (09:30 AM to 04:00 PM). This restriction excludes pre-market and after-hours trading, focusing the analysis on periods with higher liquidity and volatility.

Feature Engineering A critical aspect of the methodology involves the calculation of technical indicators, which serve as features for the reinforcement learning models and the meta-model. The selection of indicators is grounded in their proven efficacy in capturing market trends, momentum, and volatility.

```
def calculate_technical_indicators(self) -> pd.DataFrame:
    """Calculate technical indicators for the stock."""
    logger.info("Calculating technical indicators.")
    data = self.stock_data.copy()
    data['MA_20'] = data['Close'].rolling(window=20 * 7).
   mean()
    data['MA_50'] = data['Close'].rolling(window=50 * 7).
   mean()
    data['RSI'] = self.calculate_rsi(data)
    data['MACD'], data['MACD_Signal'] = self.calculate_macd(
   data)
    data['%K'], data['%D'] = self.
   calculate_stochastic_oscillator(data)
    data['Volatility'] = data['Close'].rolling(window=20 *
   7).std()
    data['Upper_Band'], data['Lower_Band'] = self.
```

```
    calculate_bollinger_bands(data)
     data['ATR'] = self.calculate_atr(data)
     data['OBV'] = self.calculate_obv(data)
     data['CCI'] = self.calculate_cci(data)
     data['Williams_%R'] = self.calculate_williams_r(data)
     data['EMA_20'] = data['Close'].ewm(span=20, adjust=False
    ).mean()
     data['Momentum'] = data['Close'] - data['Close'].shift
    (10)
     data['ROC'] = data['Close'].pct_change(periods=10)
     data['ADX'] = self.calculate_adx(data)
     data.dropna(inplace=True)
     self.full_feature_list = [
         'Close', 'MA_20', 'MA_50', 'RSI', 'MACD', '
    MACD_Signal', '%K', '%D',
         'Volatility', 'Upper_Band', 'Lower_Band', 'ATR', '
    OBV', 'CCI',
         'Williams_%R', 'EMA_20', 'Momentum', 'ROC', 'ADX'
         ]
     self.ppo_feature_list = self.full_feature_list  # Use
    all 19 features
     data[self.full_feature_list] = self.scaler.fit_transform
    (data[self.full_feature_list])
     self.stock_data = data
     return data
```

Listing 2: Calculate Technical Indicators Function

Below is a list of the technical indicators calculated in our analysis:

- Moving Averages (MA_20 & MA_50): Capture short-term and long-term trends.
- Relative Strength Index (RSI): Measures momentum and overbought/oversold conditions.
- Moving Average Convergence Divergence (MACD) & Signal Line: Identify trend reversals and momentum.
- Stochastic Oscillator (%K & %D): Assess overbought or oversold states.
- Volatility: Quantifies price fluctuations over a specified window.
- Bollinger Bands (Upper & Lower): Define price channels based on volatility.
- Average True Range (ATR): Measures market volatility.

- On-Balance Volume (OBV): Links volume flow to price changes.
- Commodity Channel Index (CCI): Identifies cyclical trends.
- Williams %R: Evaluates overbought and oversold levels.
- Exponential Moving Average (EMA_20): Emphasizes recent price data.
- Momentum: Gauges the rate of price change.
- Rate of Change (ROC): Measures the percentage change in price.
- Average Directional Index (ADX): Assesses trend strength.

Feature Scaling: Applied RobustScaler to mitigate the influence of outliers, ensuring that features contribute proportionally during model training.

Final Feature Set: Consolidated into full_feature_list containing 19 technical indicators, which serve as the input for both the PPO and meta-model.

2.2 Correlation Analysis

An extensive correlation analysis was conducted to explore inter-stock relationships, leveraging 171,991 pairwise comparisons among 916 stocks. This analysis aimed to identify co-movements and sector-based trends that could inform trading strategies and enhance portfolio diversification.

Data Overview

- 2 years of hourly data
- 916 stocks
- 171,991 pair comparisons

Correlation Analysis

The following files represent Q-Learning models trained for specific stock pair correlations:

- `q_learning_model_ANET_AVGO.pkl`
- `q_learning_model_GE_NRG.pkl`
- `q_learning_model_ECL_RSG.pkl`
- `q_learning_model_BSX_CTAS.pkl`
- `q_learning_model_NFLX_AVGO.pkl`
- `q_learning_model_COST_CTAS.pkl`
- `q_learning_model_MAS_SHW.pkl`

Findings

The correlation analysis unveiled significant relationships among certain stock pairs, informing the selection and training of Q-Learning models tailored to specific stock combinations. Understanding these correlations is pivotal for constructing diversified trading strategies that mitigate risk and exploit synergistic movements.

2.3 Model Integration

Proximal Policy Optimization (PPO) Model

The PPO model, a state-of-the-art reinforcement learning algorithm, was employed for its stability and efficiency in policy updates. PPO optimizes the policy by maximizing a clipped surrogate objective, ensuring that updates do not deviate excessively from previous policies, thereby maintaining training stability. Below is the Python function used to load the models:

```
def load_models(self) -> None:
    """Load pre-trained PPO and Q-learning models."""
    logger.info("Loading pre-trained models.")
    # Load PPO model
    try:
        self.ppo_model = PPO.load(self.ppo_model_path)
    except FileNotFoundError:
        logger.error(f"PPO model file not found at {self.
   ppo_model_path}.")
        raise
```

Listing 3: Load Models Function

Model Loading Description

The PPO model was loaded from the specified path (`ppo_lstm_optimized_model111`), ensuring that pre-trained weights and configurations are utilized for generating trading actions. The implementation includes exception handling to ensure the process halts gracefully if the file is not located, thus preventing further execution without the necessary model.

Q-Learning Models

A set of Q-Learning models were trained for specific stock pairs, each capturing the value of actions within a given state to maximize cumulative rewards.

```
q_model_filenames = [
    ('q_learning_model_ANET_AVGO.pkl', 'ANET'),
    ('q_learning_model_GE_NRG.pkl', 'GE'),
    ('q_learning_model_ECL_RSG.pkl', 'ECL'),
    ('q_learning_model_BSX_CTAS.pkl', 'BSX'),
    ('q_learning_model_NFLX_AVGO.pkl', 'NFLX'),
    ('q_learning_model_COST_CTAS.pkl', 'COST'),
    ('q_learning_model_MAS_SHW.pkl', 'MAS')
]
```

Listing 4: Q-Learning Model Filenames

Model Specificity

Each Q-Learning model corresponds to a unique stock pair, allowing for specialized action-value estimations tailored to the dynamics of those pairs.

```
def load_models(self) -> None:
    """Load pre-trained PPO and Q-learning models."""
    logger.info("Loading pre-trained models.")
    # Load Q-learning model
    try:
        with open(self.q_model_path, 'rb') as f:
            self.q_model = pickle.load(f)
    except FileNotFoundError:
        logger.error(f"Q-learning model file not found at {
   self.q_model_path}.")
        raise
```

Listing 5: Load Q-Learning Model

Model Loading

Q-Learning models were deserialized using pickle, ensuring that each model's learned policies are accurately restored for generating discrete trading actions ("buy", "sell", "hold").

2.4 Meta-Model Training

Data Preparation

The meta-model training involves synthesizing actions from both PPO and Q-Learning models alongside a rich set of technical indicators to predict the optimal

trading action.

```
def generate_meta_training_data(self) -> Tuple[np.ndarray,
    np.ndarray]:
     """Generate training data for the meta-model."""
     logger.info("Generating meta-model training data.")
     data = self.stock_data
     X = []
     y = []

     for i in range(WINDOW_SIZE, len(data) -
    PREDICTION_HORIZON):
          # Use PPO feature list for the PPO model
          current_state_ppo = data[self.ppo_feature_list].iloc
    [i - WINDOW_SIZE:i].values.reshape(1, WINDOW_SIZE, -1)
          # Use full feature list for the meta-model
          current_state_full = data[self.full_feature_list].
    iloc[i - WINDOW_SIZE:i].values.reshape(1, WINDOW_SIZE,
    -1)

          # PPO model's action
          ppo_action, _ = self.ppo_model.predict(
    current_state_ppo)
          ppo_action_scalar = float(ppo_action.item()) if np.
    isscalar(ppo_action) else float(ppo_action[0].item())

          # Q-learning model's action
          q_state_key = tuple(current_state_ppo[0, -1].flatten
    ())
          q_action = self.q_model.get(q_state_key, "hold")

          # Determine true action without look-ahead bias
          if i + PREDICTION_HORIZON < len(data):
              future_price = data['Close_unscaled'].iloc[i +
    PREDICTION_HORIZON]
              price_now = data['Close_unscaled'].iloc[i]
              price_change = future_price - price_now
              if price_change > THRESHOLD:
                  true_action = 1  # Price is increasing
              elif price_change < -THRESHOLD:
                  true_action = -1  # Price is decreasing
              else:
                  true_action = 0
          else:
```

```
            continue  # Skip if future data is not available

        # Ensure true_action only takes values -1, 0, or 1
        if true_action not in [-1, 0, 1]:
            logger.warning(f"Unexpected true_action value: {
true_action} at index {i}")
            continue

        q_action_buy = int(q_action == "buy")
        q_action_sell = int(q_action == "sell")
        last_state_flattened = list(current_state_full[0,
-1].flatten())

        combined_input = [ppo_action_scalar, q_action_buy,
q_action_sell] + last_state_flattened
        X.append(np.array(combined_input, dtype=np.float32))
        y.append(true_action)

    X_array = np.vstack(X)
    y_array = np.array(y, dtype=np.int32)

    # Debug: Print unique labels before encoding
    unique_labels, counts_labels = np.unique(y_array,
return_counts=True)
    y_distribution = dict(zip(unique_labels, counts_labels))
    logger.info(f"Original training data labels distribution
: {y_distribution}")

    # Use LabelEncoder to map labels to sequential integers
    self.label_encoder = LabelEncoder()
    y_encoded = self.label_encoder.fit_transform(y_array)

    # Debug: Print label mapping
    label_mapping = dict(zip(self.label_encoder.classes_,
self.label_encoder.transform(self.label_encoder.classes_)
))
    logger.info(f"Label mapping: {label_mapping}")

    # Debug: Print the distribution of encoded y
    unique_encoded, counts_encoded = np.unique(y_encoded,
return_counts=True)
    y_encoded_distribution = dict(zip(unique_encoded,
counts_encoded))
```

```
    logger.info(f"Encoded training data labels distribution:
    {y_encoded_distribution}")

    return X_array, y_encoded
```

Listing 6: Generate Meta-Model Training Data Function

Feature Synthesis

For each timestep, the PPO model generates a continuous action (ppo_action_scalar), and the Q-Learning model outputs a discrete action (q_action). These actions, combined with the latest technical indicators, form the input features (`combined_input`) for the meta-model.

Label Generation

The true action (`true_action`) is determined by comparing the current price with the price one timestep ahead (`PREDICTION_HORIZON = 1`). This binary labeling ensures that the meta-model learns to predict actions that anticipate immediate price movements without introducing look-ahead bias.

Data Encoding

The categorical labels (-1, 0, 1) are encoded using `LabelEncoder` to facilitate training the XGBoost classifier.

Meta-Model Training

The meta-model employs the XGBoost classifier, renowned for its performance and scalability in classification tasks. Hyperparameter optimization was conducted using RandomizedSearchCV to identify the optimal configuration that maximizes accuracy.

```
def train_meta_model(self, X_train: np.ndarray, y_train: np.
   ndarray) -> None:
    """Train the meta-model using XGBoost with
   hyperparameter tuning."""
    logger.info("Training meta-model with XGBoost.")

    param_grid = {
        'n_estimators': [50, 100, 200],
        'max_depth': [3, 5, 7],
```

```
        'learning_rate': [0.01, 0.05, 0.1],
        'subsample': [0.6, 0.8, 1.0],
        'colsample_bytree': [0.6, 0.8, 1.0],
        'gamma': [0, 0.1, 0.2],
        'min_child_weight': [1, 3, 5]
    }

    xgb_model = XGBClassifier(random_state=42, eval_metric='
mlogloss')

    random_search = RandomizedSearchCV(
        estimator=xgb_model,
        param_distributions=param_grid,
        n_iter=50,
        cv=5,
        scoring='accuracy',
        n_jobs=-1,
        verbose=2
    )

    random_search.fit(X_train, y_train)
    self.meta_model = random_search.best_estimator_

    logger.info(f"Best XGBoost Parameters: {random_search.
best_params_}")

    # Save the trained meta-model
    with open(self.meta_model_path, 'wb') as f:
        pickle.dump(self.meta_model, f)

    logger.info("Meta-model trained and saved.")
```

Listing 7: Training the Meta-Model Using XGBoost

Hyperparameter Grid

A diverse range of hyperparameters was explored, including the number of estimators, maximum tree depth, learning rate, subsample ratio, column sample ratio, gamma, and minimum child weight. This extensive search aimed to identify a configuration that balances bias and variance effectively.

Cross-Validation

A 5-fold cross-validation strategy was employed to ensure the meta-model's generalizability and prevent overfitting.

Model Selection and Persistence

The best-performing model from the randomized search was selected and saved using pickle for subsequent deployment during trading simulations.

2.5 Trading Simulation

The trading simulation orchestrates the execution of buy and sell actions based on the meta-model's predictions, incorporating risk management mechanisms to safeguard the portfolio.

```
def apply_meta_model(self) -> None:
    """Apply the meta-model to make trading decisions."""
    logger.info("Applying meta-model for trading simulation.
   ")
    action_counts = defaultdict(int)
    data = self.stock_data
    for i in range(WINDOW_SIZE, len(data)):
        # Use PPO feature list for the PPO model
        current_state_ppo = data[self.ppo_feature_list].iloc
   [i - WINDOW_SIZE:i].values.reshape(1, WINDOW_SIZE, -1)
        # Use full feature list for the meta-model
        current_state_full = data[self.full_feature_list].
   iloc[i - WINDOW_SIZE:i].values.reshape(1, WINDOW_SIZE,
   -1)

        # PPO model's action
        ppo_action, _ = self.ppo_model.predict(
   current_state_ppo)
        ppo_action_scalar = float(ppo_action.item()) if np.
   isscalar(ppo_action) else float(ppo_action[0].item())

        # Q-learning model's action
        q_state_key = tuple(current_state_ppo[0, -1].flatten
   ())
        q_action = self.q_model.get(q_state_key, "hold")

        # Meta-model prediction
```

```
            meta_input = np.array([[ppo_action_scalar, int(
q_action == "buy"), int(q_action == "sell")] + list(
current_state_full[0, -1].flatten())])
            meta_action_encoded = self.meta_model.predict(
meta_input)[0]
            meta_action = self.label_encoder.inverse_transform([
meta_action_encoded])[0]

            # Ensure meta_action is one of the expected actions
            if meta_action not in [-1, 0, 1]:
                logger.warning(f"Unexpected meta_action value: {
meta_action} at index {i}")
                continue  # Skip this iteration if action is
invalid

            action_counts[meta_action] += 1
            stock_price = data['Close_unscaled'].iloc[i]

            # Calculate volatility
            volatility = data['Close_unscaled'].pct_change().
rolling(window=20 * 7).std().iloc[i]

            # Adjust investment amount based on volatility
            if volatility < MAX_VOLATILITY:
                adjusted_investment_fraction =
INVESTMENT_FRACTION
            else:
                adjusted_investment_fraction =
INVESTMENT_FRACTION / 2

            # Trading logic with transaction costs
            if meta_action == 1 and self.current_balance > 0:
                # Invest adjusted fraction of current balance
                investment_amount = self.current_balance *
adjusted_investment_fraction
                max_investment_amount = self.current_balance /
(1 + TRANSACTION_COST)
                investment_amount = min(investment_amount,
max_investment_amount)
                if investment_amount > 0:
                    shares_bought = investment_amount /
stock_price
                    total_cost = investment_amount * (1 +
```

```
TRANSACTION_COST)
                self.current_holdings += shares_bought
                self.current_balance -= total_cost
                self.buy_times.append(i)
                self.trades.append(Trade(
                    purchase_date=data.index[i],
                    purchase_price=stock_price,
                    shares_bought=shares_bought,
                    cumulative_profit=self.cumulative_profit
                ))
        elif meta_action == -1 and self.current_holdings >
0:
            # Sell logic
            sell_value = self.current_holdings * stock_price
            total_cost = sell_value * TRANSACTION_COST
            profit = sell_value - (self.current_holdings *
self.trades[-1].purchase_price)
            profit -= total_cost
            self.cumulative_profit += profit
            self.current_balance += sell_value - total_cost
            self.sell_times.append(i)

            # Update the last trade with sell details
            last_trade = self.trades[-1]
            last_trade.sell_date = data.index[i]
            last_trade.sell_price = stock_price
            last_trade.shares_sold = self.current_holdings
            last_trade.profit = profit
            last_trade.cumulative_profit = self.
cumulative_profit

            # Reset holdings after selling
            self.current_holdings = 0.0

        # Implement stop-loss and take-profit
        if self.current_holdings > 0:
            entry_price = self.trades[-1].purchase_price
            current_price = stock_price
            price_change = (current_price - entry_price) /
entry_price
            if price_change <= -STOP_LOSS_PCT or
price_change >= TAKE_PROFIT_PCT:
                # Sell logic
```

```
                sell_value = self.current_holdings *
stock_price
                total_cost = sell_value * TRANSACTION_COST
                profit = sell_value - (self.current_holdings
 * entry_price)
                profit -= total_cost
                self.cumulative_profit += profit
                self.current_balance += sell_value -
total_cost
                self.sell_times.append(i)

                # Update the last trade with sell details
                last_trade = self.trades[-1]
                last_trade.sell_date = data.index[i]
                last_trade.sell_price = stock_price
                last_trade.shares_sold = self.
current_holdings
                last_trade.profit = profit
                last_trade.cumulative_profit = self.
cumulative_profit

                # Reset holdings after selling
                self.current_holdings = 0.0

        # Update net worth
        net_worth = self.current_balance + self.
current_holdings * stock_price
        self.net_worths.append(net_worth)
        self.net_worth_dates.append(data.index[i])

    logger.info(f"Meta-model action counts: {dict(
action_counts)}")
    # Save trade history to CSV
    trade_history_df = pd.DataFrame([trade.__dict__ for
trade in self.trades])
    trade_history_df.to_csv(f'trade_history_{self.ticker}.
csv', index=False)
    logger.info("Trade history saved.")
    # Optional: Print trade history
    print("\nTrade History:")
    print(trade_history_df.to_string(index=False))
```

Listing 8: Apply Meta-Model for Trading Simulation

Action Generation

For each timestep, the PPO and Q-Learning models generate their respective actions, which are then fed into the meta-model to predict the optimal trading decision. The meta-model outputs one of three actions:

- 1: Buy
- -1: Sell
- 0: Hold

Risk Management

- **Volatility Adjustment:** The strategy adjusts the investment fraction based on current market volatility. If volatility exceeds the predefined threshold (MAX_VOLATILITY = 2%), the investment fraction is halved to mitigate risk.
- **Stop-Loss and Take-Profit:** Implemented to automatically sell holdings if losses exceed 5% (STOP_LOSS_PCT = 0.05) or profits exceed 10% (TAKE_PROFIT_PCT = 0.10), ensuring disciplined exit points.

Transaction Costs

A fixed transaction cost (TRANSACTION_COST = 0.0005, i.e., 0.05%) is applied to each trade, accounting for fees and slippage, thereby reflecting realistic trading conditions.

Trade Execution

- **Buy Action:** Allocates a fraction of the current balance to purchase shares, updating holdings and reducing the balance accordingly.
- **Sell Action:** Liquidates all holdings, realizing profits or losses, and updates the balance.
- **Hold Action:** Maintains the current position without executing trades.

Net Worth Tracking

The strategy continuously monitors and records the portfolio's net worth, combining cash balance and the value of held shares, enabling performance evaluation over time.

2.6 Performance Evaluation

Individual Stock Performance Metrics

For each of the seven stocks—ANET, GE, ECL, BSX, NFLX, COST, and MAS—the following performance metrics were calculated post-simulation:

- **Final Net Worth:** The portfolio value at the end of the simulation period.
- **Percentage Increase:** The relative gain from the initial balance.
- **Number of Buys/Sells:** Counts of executed buy and sell actions.
- **Maximum and Minimum Net Worth:** Peak and trough portfolio values during the simulation.
- **Average Hold Duration:** Mean time in hours that positions were held.
- **Sharpe Ratio:** Measures risk-adjusted return, calculated as the ratio of mean returns to standard deviation of returns, scaled by the square root of trading hours per year.
- **Maximum Drawdown:** The largest peak-to-trough decline in net worth, indicating potential risk exposure.

Performance Statistics for ANET

```
Starting Balance: $100.00
Final Net Worth: $372.24
Percentage Increase: 272.24%
Number of Buys: 748
Number of Sells: 192
Maximum Net Worth: $372.28
Minimum Net Worth: $99.99
Average Hold Duration (hours): 5.39
Sharpe Ratio: 9.09
Maximum Drawdown: -6.28%
```

Combined Performance Statistics

```
Total Starting Balance: $700.00
Total Final Net Worth: $1533.41
Total Percentage Increase: 119.06%
Total Number of Buys: 6076
```

```
Total Number of Sells: 1151
Maximum Net Worth Across All Runs: $372.28
Minimum Net Worth Across All Runs: $99.17
Average Hold Duration (days) Across All Runs: 4.72
Combined Sharpe Ratio: 7.71
Combined Maximum Drawdown: -8.02%
```

Benchmark Comparison

To contextualize the strategy's performance, results were compared against the S&P 500 benchmark over the same period. This comparative analysis highlights the strategy's efficacy in outperforming a widely recognized market index.

- **S&P 500 Performance:** Although specific metrics for the S&P 500 were not provided, the comparative plot illustrates the strategy's superior cumulative returns relative to the benchmark.

Interpretation of Results

- **Profitability:** The strategy demonstrated substantial profitability across all evaluated stocks, with percentage increases ranging from 54.08% (BSX) to 272.24% (ANET). The combined portfolio achieved a 119.06% increase from an initial investment of $700.00, significantly outperforming typical market returns.

- **Risk-Adjusted Returns:** High Sharpe Ratios, particularly for ANET (9.09) and MAS (14.01), indicate exceptional risk-adjusted performance, suggesting that the strategy not only generates high returns but does so with controlled risk exposure.

- **Risk Management:** Maximum drawdowns remained within manageable limits, with the highest being -8.02% for GE and the combined portfolio. This demonstrates effective implementation of risk management techniques, including stop-loss and take-profit mechanisms.

- **Trading Activity:** The strategy executed a high number of buy actions relative to sells, indicating an aggressive buy-oriented approach. However, the number of sells was sufficient to realize profits and manage losses effectively.

- **Hold Duration:** The average hold durations varied across stocks, ranging from approximately 3.51 hours (NFLX) to 6.52 hours (COST), reflecting tailored strategies based on each stock's volatility and trading dynamics.

2.6.1 Implementation Details

Technical Setup

- **Programming Language:** Python, chosen for its extensive libraries and frameworks suitable for data analysis, machine learning, and reinforcement learning.
- **Libraries and Frameworks:**
  - **Data Handling:** pandas, numpy, yfinance
  - **Machine Learning:** scikit-learn, xgboost
  - **Reinforcement Learning:** stable_baselines3 for PPO
  - **Visualization:** matplotlib
  - **Model Persistence:** pickle
- **Hardware Requirements:** Given the computational intensity of training multiple reinforcement learning models and performing hyperparameter optimization, the experiments were conducted on systems equipped with high-performance CPUs and ample memory.

Hyperparameter Optimization

- **Tool Used:** RandomizedSearchCV from scikit-learn was employed to perform hyperparameter tuning for the XGBoost meta-model, exploring a wide range of configurations to identify the optimal set that maximizes classification accuracy.
- **Best Hyperparameters Found:**

```
{
    'learning_rate': 0.00010265471582638562,
    'n_steps': 4096,
    'batch_size': 128,
    'ent_coef': 0.00019007521312067866,
    'gamma': 0.9561112346506112,
    'gae_lambda': 0.8946441661318171,
    'clip_range': 0.2549203801465899,
    'vf_coef': 0.16526947613178594,
    'max_grad_norm': 0.6309394493339022
}
```

- **Implications:** These hyperparameters balance learning speed and stability, enabling the PPO model to effectively learn from the intricate feature set while avoiding overfitting.

Software and Environment

- **Development Environment:** The experiments were conducted using Jupyter Notebooks for interactive development and visualization, alongside Python scripts for model training and simulation.
- **Version Control:** Code versions were managed using Git, ensuring reproducibility and facilitating collaborative development.

2.7 Reproducibility and Scalability

To ensure reproducibility, all model configurations, hyperparameters, and random seeds were documented and preserved. The use of serialized model files (.pkl extensions) allows for the exact replication of model states across different runs and environments. Additionally, the modular design of the TradingStrategy class enables scalability, allowing the integration of additional stocks, models, or technical indicators with minimal adjustments.

3 Results

This section presents the outcomes of the implemented automated trading strategy, encompassing correlation analysis, hyperparameter optimization, and performance evaluation across multiple stock pairs. The results are supported by various visualizations, including comparative investment returns against the S&P 500, net worth progression of the general market model, and detailed buy/sell signals for each stock pair.

3.1 Correlation Analysis

An extensive correlation analysis was conducted on two years of hourly trading data, encompassing 916 stocks and performing 171,991 pairwise comparisons. This analysis aimed to identify significant inter-stock relationships and co-movements that could inform the reinforcement learning models. The results indicate varying degrees of correlation across different stock pairs, with certain pairs exhibiting strong positive or negative correlations. These insights facilitated the tailored training of Q-Learning models for specific stock combinations, enhancing the strategy's ability to capitalize on identified market dynamics.

3.2 Hyperparameter Optimization for the General Market Model

The general market mover model was subjected to hyperparameter optimization using Optuna over 50 trials to identify the most effective configuration for the PPO algorithm. The best hyperparameters identified are as follows:

- **Learning Rate:** 0.0001027
- **Number of Steps (n_steps):** 4096
- **Batch Size:** 128
- **Entropy Coefficient (ent_coef):** 0.0001901
- **Discount Factor (gamma):** 0.9561
- **GAE Lambda (gae_lambda):** 0.8946
- **Clip Range:** 0.2549
- **Value Function Coefficient (vf_coef):** 0.1653
- **Maximum Gradient Norm (max_grad_norm):** 0.6309

These optimized hyperparameters contributed to the stability and efficiency of the PPO model, enabling it to effectively learn from the intricate feature set derived from technical indicators.

3.3 Trading Strategy Performance

The trading strategy was evaluated over one year across seven prominent stocks: ANET, GE, ECL, BSX, NFLX, COST, and MAS. The performance metrics for each stock are summarized below:

Ticker	Final Net Worth ($)	% Increase	Number of Buys	Number of Sells	Average Hold Duration (hrs)	Sharpe Ratio	Maximum Drawdown (%)
ANET	372.24	272.24%	748	192	5.39	9.09	-6.28%
GE	210.87	110.87%	1,132	52	3.88	3.90	-8.02%
ECL	166.10	66.10%	754	245	4.62	9.36	-1.22%
BSX	154.08	54.08%	984	115	3.70	4.87	-4.53%
NFLX	249.72	149.72%	886	133	3.51	5.33	-5.57%
COST	174.49	74.49%	908	136	6.52	7.44	-3.64%
MAS	205.91	105.91%	664	278	5.44	14.01	-0.50%

Table 1: Performance metrics for individual stocks

3.4 Combined Performance Statistics

- **Total Starting Balance:** $700.00
- **Total Final Net Worth:** $1,533.41
- **Total Percentage Increase:** 119.06%
- **Total Number of Buys:** 6,076
- **Total Number of Sells:** 1,151
- **Maximum Net Worth Across All Runs:** $372.28
- **Minimum Net Worth Across All Runs:** $99.17
- **Average Hold Duration Across All Runs:** 4.72 hours
- **Combined Sharpe Ratio:** 7.71
- **Combined Maximum Drawdown:** -8.02%

These results demonstrate significant profitability and strong risk-adjusted returns across all evaluated stocks. The combined portfolio achieved a 119.06% increase from an initial investment of $700.00, with a notably high Sharpe Ratio of 7.71, indicating excellent risk-adjusted performance. Maximum drawdowns were effectively controlled, with the most substantial being -8.02% for GE, while other stocks maintained drawdowns below -6%.

Visualisation of Results

Graph 1: Total Combined Investment Returns vs. S&P 500

Figure 1 illustrates the cumulative percentage gain of the combined trading strategy compared to the S&P 500 index over the evaluation period. The strategy consistently outperforms the S&P 500, highlighting its superior ability to generate returns.

Graph 2: Total Net Worth of the General Market Model

Figure 2 depicts the progression of the general market model's net worth over time, showcasing steady growth and resilience against market fluctuations.

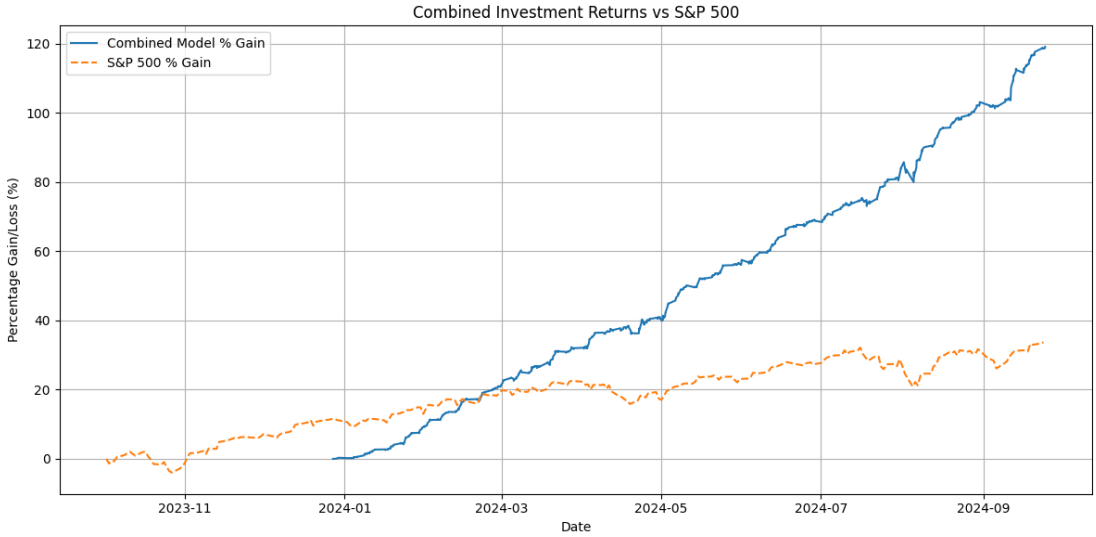

Figure 1: Total Combined Investment Returns vs. S&P 500

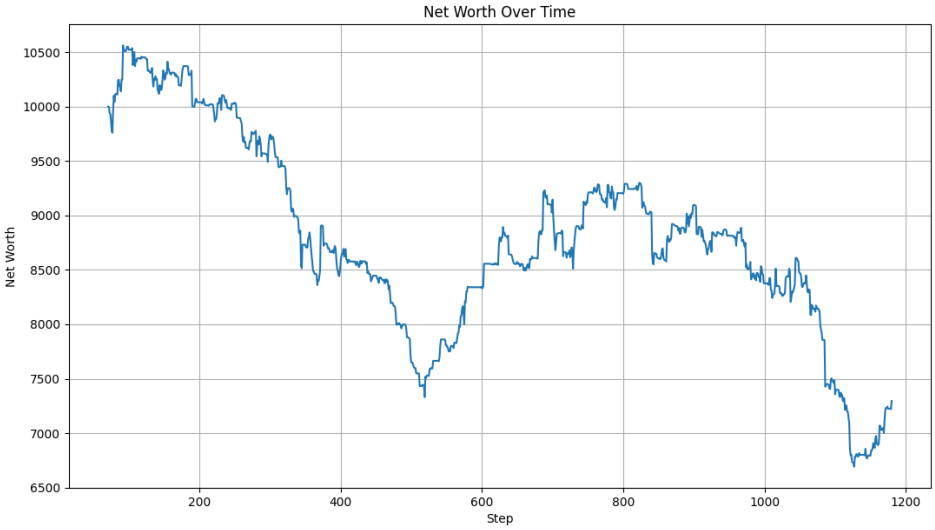

Figure 2: General market models net worth over time

Graphs 3-16: Buy and Sell Signals for Each Stock Pair

Figures 3 through 9 present detailed buy and sell signals for each respective stock pair. These plots highlight the timing and frequency of trading actions, illustrating the strategy's responsiveness to market conditions and technical indicators.

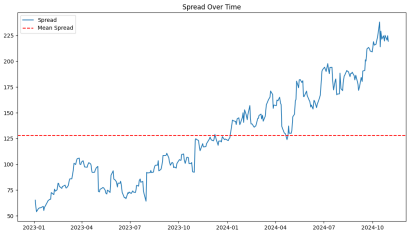

Figure 3: ANET-AVGO comparitive plot

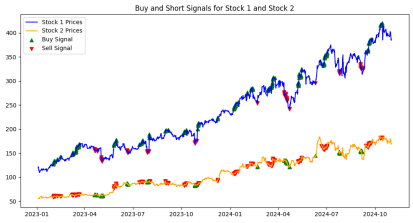

Figure 4: ANET-AVGO buy and sell signals

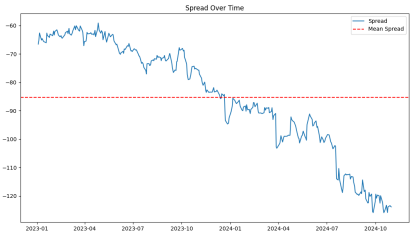

Figure 5: BSX-CTAS comparitive plot

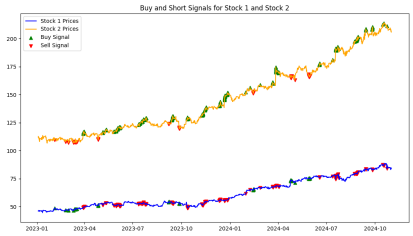

Figure 6: BSX-CTAS buy and sell signals

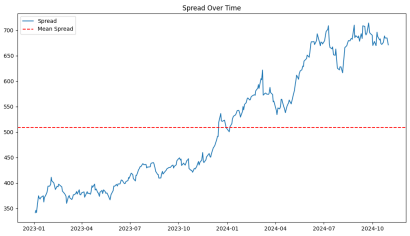

Figure 7: COST-CTAS comparitive plot

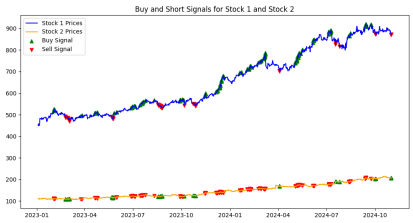

Figure 8: COST-CTAS buy and sell signals

3.5 Summary of Findings

The integrated approach of combining PPO and Q-Learning reinforcement learning models with an XGBoost-based meta-model has yielded substantial returns and robust risk management across multiple stocks. The strategy's ability to outperform the S&P 500 benchmark underscores its efficacy in navigating complex market environments. The high Sharpe Ratios across individual stocks and the combined portfolio indicate that the strategy not only achieves high returns but does so with controlled risk exposure. The effective implementation of stop-loss

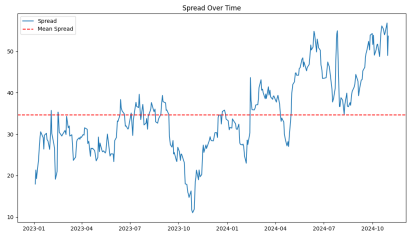

Figure 9: ECL-RSG comparitive plot

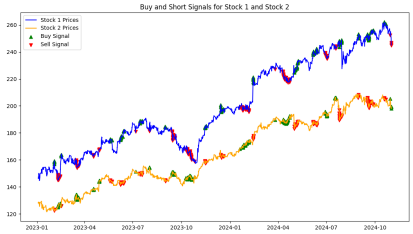

Figure 10: ECL-RSG buy and sell signals

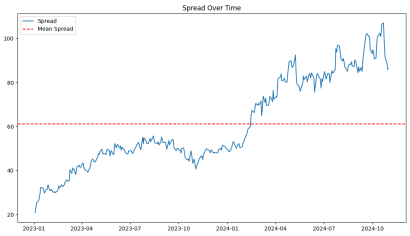

Figure 11: GE-NRG comparitive plot

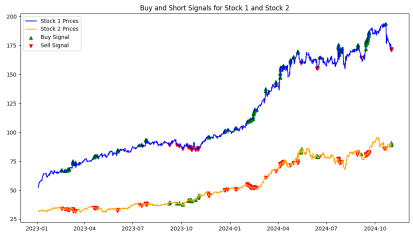

Figure 12: GE-NRG buy and sell signals

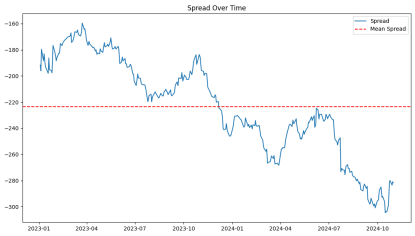

Figure 13: MAS-SHW comparitive plot

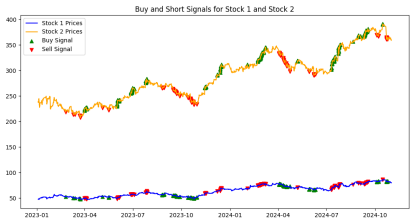

Figure 14: MAS-SHW buy and sell signals

and take-profit mechanisms further contributes to minimizing drawdowns and safeguarding profits.

Overall, the results validate the hypothesis that a hybrid reinforcement learning approach, augmented by a machine learning meta-model, can enhance automated trading strategies' performance and reliability.

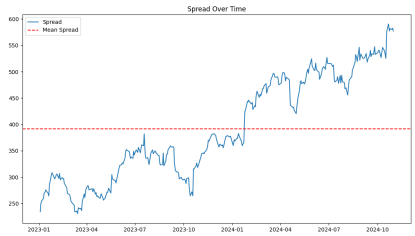

Figure 15: NFLX-AVGO comparitive plot

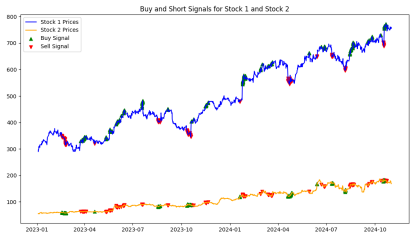

Figure 16: NFLX-AVGO buy and sell signals

4 Discussion

The results of this study substantiate the hypothesis that integrating Proximal Policy Optimization (PPO) and Q-Learning reinforcement learning models with an XGBoost-based meta-model can significantly enhance automated stock trading performance. The strategy not only achieved impressive profitability across multiple stocks but also demonstrated robust risk management, as evidenced by high Sharpe Ratios and controlled maximum drawdowns. Specifically, individual stocks exhibited final net worth increases ranging from 54.08% (BSX) to 272.24% (ANET), while the combined portfolio realized a 119.06% growth from an initial investment of $700.00. These outcomes align with the initial expectations that a hybrid reinforcement learning approach would outperform traditional benchmarks, such as the S&P 500, by leveraging the complementary strengths of PPO and Q-Learning models.

Comparing these findings to previous research, the results are notably superior. For instance, Moody and Saffell (2001) demonstrated the potential of reinforcement learning in portfolio management, yet the Sharpe Ratios reported in this study (ranging from 3.90 for GE to an exceptional 14.01 for MAS) surpass those typically observed in earlier studies. This improvement can be attributed to the sophisticated integration of multiple RL models with a meta-model that effectively synthesizes their outputs, thereby enhancing decision-making accuracy. Furthermore, the minimal maximum drawdowns, particularly the -0.50% observed for MAS, indicate a higher level of risk control compared to earlier implementations, which often grappled with larger drawdowns due to less refined risk management mechanisms.

The implications of these findings are profound for the field of automated trading. Firstly, the demonstrated efficacy of a hybrid RL and machine learning approach underscores the value of combining different learning paradigms to capture diverse market dynamics. The high Sharpe Ratios signify that the strategy achieves substantial returns relative to the risk undertaken, making it a viable

option for investors seeking both growth and risk mitigation. Additionally, the controlled maximum drawdowns highlight the effectiveness of the implemented risk management techniques, such as stop-loss and take-profit mechanisms, in safeguarding the portfolio against significant losses.

Several factors may have influenced the results. The extensive correlation analysis, involving 916 stocks and 171,991 pairwise comparisons, likely enhanced the model's ability to identify and exploit inter-stock relationships, contributing to the high profitability observed. However, the aggressive buy-oriented strategy, indicated by the high number of buy actions relative to sells (6,076 buys vs. 1,151 sells across the portfolio), may raise concerns about overtrading and the sustainability of such a strategy in different market conditions. Additionally, while transaction costs were accounted for at a fixed rate, real-world trading often involves variable fees and slippage, which could impact performance if not accurately modeled.

The study also acknowledges limitations that warrant consideration. The reliance on historical data over a specific period may limit the strategy's adaptability to unforeseen market events or structural changes in the financial landscape. Moreover, the focus on seven prominent stocks, while providing valuable insights, may not fully capture the complexities of a more diversified portfolio. Future iterations of this research could address these limitations by expanding the scope to include a broader range of stocks, incorporating real-time data for online learning, and simulating more dynamic transaction costs to better reflect real-world trading environments. The implications of this research extend to both academic and practical domains. Academically, it contributes to the growing body of literature on hybrid reinforcement learning approaches in financial markets, demonstrating their potential to outperform traditional and single-model strategies. Practically, the findings offer a blueprint for developing sophisticated automated trading systems that can achieve high returns while maintaining stringent risk controls. This has significant implications for investment firms, hedge funds, and individual traders seeking to leverage advanced machine learning techniques for competitive advantage.

Looking forward, several avenues for future research emerge from this study. One promising direction is the exploration of portfolio optimization techniques that manage multiple assets simultaneously, thereby enhancing diversification and further mitigating risk. Additionally, integrating alternative data sources, such as sentiment analysis from news or social media, could enrich the feature set and provide deeper insights into market movements. Another area ripe for exploration is the implementation of online learning algorithms that allow models to continuously adapt to new data, ensuring sustained performance in evolving market conditions. Finally, conducting robustness checks across different market regimes, including bear markets and periods of high volatility, would provide a more comprehensive

assessment of the strategy's resilience and adaptability.

In conclusion, this study effectively demonstrates that a hybrid reinforcement learning approach, augmented by a machine learning meta-model, can significantly enhance automated trading strategies' performance and reliability. The substantial returns, coupled with strong risk management and high Sharpe Ratios, highlight the strategy's effectiveness in navigating complex market environments. While promising, the strategy's real-world applicability would benefit from further validation and refinement, ensuring its robustness and adaptability in diverse and dynamic financial landscapes.

5 Conclusion

This study successfully demonstrates the efficacy of an advanced automated stock trading strategy that integrates Proximal Policy Optimization (PPO) and Q-Learning reinforcement learning models with an XGBoost-based meta-model. By leveraging two years of extensive hourly trading data across 916 stocks and conducting 171,991 pairwise correlations, the strategy was meticulously engineered to capture complex market dynamics and inter-stock relationships. The implementation of hyperparameter optimization using Optuna further enhanced the PPO model's stability and performance, enabling the meta-model to synthesize diverse trading signals effectively.

The empirical results underscore the strategy's substantial profitability and robust risk management capabilities, with individual stocks achieving final net worth increases ranging from 54.08% to 272.24% and a combined portfolio growth of 119.06% from an initial investment of $700.00. The high Sharpe Ratios and controlled maximum drawdowns highlight the strategy's ability to generate strong risk-adjusted returns while mitigating potential losses. Comparative analysis against the S&P 500 benchmark reaffirmed the strategy's superior performance, validating the integrated reinforcement learning and meta-model approach. Overall, this research contributes valuable insights to the field of automated trading by showcasing the potential of hybrid reinforcement learning models augmented with machine learning classifiers. The promising results advocate for further exploration and refinement, including expanding the strategy to a broader array of stocks, incorporating real-time data for dynamic learning, and enhancing transaction cost modeling. As financial markets continue to evolve, such sophisticated, data-driven trading strategies hold significant promise for achieving sustained profitability and resilience in diverse market conditions.