Backtesting and Deploying Algo Trading Strategies

Usam Sersultanov

This documentation describes the following project:

https://github.com/Usam95/backtester.git

Contents

1 Introduction

1.1 Motivation

Trading involves buying and selling assets like stocks and cryptocurrencies to make a profit. With technology's evolution, trading has also advanced, using computers to make trades based on algorithms that analyze market data. These algorithms identify optimal buy or sell moments, benefiting from market trends or inefficiencies. A major advantage of this method is its precision, as computers can process vast data quickly and make consistent, emotion-free decisions. Integrating this with cloud technology enables a 24/7 trading system that can autonomously execute trades in various markets without human intervention.

Backtesting is the process of testing a trading strategy using historical data, allowing traders to evaluate and refine their strategies before they use them in real trading. Driven by my hypothesis, questioning if algorithmic trading can indeed be beneficial, I found myself investing a significant amount of my personal time diving into this topic. Armed with the knowledge I gained and leveraging my data science and software engineering expertise, I started on building this backtesting framework. Throughout this journey, I wrote more than 7,000 lines of code to create a modular and customizable framework designed for evaluating various trading strategies, covering both technical indicators and machine learning methods. While originally designed for cryptocurrency markets, the adaptability of the framework allows its application across various markets, including stocks, energy, and more. As long as the input data conforms to the Open, High, Low, Close, Volume (OHLCV) format, the backtesting framework can be utilized.

This framework leverages the vectorized approach of Python, facilitating compact code, faster execution compared to conventional Python loops, and efficient handling of time series data. Such an approach is particularly advantageous in financial algorithm implementations, especially when it comes to vectorized backtesting. The next section provides a brief overview of the main features of the framework. Detailed explanations of these features are provided in the following chapters.

1.2 Key Highlights

- **Efficient Backtesting**: In the current version of the framework, backtesters for the following technical indicators are implemented:
  - Moving Average Convergence Divergence (MACD)
  - Exponential Moving Average (EMA)
  - Relative Strength Index (RSI)
  - Bollinger Bands (BB)
  - Stochastic Oscillator (SO)

  Additionally, various classification machine learning models are supported, as detailed in the next Chapter 4. Furthermore, the system is designed with modularity in mind, ensuring easy extensibility. New strategies can be seamlessly integrated by

simply adding new classes with the desired implementations, eliminating the need to modify previously implemented and validated code.

- **Data Source**: The data for this project is directly sourced from Binance using the `data_retriever` module, primarily in 1-minute intervals. The data for this project is directly sourced from Binance using the `data_retriever` module, primarily in 1-minute intervals.

- **Optimization Support**: The strategies using based on technical indicators and machine learning models can be backtested and fine-tuned using different performance measures. Optimization settings can be specified in a dedicated JSON configuration file (see lst. 3.1 for an example).Currently, the optimization supports GridSearch for both technical indicators and machine learning models, as well as Bayesian Optimization for technical indicators.

- **Configuration Support**: As mentioned earlier, both technical indicators and machine learning models can be configured for optimization using JSON configuration files.These files also allow to easily set up deployment settings, including choosing algorithms, deciding which symbols to trade, and setting other parameters.

- **Deployment Simplified**: After backtesting, the optimized amd backtested strategies can be used for actual trading. The deployment infrastructure is designed to be adaptable and can be extended to support brokers with API capabilities for algorithmic trading. The deployed trading system can manage multiple symbols, applying various strategies to each symbol at the same time.

- **Notifications Support**: Thanks to integrated logging mechanisms and email notifications, users remain consistently informed regarding backtesting outcomes and live trading activities.

2 Setup Installation and Usage

2.1 Installation

To set up the environment for backtesting and deploying trading strategies, follow the steps below:

2.1.1 Prerequisites:

Ensure that you have Anaconda or Miniconda installed on your machine. If not, download and install the appropriate version for your operating system.

2.1.2 Steps:

1. **Navigate to the project directory:** Open a terminal and navigate to the project directory where the `requirements.yaml` file is located. This file contains all the necessary package dependencies and configurations for the backtesting environment.

2. **Create the Conda environment:** Run the following command to create the `trading` environment using the configurations specified in the `requirements.yaml` file:

   ```
   conda env create -f requirements.yaml
   ```

   This command will set up a Conda environment named `trading` with Python version 3.8 and all the required packages.

3. **Activate the environment:**

   After the environment is successfully created, it can be activated with the following command:

   ```
   conda activate trading
   ```

2.2 Usage

TODO

3 Technical Indicator based Backtesting

3.1 Input Data

All the cryptocurrency historical data used in this project was downloaded using the `data_retriever` module, which establishes a connection with the Binance server and fetches data for defined crypto symbols (a crypto symbol represents a specific cryptocurrency, like BTC for Bitcoin or ETH for Ethereum) over a specified period. The downloaded data is stored in the `historical_data` directory. Each symbol has its own directory, where the corresponding data is saved in a `.parquet.gzip` format. Data is downloaded at a one-minute frequency but can be downsampled to longer intervals, such as 1 hour or 1 day.

The `load_data.py` module facilitates loading the historical data of a symbol from its `.parquet.gzip` file for use in backtesting strategies.

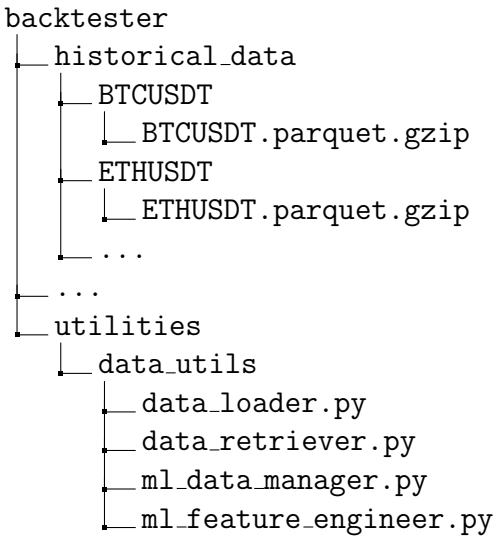


Figure 3.1: Directory Structure Highlighting Data-Related Modules and Files.

3.2 Technical Indicator-based Strategies

Technical indicators are statistical calculations based on historical price and volume information that aim to predict future price movements. Commonly used in technical analysis, these indicators provide insight into the market's direction, strength, momentum, and volatility. The backtesting architecture, including the necessary classes and components for strategies based on technical indicators is depicted in UML diagram 3.2. One of the main advantages of the framework is its extensibility. It allows the integration and configuration of new strategies. To integrate a new algorithm, it should inherit from the `BacktesterBase` class found in the `backtester_base.py` module. Additionally, it must implement the following methods: `prepare_data`, `optimize_strategy`, `objective()`, and `test_strategy()`. Figure 3.2 showcases the framework's architecture specifically for technical indicator-based backtesting. The `MacdBacktester` class (high-

lighted in red) inherits common attributes and methods from the `BacktesterBase` class, such as `get_data`, `run_backtest`, `add_stop_loss`, and `add_take_profit`, among others.

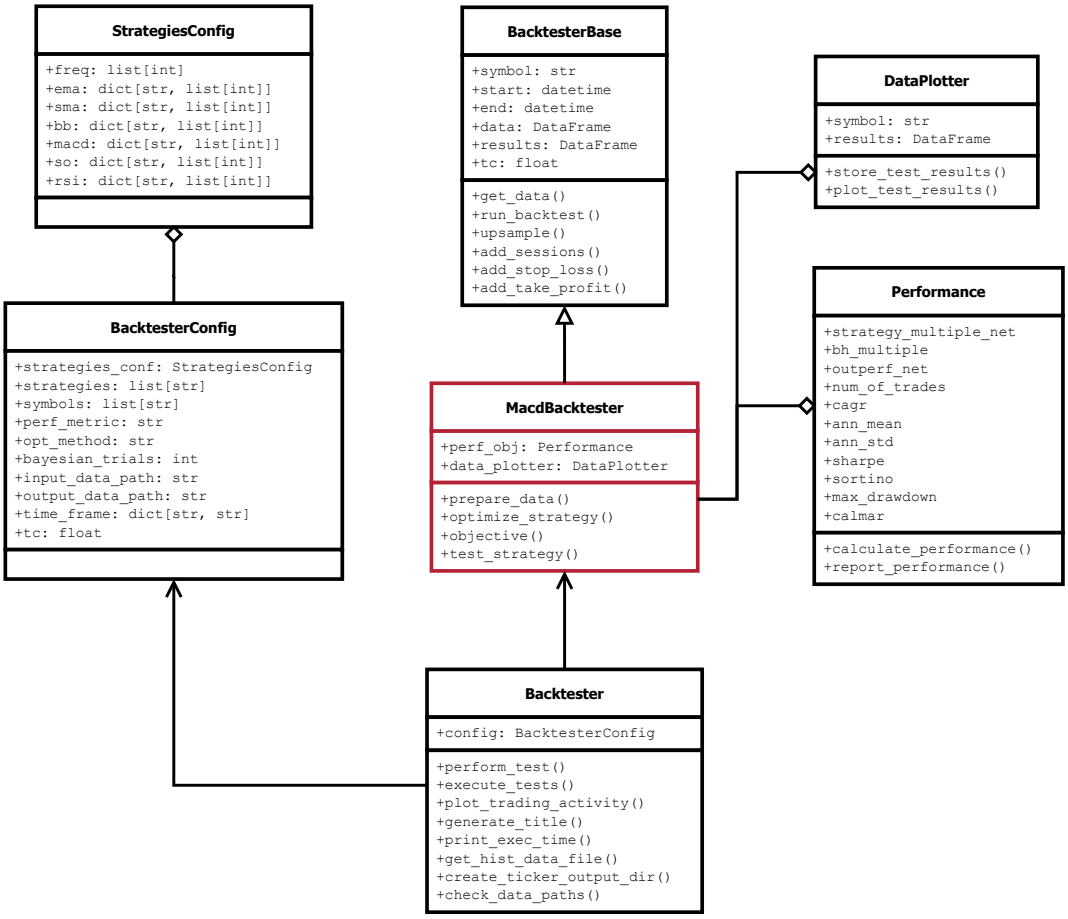


Figure 3.2: Architecture diagram of technical indicator-based backtesting component of the framework.

The `DataPlotter` and `Performance` classes are utilized to compute and visually display performance metrics for each test run. The `Backtester` class manages the entire backtesting process, leveraging the functionalities of other classes to perform its tasks. The `BacktesterConfig` class represents the configuration, which is validated and ingested in the `Backtester` class via the Python `pydantic` library.

For optimization, either the grid search method or the Bayesian method can be chosen. The former comprehensively explores all possible combinations in the parameter ranges, while the latter utilizes probability distributions to refine its search. Depending on the defined parameter ranges for each algorithm, a vast array of combinations, potentially in the tens of thousands, can be assessed for each symbol. Performance results for all these tests are saved into a dedicated `{symbol}.csv` file located in the `results/{symbol}` directory, facilitating subsequent analysis and visualizations.

The Listing 3.1 shows how the Backtester is set up to run backtests for XRP, LTC, and TRX with an EMA-based strategy. The Bayesian method is used for optimization, and EMA parameter ranges are defined. Each symbol's results will be stored in its respective .csv file.

```
{
  "opt_method": "bayesian",
  "bayesian_trials": 1500,
  "time_frame": {
    "start_date": "",
    "end_date": "2022-12-31 23:59:00"
  },
  "symbols": ["XRPUSDT", "LTCUSDT", "TRXUSDT"],
  "strategies_config":
  {
    "freq" : [5, 125, 5],
    "ema": {
       "ema_s": [6, 60, 4],
       "ema_l": [10, 200, 5]
    }
  }
}
```

Listing 3.1: Backtesting configuration

Furthermore, individual backtesting results for each symbol and applied strategy can be visualized, as shown in fig. 3.3. This figure displays the effectiveness of the optimized strategy compared to the buy-and-hold approach. As an example, the `EMA` strategy was used for the `LTCUSDT` symbol. All pertinent parameters are illustrated above the primary plot. Testing was executed on historical data ranging from 2021-01-01 to 2021-12-31.

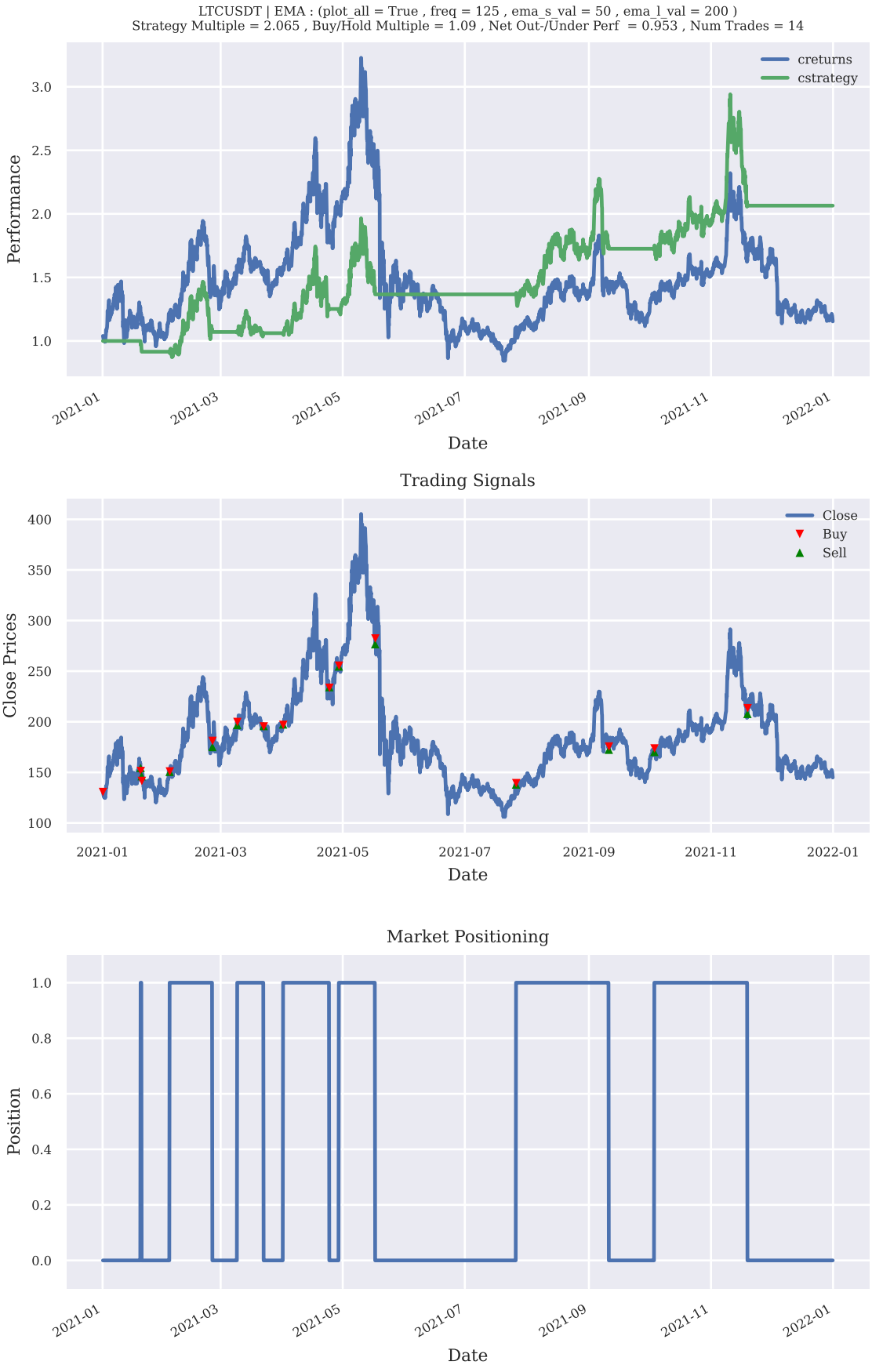


Figure 3.3: Results of Backtesting of EMA Stragegy of LTCUSDT.

The secondary plot, demonstrates the close prices of LTCUSDT, marked with buy (red triangle) and sell (green triangle) signals generated by the EMA strategy. The bottom plot, "Market Positioning", illustrates the duration of each position, with 1 indicating a long position and 0 no position.

```
LTCUSDT | EMA : (freq = 125 , ema_s_val = 50 , ema_l_val = 200)
Strategy Multiple = 2.065 , Buy/Hold Multiple = 1.09
Net Out-/Under Perf = 0.953 , Num Trades = 14
```

Table 3.1: Backtesting results and parameters for the EMA strategy on LTCUSDT .

Table 3.1 provides a detailed overview of the parameters used for the EMA strategy, such as frequencies, long and short window periods as well as performance metrics. The strategy performed quite well and outperformed the buy-and-hold benchmark, achieving an outperformance of a 95%.

4 Machine Learning based Backtesting

In the world of machine learning (ML), classification is about sorting things into categories. When applied to trading, ML Models predict whether the price will go up or down based on today's as well as historical data. Using ML for algorithmic trading requires a systematic approach. This structured process is visualized as a pipeline, detailed in fig. 4.1.

4.1 ML Trading Pipeline

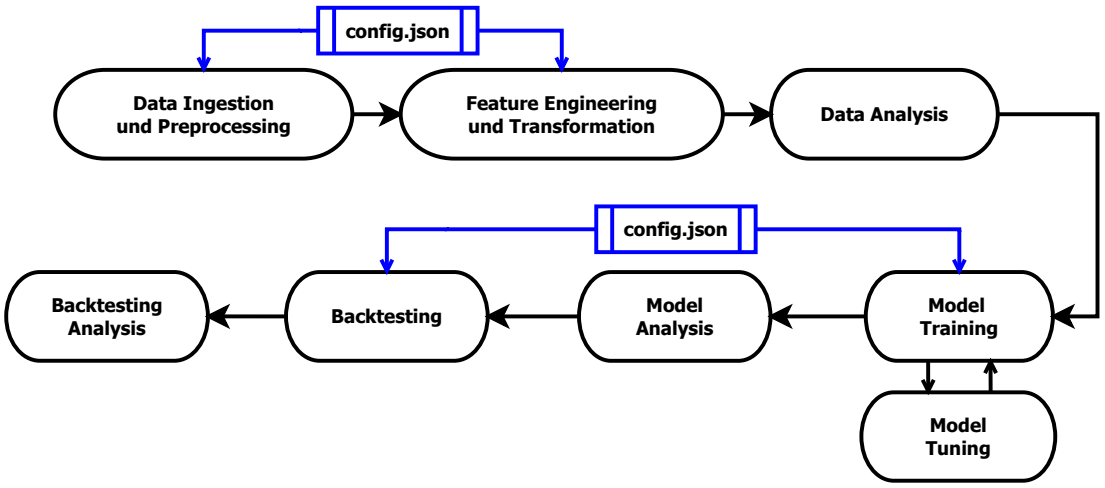


Figure 4.1: Machine Learning Trading Pipeline: Process steps from data acquisition and preprocessing, feature engineering and transformation, model training and optimization, to the final execution and evaluation of trades based on model predictions.

The process begins by loading historical data previously downloaded from Binance and then proceeds with backtesting using the trained and optimized ML model on that data. The pipeline steps can also be executed independently. For instance, starting with data ingestion and preprocessing can be followed by feature engineering and data visualization to gain a deeper understanding of the dataset.The steps of the pipeline in fig. 4.1 are detailed in the subsequent subsections.

4.1.1 ML Trading Pipeline Configuration

The JSON configuration in lst. 4.1 is used to set up the machine learning pipeline. The `dataset_conf` element specifies details such as the cryptocurrency symbol to be backtested (BTCUSDT in this case), the frequency of the time series data, and the start and end dates. Furthermore, it provides the option to specify the date for splitting the dataset into training and test sets. The configuration also includes options to set the desired model and specify the parameter ranges. This enables fine-tuning via grid search for the selection of optimal parameters. Also the `target_conf` element allows for the

selection of the target for the classification model. There are several target variants to choose from, that are describe later.

```
{
  "tc": 0.001,
  "dataset_conf": {
     "symbol": "BTCUSDT",
     "freq": "1d",
     "start_date": "2018-01-01 00:00:00",
     "end_date": "2023-08-30 23:59:00",
     "split_date": "2022-12-31 23:59:00",
     "mode": "full",
     "target_conf": {
       "target": "Exceed_Avg_Returns"
    }
  },
  "model_name": "RandomForestClassifier",
  "model_type": "classification",
  "models_config": [
     {
       "model": "AdaBoostClassifier",
       "params": {
          "n_estimators": [10, 30, 100],
          "learning_rate": [0.01, 0.1, 1.0],
          "algorithm": ["SAMME", "SAMME.R"]
       }
     }
  ]
}
```

Listing 4.1: Machine Learning Pipeline Configuration

4.1.2 Data Ingestion and Preprocessing

The first step involves ingesting the data required for backtesting. This data can be sourced from Binance using the `data_retrieve.py` module, as detailed in the previous chapter. The data is saved in the `historical_data` folder, which is located within the corresponding symbol's subfolder, created during the data download process. After loading the stored data into pipeline, is downsampled based on the bar length specified in the configuration (see 4.1, where the `freq` parameter is set to `1d`. Furthermore, a validation check is run to ensure data integrity.

4.1.3 Feature Engineering

Feature engineering is a critical step in improving the model's predictive capability. The goal is to identify features that could potentially influence the model's performance. Technical indicators are utilized to create features that help models predict upcoming price movements based on historical market data. The technical indicators used in the `ml_feature_engineer` module can be broadly categorized into momentum-based,

trend-based, and volume-based. Additionally, in this pipeline stage, more features can be created and added into the dataset. This can be achieved by adding the new relevant methods to the `FeatureEngineer` class. Ths methods should be called within the `add_features` method of the same class.

- **Momentum-based Indicators:** These are primarily concerned with the speed of price movements. Features like Rate of Change (ROC), Momentum, variations of the Stochastic Oscillator, and the Relative Strength Index (RSI) fall into this category.
- **Trend-based Indicators:** Aimed at identifying the movement direction over time, these include indicators such as the Moving Average (MA) and the Exponential Moving Average (EMA).
- **Volume-based Indicators:** Volume, in trading, refers to the number of shares or contracts traded in a security or market. The feature set for this category includes the On-Balance Volume (OBV).

```
def add_features(self):
    """Add various technical indicators to the dataset."""
    for n in self.periods:
        self._add_ma(n)
        self._add_ema(n)
        self._add_rsi(n)
        self._add_sto(n)
        self._add_mom(n)
        self._add_roc(n)
        self._add_stos(n)
        self._add_stomom(n)
        self._add_storoc(n)
        self._add_stoch_rsi(n)
        self._add_sto_cross(n)
    self._add_obv()
    self._add_returns()
    self._add_rolling_cum_returns()
    self._add_rolling_cum_range()
    self._add_day_of_week()
    self._calculate_range()
```

Listing 4.2: Function to add features to the dataset

Following this, a target for classification ML model is computed, which can also be specified in the configuration file. The available target variants include:

- **Simple**: Predicts if the next day's price will go up or down based on today's closing price.
- **MA_Relative**: Checks if today's closing price is above its short-term average and rising.
- **Momentum**: Examines if the stock's momentum is positive and increasing.

- **ROC**: Looks at if the rate of change (ROC) is positive and on the rise.
- **Consecutive_Increases**: Predicts if returns will rise for the third consecutive time after a decline.

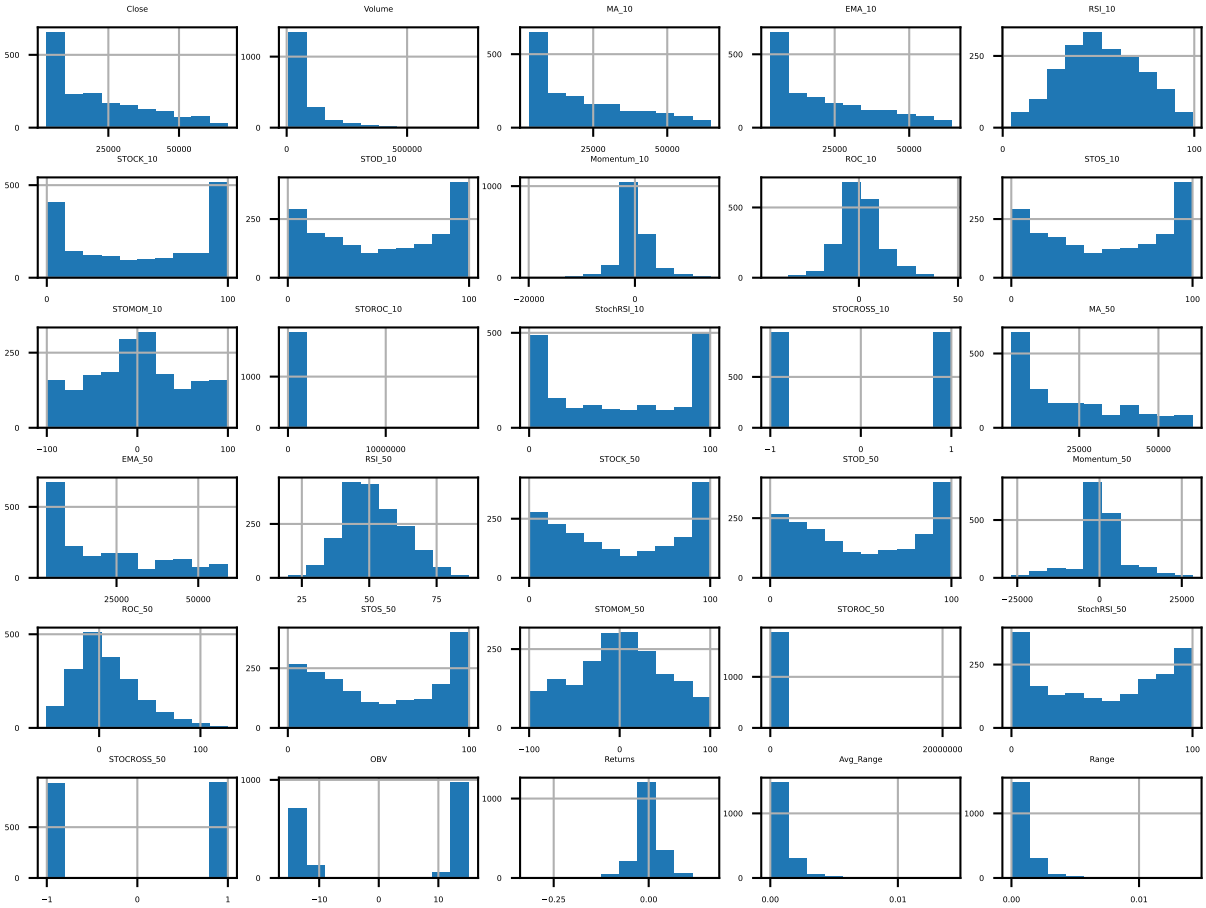


Figure 4.2: Histogram of created features from the BTCUSDT historical data.

4.1.4 Feature Transformation

After engineering the features, it's essential to transform them to meet the specific requirements of machine learning models. Many algorithms have assumptions about the data's scale and distribution. Hence, features undergo in this pipeline step the transformations using techniques like `Standardization` and MinMax Scaling.

It's important to apply the same transformations with according parameters, to both the training and testing datasets or any new incoming data. This ensures consistency in model predictions.

The entire transformation workflow, from data loading, preprocessing, to feature-related operations, is managed by the `DataManager` and `FeatureEngineer` classes (see fig. 4.7).

4.1.5 Data Visualization

For data analysis the `ml_model_evaluator` module was implemented with appropriate methods (see fig. 4.7). It provides tools for both data and model checks. The prepared data can be visualized in various ways, including target distributions, feature covariance matrices, and feature histograms.

The **distribution of the predicted variable** across both training and validation datasets is showed in fig. 4.3. Notably, the distribution between the two classes (1's and 0's) is nearly balanced in both datasets. Such a balanced distribution is crucial for training machine learning models effectively, ensuring that they aren't biased towards one class.

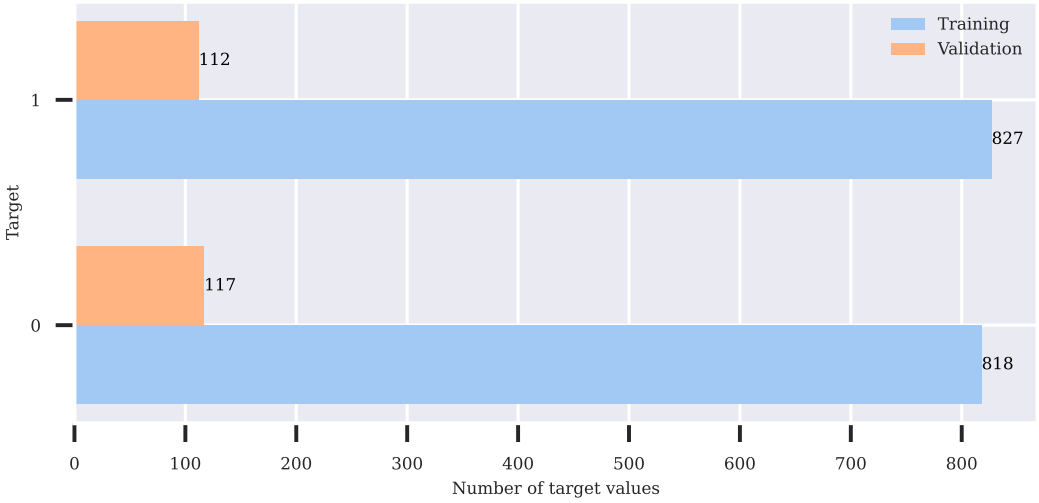


Figure 4.3: The distribution of target values in both the training testing datasets.

The fig. 4.4 illustrates the **correlation coefficients** between various feature variables in the dataset. Each cell in the matrix represents the degree of correlation between two features: a value close to 1, represented by dark green, signifies a strong positive correlation, while a value close to -1, shown in white, indicates a strong negative correlation.

The **histogram** fig. 4.2 presents the distribution of engineered features derived from the BTCUSDT historical data. Each bar in the histogram represents the frequency of occurrences for a particular value or range of values. This visualization provides insights into the distribution patterns and potential skewness of the different features in the dataset.

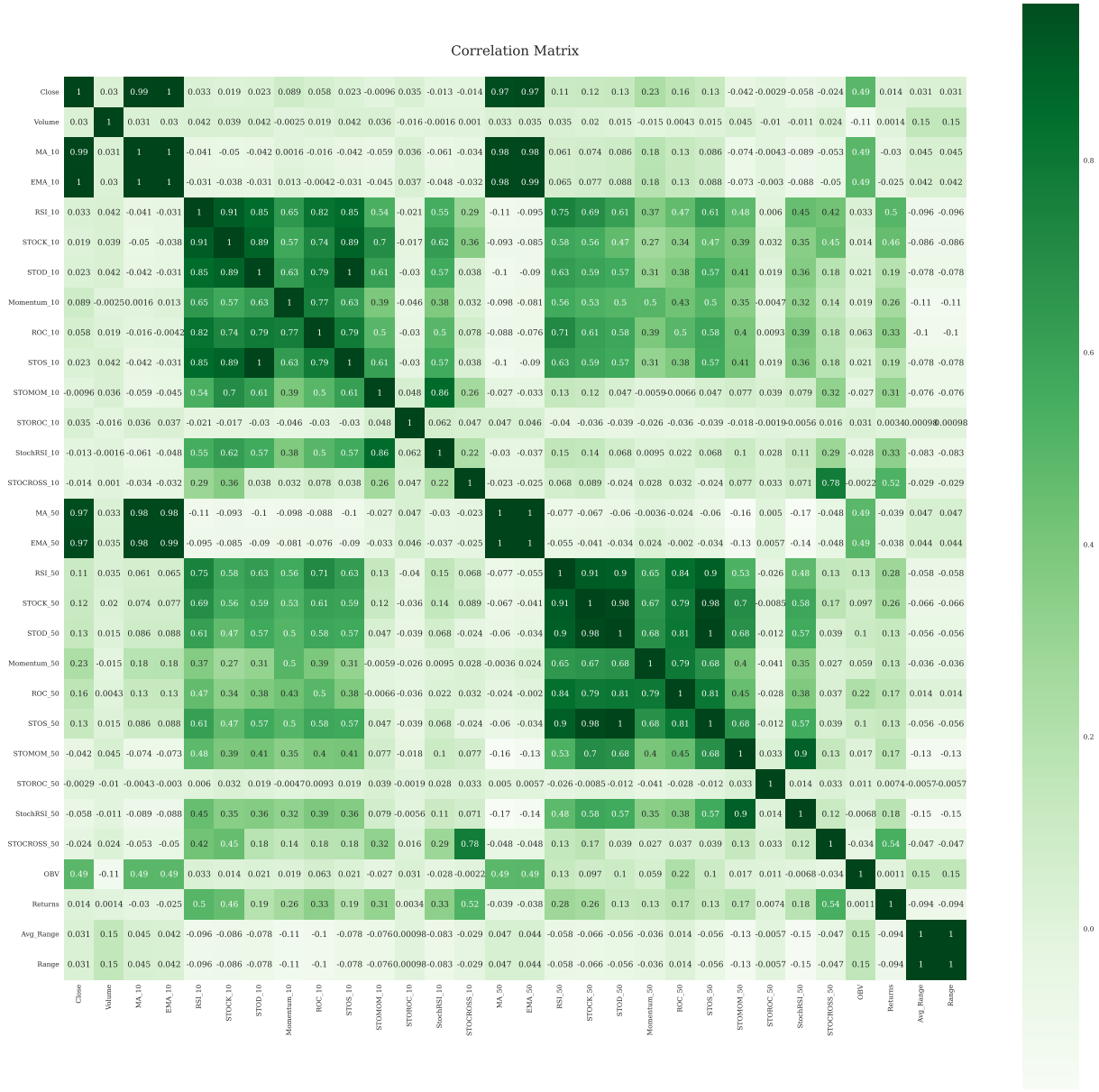


Figure 4.4: Correlation Coefficient between features variables.

4.1.6 Model Training and Optimization

The logic for model training and backtesting is encapsulated within the `ml_backtester.py` module, while the `ml_optimizer.py` module is dedicated to optimization purposes, as depicted in fig. 4.7. The `_optimize_model` function in the `ml_backtester.py` module is responsible for the model optimization process. It utilizes the instance of the `MlOptimizer` class to perform searching for the best parameters of the model using techniques such as Grid Search with Cross-Validation. In a grid search, a grid of all possible hyperparameter combinations is created, and the model is trained using each one of them.

```
def _optimize_model(self):
    # Instantiate the GridSearchOptimizer
    optimizer = MlOptimizer(self.data_manager.X_train,
                                    self.data_manager.y_train,
                                    self.config.dataset_conf.symbol,
                                    task_type=self.config.model_type,
                                    num_folds=self.config.num_folds,
                                    scoring=self.config.scoring)

    # Optimize the model using configured parameters grid
    best_model = optimizer.grid_search(self.config.model_name,
                                       self.model,
                                       self.get_model_params_grid())

    # Update the model of the strategy to the optimized model
    self.model = best_model

def _generate_predictions(self):
    '''Generate predictions using the model.'''
    predictions = self.model.predict(self.data.X_train)
    self.data['predictions'] = predictions
    self.data['predictions'].ffill().fillna(0, inplace=True)
```

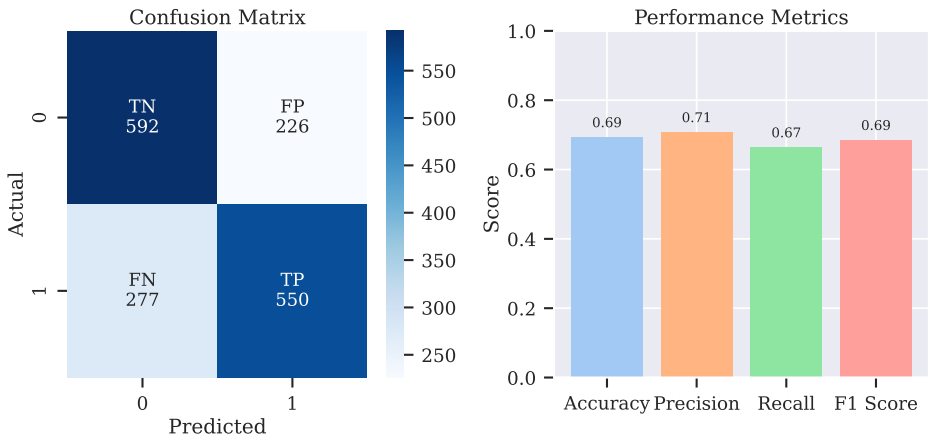
Listing 4.3: Functions of MlBacktester class for model optimization and predictions.

The best model is then used in the `_generate_predictions` function to predict price movements trends on unseen data. Once optimized, the trained model specific to a cryptocurrency can be saved for subsequent use and reloaded as needed. As previously mentioned, all model details including the model type (for instance, AdaBoostClassifier) and its hyperparameter ranges can be configured in the appropriate config file (see lst. 4.1).

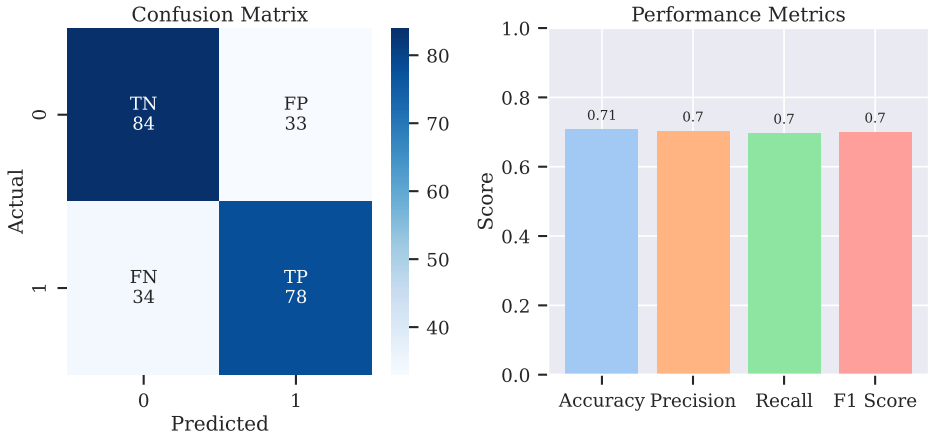
4.1.7 Model Evaluation using Classification Metrics

Once the trading model is trained, its performance can be assessed using various metrics. These metrics provide insights into the model's predictions and highlight potential areas for improvement. The performance data for both the training and testing of the model are visualized in fig. 4.5. For each phase (training and testing): The left diagram presents the confusion matrix, which shows the number of correct and incorrect predictions made by the model. The right diagram visualizes key performance metrics, specifically: accuracy, precision, recall, and F1 score. Each metric is also further described in more detail below.

- **True Positive:** Represents the number of times the model correctly predicted an upward trend and the price actually went up.
- **True Negative:** Refers to the instances where the model accurately predicted a downward trend, and the price indeed decreased.



(a) Model Performance Metrics based on Training Data.



(b) Model Performance Metrics based on Testing Data.

Figure 4.5: Visualization of the Classification Report.

- **False Positive:** Represents instances where the model incorrectly predicted an upward trend, but the price either decreased or remained unchanged. Such errors could lead to missed shorting opportunities or potential losses in Long Only strategies.

- **False Negative:** Represents the number of times the model incorrectly predicted the downward trend and the price went up or stayed the same. This type of error could result in missed profit opportunities due to the decision not to buy.

- **Recall:** The ratio of the number of correct positive predictions (TP) to the total actual positives (TP + FN). In mathematical terms, Recall = $\frac{TP}{TP+FN}$. In trading, it indicates how well the model identifies actual upward movements. A high recall means the model captures most of the upward trends, but this can also be achieved at the expense of more false positives.

- **Precision:** The ratio of correct positive predictions (TP) to the total predicted positives (TP + FP). In mathematical terms, Precision = $\frac{TP}{TP+FP}$. Shows how many of the predicted upward trends by the model were actually correct. High precision means that when the model predicts an upward trend, it's likely correct.

But a higher precision might come at the expense of missing some actual upward trends (lower recall).

- **Accuracy:** The ratio of correct predictions (both TP and TN) to the total number of predictions (TP + TN + FP + FN). In mathematical terms, Accuracy $= \frac{TP+TN}{TP+TN+FP+FN}$. In trading, it gives an overall measure of how often the model is correct, regardless of whether it's predicting upward or downward movement.

- **F1 Score:** The harmonic mean of precision and recall, providing a balance between the two.

- It is given by the formula: $\text{F1} = 2 \times \frac{\text{Precision} \times \text{Recall}}{\text{Precision} + \text{Recall}}$. It's especially useful when the class distribution is imbalanced. In trading, a high F1 score suggests the model has a balanced performance in terms of identifying upward trends and avoiding false alarms.

4.1.8 Backtesting and Analysis

After the model is trained and fine-tuned, it's crucial to check how good it is by testing its effectiveness not just on the basis of typical classification metrics but also on how well it performs in a simulated trading environment.

The code listing 4.4 showcases the `run_strategy` function, a part of the `MlBacktester` class. This function is designed to run the backtesting process.

```
def run_strategy(self):
    '''Backtests the trading strategy.'''
    self._generate_predictions()
    self._calculate_strategy_values()
    self._calculate_trades()
    self._adjust_for_transaction_costs()
    self._calculate_cumulative_metrics()
    self.perf_evaluator._calculate_perf(self.data)
```

Listing 4.4: Function of MlBacktester class for backtesting execution.

After generating predictions, the strategy values are calculated. These strategy values reflect the positions the model would take based on its predictions. Further, trades are determined and adjustments are made to account for transaction costs, which are often overlooked but can significantly impact net returns in real-world scenarios. Finally, cumulative metrics are computed, followed by a performance evaluation to assess the overall success of the trading strategy. By summing the strategy returns, the model's hypothetical profitability over the backtesting period is determined.

A common benchmark to assess the strategy's success is the "buy and hold" method. In this approach, the asset is bought and kept throughout the entire period without incurring extra costs or decisions based on price changes. Comparing the total strategy returns to the "buy and hold" method's returns provides an understanding of the ML-based strategy's added benefit. If the model outperformes this benchmark, it can be considered as useful for trading. The fig. 4.6 illustrates the comparison between the

Figure 4.6: Feature Significance in Model Prediction

returns of the "buy and hold" strategy and those of the configured `AdaBoostClassifier` model's strategy. The model's strategy achieved a return multiple of 4.376.This surpasses the 3.633 return from the "buy and hold" benchmark, resulting in a net outperformance of 0.743, or 74.3%. The observed outperformance corresponds to the final date of the dataset, which is the end of the backtesting period. This date can be configured in the configuration file as shown in lst. 4.1. Moreover, the analysis took into account Binance's trading fees of 0.1% for spot trading, ensuring a more realistic representation of potential real-world returns. The backtesting results show the potential of the model to make profits for real-world trading scenarios.

4.2 ML Pipeline Architecture

The architecture of the pipeline is visually depicted in a UML diagram, as shown in fig. 4.7. The main part of this diagram is the `MlBacktester` class. This class works as the main controller, managing different tasks and using other classes for specific jobs that represent the ML Pipeline 4.1, as described in the previous section 4.1.

The configuration of the model, its parameters, and the dataset is managed by the `BacktesterConfig` class. Data preprocessing, feature engineering, transformation, and

selection are handled by the `MlDataManager` and `MlFeatureEngineer` classes.

For optimization purposes, `MlOptimizer` fine-tunes the model based on the configuration provided in `MlBacktesterConfig`. This class collaborates with `MlModelFactory` to instantiate models leveraging the `sklearn` library.

Visualization related to model training and dataset analysis is done by the `MlDataPlotter` class. The `Performance` class computes the performance of both the machine learning-based strategy and its corresponding benchmark. The same class was utilized for Technical Indicator-based Backtesting from the previous chapter 3.

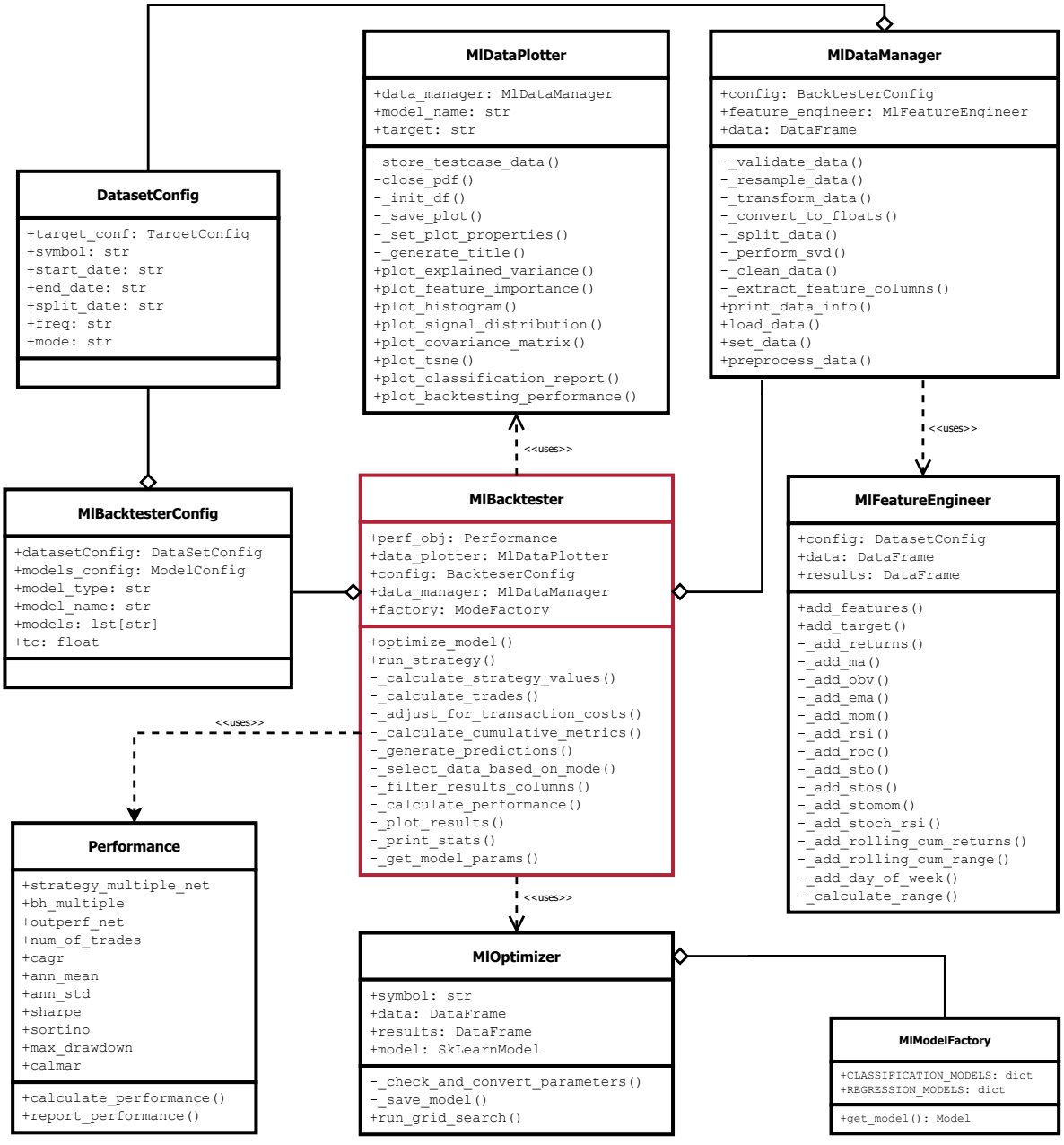


Figure 4.7: Architecture diagram of technical ML-based backtesting component of the framework.

4.3 Feature Importance

Feature importance gives a clear measure to understand how much each feature matters when the model makes predictions. Tree-based models like the `AdaBoostClassifier`, referenced in fig. 4.6 and commonly found in the scikit-learn library, have an attribute called `feature_importances_`‘. This attribute derives its importance from the structure of trees and is inherent to models such as `Decision Trees`, `Random Forests`, `Extra Trees`, `Gradient Boosted Trees`, and `AdaBoost` when the base estimator is a **decision tree**.

The analysis in fig. 4.8 shows which features have the most impact on predictions. Features like `STOCK_10`, `STOROC_10`, and `RSI_10` are among the most important. Other features, such as `STOD_10` and `Stoch_RSI_10`, are also considerably important. These features, based on momentum indicators and primarily derived from recent periods, play a crucial role in predicting the outcome. Consequently, their inclusion has enhanced the model's predictions.

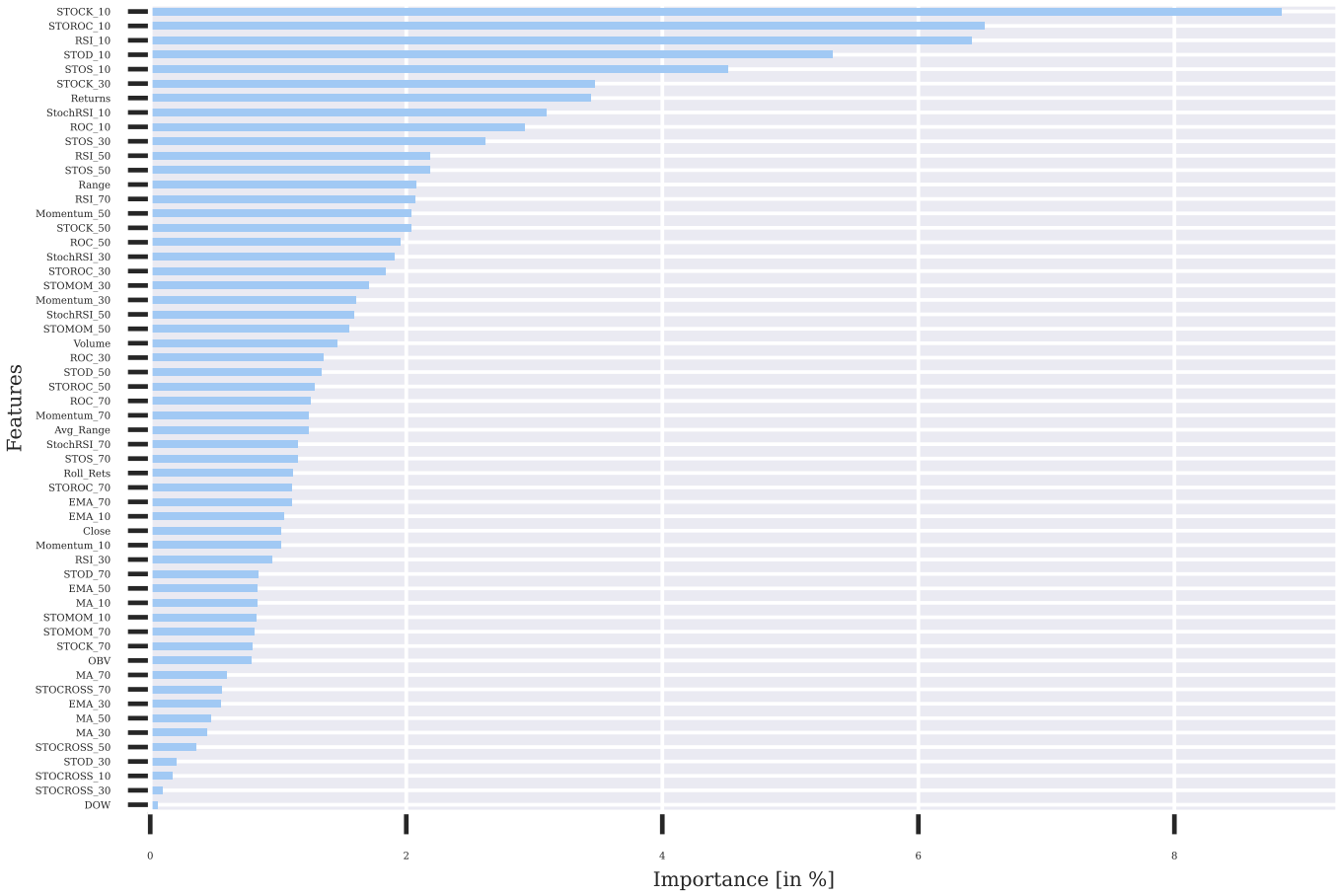


Figure 4.8: Feature Significance in Model Prediction

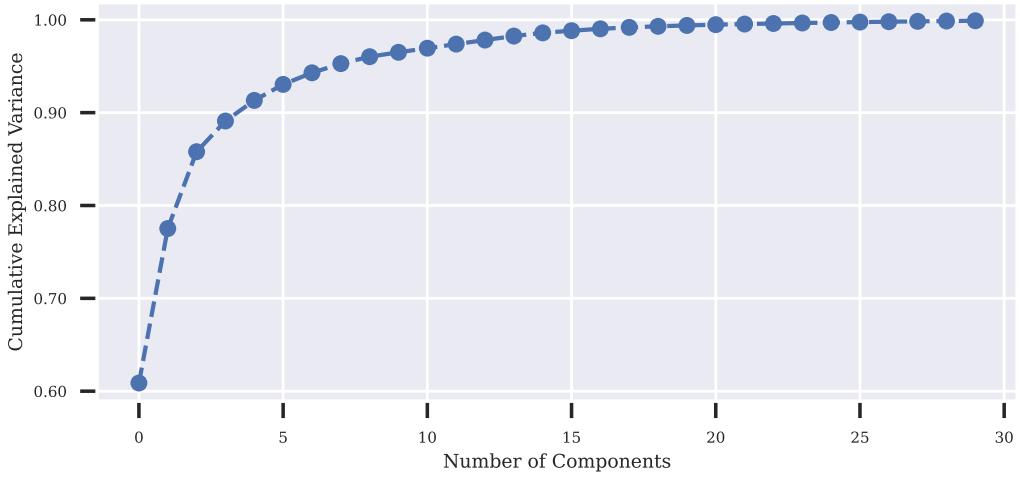


Figure 4.9: Explained Variance of Features after Performing Feature Reduction.

4.4 Feature Reduction

In trading too many features in dataset might sometimes weaken the model's performance, since many of these features can be redundant or just add noise. Furthermore, recalculating all these features every time the new market data arrives can be time-consuming, which can slow down the model. To address this, dimensionality reduction techniques are employed. Through these methods, the number of features is reduced while preserving most of the important information. This makes model training faster and helps avoid problems like overfitting, where a model might perform well on training data but not on new, unseen data. The implemented ML Pipeline supports dimensionality reduction, providing the method `perform_svd` of `MlDataManager` that utilizes the `TruncatedSVD` module from the `sklearn` library, reducing the features through Principal Component Analysis (PCA) based on Singular Value Decomposition (SVD). The method `plot_explained_variance` of the `MlPlotter` class visualizes the reduced features and their aggregated variance, as depicted in fig. 4.9. This figure shows that just 10 components capture over 95% of the important information in our dataset. This indicates that feature reduction can be very effective while still keeping most of the valuable data insights.

5 Deployment

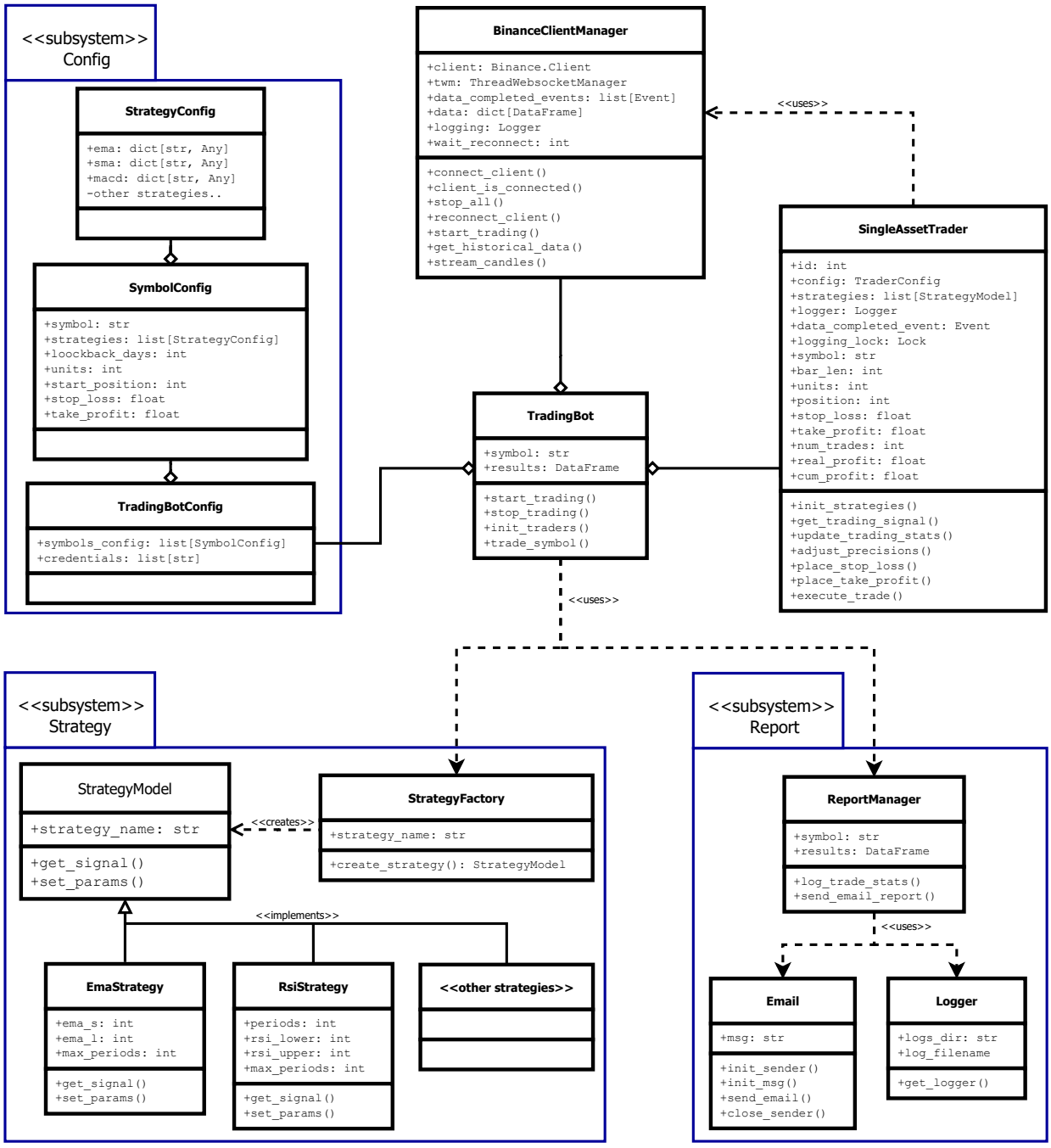


Figure 5.1: Architecture of the Automated Bot for Trading on the Binance Platform.

The deployment component of our backtesting framework is currently designed to support the Binance cryptocurrency trading platform. However, it has been architected to be extensible, allowing for potential integration with other trading platforms like Interactive

Brokers for stock trading. For any new trading platform, a dedicated client class must be provided. In the case of Binance, all platform-specific functionalities are encapsulated within the `BinanceClientManager` class (see fig. 5.1). Any new client class should implement essential methods like `connect_client`, `start_trading`, `get_historical_data`, and `stream_candle`.

The deployment is designed to support trading for multiple symbols using different strategies. Because each symbol requires unique trading details, historical data, and strategy settings, a `SingleAssetTrader` class instance is initiated for each symbol. Each of these instances operates on a separate thread, with the `TradingManager` class facilitating synchronization and coordination between them.

Configuration for the deployment is handled via a JSON config file. This file specifies details like which symbols to trade on, the strategies for each symbol, their respective parameters, and more. The `StrategyFactory` is responsible for instantiating the defined strategy models for each symbol. These models use the retrieved historical data to generate trading signals. The `SingleAssetTrader` instances then act on these signals, placing buy or sell trades as necessary.

All trading activities are logged using a central logging system. Additionally, the framework supports email notifications for key trading events.

6 Conclusion and Future Work

6.1 Conclusion

This project was a big challenge, and I really learned a lot in this journey. It made me even more excited about algorithmic trading and the application of data science for it. I'm now eager to build better models, explore different assets and markets, and add new features into the backtesting framework as mentioned in the next section for "Future Work". I'm more motivated than ever to push my limits, see what more I can achieve, and am very curious about the outcomes of the experiments.

6.2 Further Work

In the continued development of the backtesting framework, several enhancements and updates are planned:

- Implementation of regression-based and deep learning-based backtesting strategies.
- Development of an assembler that leverages multiple strategies from diverse machine learning and deep learning paradigms such as regression and classification.
- Integration of deep reinforcement learning for advanced trading strategies.
- Consideration for automated portfolio management functionalities.
- Exploring the incorporation of Natural Language Understanding for sentiment analysis. Potential data sources for this sentiment analysis include platforms like Twitter, Reddit, and other trading-relevant sources.
- Adding notifications about trades using a Telegram bot.
- Adding support for backtesting and trading stocks with the Interactive Brokers platform.
- Incorporating more optimization techniques.
- Configuration options for backtesting and deployment for both long-only and long-short strategies.
- Adding a graphical user interface for the framework, enhancing user interaction and ease of use.

Adding these features will enhance the flexibility of the backtesting framework, making it more resilient, and accurate.

Bibliography