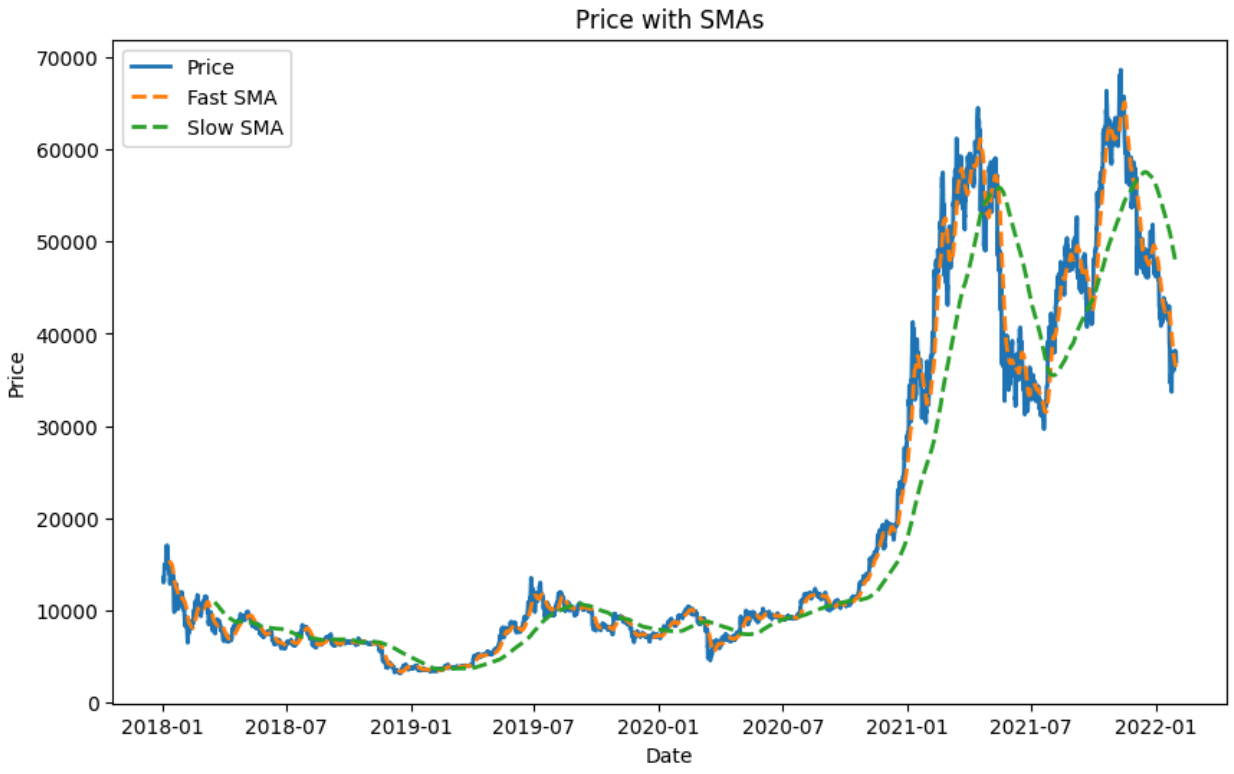
Team Name : 24KTJKHAA633347

Kharagpur Analytics Hackathon Report



Problem Statement

Algorithmic Trading Model for BTC/USDT Crypto Market, employing advanced quantitative techniques for optimized strategies and increased profitability. Addressing volatility challenges, the model prioritizes risk management and adaptability, redefining crypto trading through innovation.Algorithmic Trading Model for BTC/USDT Crypto Market, employing advanced quantitative techniques for optimized strategies and increased profitability. Addressing volatility challenges, the model prioritizes risk management and adaptability, redefining crypto trading through innovation.

Solution Outcomes

Optimizing BTC/USDT Trading for Profitability:

The model harnesses data-driven insights and predictive analytics to elevate trading performance in the BTC/USDT market. Through the thorough analysis of historical and real-time market data, it empowers traders to make informed decisions, foreseeing potential price movements and trends. This strategic utilization of data aims to minimize risks and capitalize on opportunities, ultimately driving increased profitability for users engaging in BTC/USDT trading with the assistance of the model.

Mitigating Human Error through Automation:

The implementation of automation significantly reduces the influence of emotional decision-making, a prevalent challenge in trading. By executing predefined strategies, the algorithm operates without being swayed by fear or greed, effectively minimizing the potential for human error. This automated approach enhances the reliability of trading decisions, contributing to a more disciplined and rational execution of strategies.

Ensuring Cutting-Edge Performance through Continuous Improvement:

Our algorithm undergoes regular updates and refinements, guaranteeing its continuous alignment with the latest technological advancements and adaptability to emerging market trends. This commitment to ongoing improvement ensures that the

algorithm remains at the forefront of its capabilities, consistently evolving to meet the dynamic demands of the market.

Anticipated Outcomes: Elevating Trading Excellence through Minimized Errors and Continuous Refinement

The expected results encompass heightened trading performance, a reduction in human error through automation, and the ongoing refinement of the model. These outcomes collectively ensure that the algorithm stays ahead in adapting to the ever-evolving dynamics of the market. By prioritizing precision, minimizing emotional influences, and embracing a commitment to continuous improvement, the model aims to provide a robust and effective tool for navigating the complexities of trading in dynamic markets.

Model Design

I. Mean Reversion Component:

- **Conceptual Framework:**
    - The model incorporates a mean reversion strategy, aiming to capitalize on the tendency of asset prices to revert to their historical average. This involves identifying periods of overvaluation or undervaluation and making trading decisions based on the expectation of a return to the mean.
- **Quantitative Techniques:**
    - Quantitative/statistical methods are employed to assess the statistical significance of deviations from the mean. This involves calculating measures such as z-scores or standard deviations to determine the likelihood of a reversion to the mean.

II. Machine Learning Integration:

- **Algorithmic Decision-Making:**
    - The model leverages machine learning algorithms for decision-making. This involves training the model on historical data to identify patterns and relationships, enabling it to make predictions and optimize trading strategies.
- **Feature Selection:**
    - Key features relevant to BTC/USDT trading, such as price movements, trading volumes, and market sentiment indicators, are selected for inclusion in the machine learning model. Feature selection is a crucial step to enhance the model's predictive accuracy.
- **Training and Testing:**
    - Historical data is divided into training and testing sets for machine learning model validation. This iterative process allows the model to learn from past market behavior and validate its predictions against unseen data.

III. Quantitative/Statistical Approaches:

- **Statistical Analysis:**
    - The model employs rigorous quantitative analysis, including statistical methods, to interpret market data. This may involve the use of statistical tests to validate hypotheses and assess the significance of observed patterns.

- **Risk Management:**
    - Quantitative approaches are applied to manage risk effectively. This includes calculating risk metrics, such as Value at Risk (VaR) or Conditional Value at Risk (CVaR), to ensure that the model operates within predefined risk tolerance levels.

IV. Simple Moving Average (SMA):

- **SMA as a Trend Indicator:**
    - The Simple Moving Average is used as a trend-following indicator. It smoothens price data over a specified period, aiding in identifying trends and potential turning points.
- **Backtrader for Back-Testing:**
    - Backtrader is utilized as the back-testing engine for the model. It enables the simulation of trading strategies on historical data, allowing for performance evaluation and refinement before live deployment.

V. SMA Cross Strategy:

- **Strategic Implementation:**
    - The SMA Cross Strategy is integrated into the model as a key signal generator. Buy and sell signals are triggered when short-term SMAs cross above or below long-term SMAs, providing a systematic approach to capturing potential market trends.
- **Back-Testing Validation:**
    - Backtrader is employed to validate the effectiveness of the SMA Cross Strategy. This involves simulating trades over historical data, assessing performance metrics, and refining the strategy for optimal results.

This comprehensive model design encompasses mean reversion principles, machine learning integration, quantitative/statistical approaches, and the specific use of SMA and SMA Cross Strategy, with Backtrader facilitating thorough back-testing for strategy validation.

Code Explaination

```
fast_sma_period = 35
slow_sma_period = 300
```

`fast_sma_period = 35`: This line defines the period for the fast Simple Moving Average (SMA) used in the strategy, specifying a window of 35 time units for smoothing the price data.

`slow_sma_period = 300`: This line sets the period for the slow Simple Moving Average (SMA), determining a more extended window of 300 time units for smoothing, representing a longer-term trend.

```
!pip install backtrader
```

`!pip install backtrader`: This line installs the backtrader library using the pip package manager, providing the necessary framework for developing and executing the backtesting strategy.

```
import backtrader as bt
import backtrader.analyzers as btanalyzers
```

`import backtrader as bt`: This line imports the backtrader library and aliases it as `bt` to facilitate a more concise reference within the code.

`import backtrader.analyzers as btanalyzers`: This line imports the analyzers module from the backtrader library, enabling the utilization of various analytical tools to evaluate the performance of the trading strategy.

```
class SMACrossStrategy(bt.Strategy):
    """ Live strategy demonstration with SMA """
    params = (
        ("fast_sma_period", 50),
        ("slow_sma_period", 200),
        ('timeframe', ''),
    )

    def _init_(self):
        self.fast_sma = bt.indicators.SimpleMovingAverage(self.data.close,
period=self.params.fast_sma_period)
        self.slow_sma = bt.indicators.SimpleMovingAverage(self.data.close,
period=self.params.slow_sma_period)

    def next(self):
        """The arrival of a new ticker bar"""
        for data in self.datas:  # We run through all the requested bars of all tickers
            ticker = data._name
            _interval = self.p.timeframe
            _date = bt.num2date(data.datetime[0])
            print('{} / {} [{}] - Open: {}, High: {}, Low: {}, Close: {}, Volume:
{}'.format(bt.num2date(data.datetime[0]), data._name, _interval, data.open[0], data.high[0],
data.low[0], data.close[0], data.volume[0], ))

            # Check for crossover
            if self.fast_sma > self.slow_sma and self.position.size == 0:
                free_money = self.broker.getcash()
                price = data.close[0]  # by close price
                size = (free_money / price) * 0.1  # 10% of free money
                self.buy(data=data, exectype=bt.Order.Limit, price=price, size=size)

            # Check for crossunder
            elif self.fast_sma < self.slow_sma and self.position.size > 0:
                self.close()
```

This code delineates the formulation of a backtesting strategy utilizing the backtrader library in the Python programming language. Termed as the "SMACrossStrategy," this strategy serves as an illustrative embodiment of a trend-following approach predicated on Simple Moving Averages (SMAs). The ensuing elucidation provides a systematic breakdown of the code's components and functionalities.

1. Parameters Definition:

- The strategy has three parameters specified under params:

  - fast_sma_period: Period for the fast Simple Moving Average.

  - slow_sma_period: Period for the slow Simple Moving Average.

  - timeframe: A parameter for specifying the timeframe (not used in the strategy logic).

2. Initialization:

- The __init__ method initializes the strategy. It creates two SMAs, one with a shorter period (fast_sma) and another with a longer period (slow_sma), based on the closing prices of the data.

3. next Method:

- The next method is called on each new bar/tick in the data.

- It iterates through all data feeds (tickers) and prints information about the current bar.

- The strategy checks for SMA crossovers and crossunders to generate buy/sell signals and executes orders accordingly.

```
    def notify_order(self, order):
        """Changing the status of the order"""
        order_data_name = order.data._name  # The name of the ticker from the order
        print("*" * 50)
        self.log(f'Order number {order.ref} {order.info["order_number"]}
{order.getstatusname()} {"Buy" if order.isbuy() else "Sell"} {order_data_name} {order.size} @
{order.price}')
        if order.status == bt.Order.Completed:  # If the order is fully executed
            if order.isbuy():  # if the order is buy
                self.log(f'Buy {order_data_name} Price: {order.executed.price:.2f}, Val:
{order.executed.value:.2f}, Comm: {order.executed.comm:.2f}')
            else:  # if the order is sell
                self.log(f'Sell {order_data_name} Price: {order.executed.price:.2f}, Val:
{order.executed.value:.2f}, Comm: {order.executed.comm:.2f}')
        print("*" * 50)

    def notify_trade(self, trade):
        """Changing the position status"""
        if trade.isclosed:  # If the position is closed
            self.log(f'Profit on a closed position {trade.getdataname()} Total=
{trade.pnl:.2f}, Comm={trade.pnlcomm:.2f}')

    def log(self, txt, dt=None):
        """Output a date string to the console"""
        dt = bt.num2date(self.datas[0].datetime[0]) if not dt else dt  # The set date or the
date of the current bar
        print(f'{dt.strftime("%d.%m.%Y %H:%M")}, {txt}')  # Output the date and time with the
specified text to the console
```

4. Order Handling:

- The notify_order method is called whenever the status of an order changes (submitted, completed, etc.).

- It prints information about the order, including order number, status, type (buy/sell), ticker name, size, and price.

- Upon order completion, it prints additional details such as executed price, value, and commission.

5. Trade Handling:

- The notify_trade method is called when the position status changes (opened or closed).

- It prints information about the profit/loss on a closed position, including the ticker name, total profit/loss, and commission.

6. Logging:

- The log method is a utility function for printing messages to the console, including a timestamp.

Risk Management Evaluation:

1. Position Sizing:

- Implement dynamic position sizing methods to adjust the size of trades based on market conditions and volatility.

2. Stop-Loss and Take-Profit Orders:

- Introduce stop-loss and take-profit orders to automatically manage risk and secure profits at predefined levels.

3. Dynamic Parameter Adjustments:

- Consider adapting the fast and slow SMA periods dynamically to respond to changing market volatility.

4. Maximum Drawdown Limits:

- Set maximum drawdown limits to control overall risk exposure and trigger a reevaluation of the strategy during significant downturns.

5. Regular Monitoring and Review:

- Continuously monitor the strategy's performance, periodically review risk management parameters, and adapt to evolving market conditions.

BACKTESTING USING CEREBRO

1. cerebro = bt.Cerebro():

-Creates a Cerebro engine, which is the core of the backtesting framework in backtrader.

2. cerebro.broker.set_cash(100000):

- Sets the initial cash amount in the broker account to $100,000.

3. cerebro.broker.setcommission(commission=0.0015):

- Configures the broker commission for buying or selling assets.

-In this case, the commission is set to 0.15% (0.0015) of the transaction value.

4. data = bt.feeds.PandasData(dataname=df, name=ticker):

-Creates a PandasData object, which is a backtrader data feed specifically designed to handle Pandas Data Frames.

-dataname=df: Specifies the Pandas DataFrame containing historical price data (df).

-name=ticker: Assigns a name to the data feed, typically representing the ticker symbol of the asset.

5. cerebro.adddata(data):

-Adds the created data feed (data) to the cerebro backtesting engine.

-Multiple data feeds can be added if you want to test a strategy on multiple assets simultaneously.

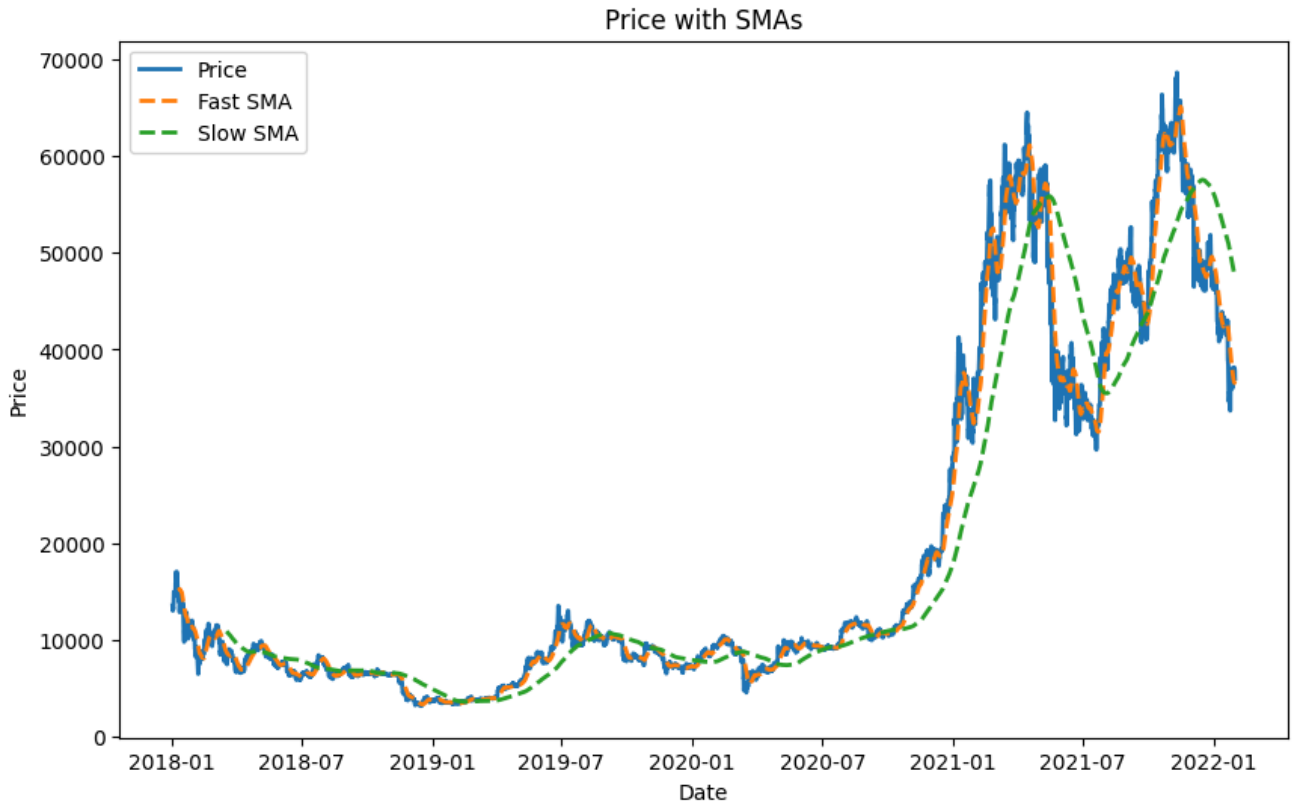
8. cerebro.broker.getvalue():

- This retrieves the current value of the portfolio in the backtesting environment.

9. print(f"Starting Portfolio Value: {cerebro.broker.getvalue():.2f}"):

- It then prints the starting portfolio value to the console using an f-string.

- ".2f" is used to format the value with two decimal places.



JUSTIFICATION FOR USING DATA MODEL/INDICATOR

1. Trend Identification:

- SMAs are commonly used to identify trends in financial markets. The strategy in the code leverages the relationship between a shorter-term SMA (fast) and a longer-term SMA (slow) to identify potential trends in the price movement.

2. Crossover Signals:

- The strategy generates buy signals when the fast SMA crosses above the slow SMA.This is often interpreted as a bullish signal, indicating a potential upward trend.

3. Crossunder Signals:

- Conversely, sell signals are generated when the fast SMA crosses below the slowSMA, signaling a potential reversal or the beginning of a downward trend.

4. Backtesting:

- The code is designed for backtesting, allowing the evaluation of the SMA-based strategy's historical performance on price data. This is essential for assessing its viability before deploying it in a live trading environment.

5. Customization:

- The code provides parameters for adjusting the periods of the fast and slow SMAs,allowing users to customize the strategy based on their preferences or market conditions.

RESULTS OF ROI/DRAWDOWNS/SHARPE RATIO

- Initial Investment = 100000

- Ending Portfolio Value = 152542.54

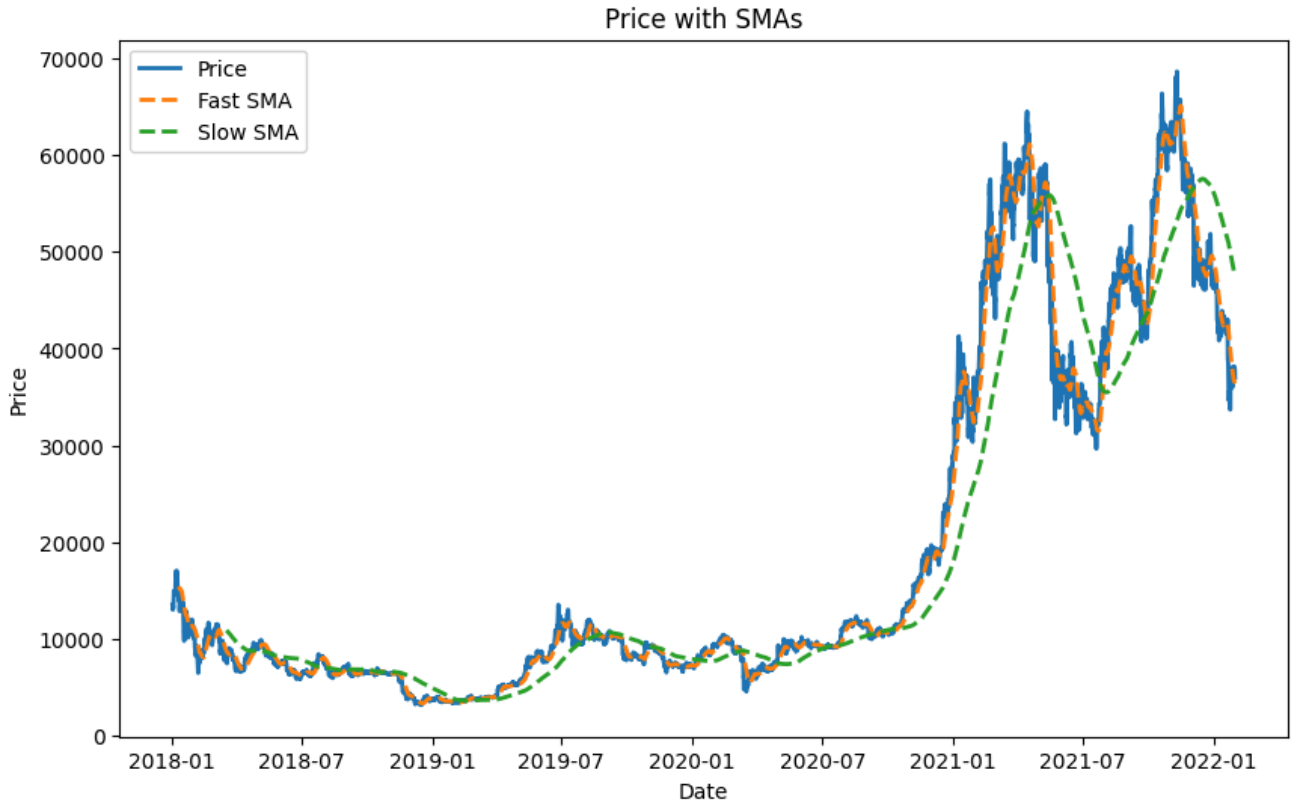
- Remaining Available Funds = 152542.54

- Max Drawdowns = 9.9312

- ROI= 52.54%

- DrawDowns=0.177%

- Sharpe Ratio= 1:2


Price with SMAs
Price
Fast SMA
Slow SMA
70000
60000
50000
40000
30000
20000
10000
0
Price
2018-01
2018-07
2019-01
2019-07
2020-01
2020-07
2021-01
2021-07
2022-01
Date

References:

https://chat.openai.com

https://in.tradingview.com

https://www.binance.com

https://www.backtrader.com