Building a Mean reverting strategy on provided data

by Aditya Paradkar

**Custom Functions and their properties:**

**- select\_ticker:**

The `select\_ticker` function is part of a class and serves the purpose of filtering market data for a specific financial instrument (referred to as "ticker") within a specified time range. Here's what the function does:

1. \*\*Set Index\*\*: It first sets the index of the DataFrame `data` to the "CreatedOn" column. This allows for more efficient data manipulation and filtering based on timestamps.

2. \*\*Time Range Slicing\*\*: The function then performs a time range slicing on the DataFrame using the `start` and `end` attributes of the class instance. This effectively selects only the rows within the specified time range.

3. \*\*Reset Index\*\*: After slicing, the function resets the index of the DataFrame to the default numeric index using the `reset\_index` method with the `inplace=True` argument.

4. \*\*Instrument Identifier Filtering\*\*: The DataFrame is then filtered based on the "InstrumentIdentifier" column. It retains only the rows where the "InstrumentIdentifier" matches the value of the class attribute `ticker`.

5. \*\*Return Filtered Data\*\*: Finally, the filtered DataFrame containing data for the specific financial instrument ("ticker") within the specified time range is returned.

In summary, this function allows you to extract and isolate the data associated with a particular financial instrument (ticker) within a defined time period from a larger dataset. This is useful for further analysis, processing, and backtesting purposes.

**- clean\_and\_resample:**

The `clean\_and\_resample` function processes and resamples the input DataFrame containing market data. It performs the following tasks:

1. \*\*Date and Time Extraction\*\*: It adds two new columns to the DataFrame: "Date" and "Time." These columns are populated by extracting the date and time components from the "CreatedOn" column using lambda functions.

2. \*\*Time-based Filtering\*\*: It iterates through the DataFrame rows and removes rows that have a "Time" outside the trading hours of 9:15 AM to 3:30 PM. This filtering removes data points outside of regular trading hours.

3. \*\*Drop Temporary Columns\*\*: The "Date" and "Time" columns are dropped from the DataFrame since they were used for filtering and are no longer needed.

4. \*\*Index Reset\*\*: The index of the DataFrame is set to "CreatedOn," which is the original timestamp column.

5. \*\*OHLC Resampling\*\*: An OHLC (Open, High, Low, Close) resampling is performed on the DataFrame using the specified resampling interval (self.interval). The resampling aggregates the data within each interval and calculates the first open price, maximum high price, minimum low price, and last close price.

6. \*\*Reset Index Again\*\*: After resampling, the index is reset to the default numeric index.

Finally, the function returns the resampled DataFrame, which now contains aggregated OHLC data based on the specified resampling interval.

Overall, this function is responsible for cleaning the data by removing unwanted time periods and then resampling the data into the desired intervals, making it more suitable for analysis and visualization in the context of the mean-reverting strategy backtesting.

**- BB:**

The `BB` function appears to implement the calculation of Bollinger Bands for a given DataFrame containing market data. Here's a breakdown of what this function does:

1. \*\*Bollinger Bands Calculation\*\*: The function calculates Bollinger Bands using the `ta.bbands` function from the `ta-lib` library. Bollinger Bands are a technical indicator that consists of three lines: the upper band, the middle band (also called the moving average), and the lower band. The `ta.bbands` function calculates these bands based on the specified input values.

2. \*\*Bollinger Bands Naming\*\*: The function defines the names for the three bands: "Lower\_BB," "Mid\_BB," and "Upper\_BB."

3. \*\*Bollinger Bands DataFrame\*\*: The calculated Bollinger Bands are stored in the variable `bb`. Only the first three columns (related to the Bollinger Bands) are selected using the `iloc` indexer with `[:,:3]`. This selection includes the lower, middle, and upper bands.

4. \*\*Column Renaming\*\*: The column names of the selected Bollinger Bands are updated to the names specified earlier.

5. \*\*Concatenation\*\*: The calculated Bollinger Bands are concatenated horizontally (axis 1) with the original DataFrame `data` using the `pd.concat` function. This adds the Bollinger Bands data as new columns to the DataFrame.

6. \*\*Data Cleaning\*\*: Any rows containing NaN (missing) values are removed using the `dropna` method. This step is performed to ensure that only valid data points are used for analysis.

7. \*\*Return Data\*\*: The cleaned DataFrame, which now includes the calculated Bollinger Bands, is returned from the function.

In summary, this function calculates Bollinger Bands for the "CloseValue" column of the input DataFrame and adds the lower, middle, and upper bands to the DataFrame. The cleaned DataFrame with Bollinger Bands data is then returned for further analysis and use within the context of the mean-reverting strategy backtesting.

**- signal\_generation:**

The `signal\_generation` function generates trading signals based on the provided DataFrame `df` containing market data and the calculated Bollinger Bands. It evaluates the market conditions and generates signals for potential trade entries and exits. Here's a detailed breakdown of what this function does:

1. \*\*Index Reset\*\*: The function first resets the index of the DataFrame `df` to have a new numeric index starting from 0. This allows for easier iteration through the DataFrame.

2. \*\*Variable Initialization\*\*: Several variables are initialized, including `position`, `sell\_stats`, `buy\_stats`, and two new columns "Sell\_Signal" and "Buy\_Signal" are added to the DataFrame to store signal information.

3. \*\*Signal Generation Loop\*\*: The function iterates through each row (timestamp) in the DataFrame using the index `i`.

- \*\*Entry Signal Generation\*\*: If the algorithm is not in a position (`position` is False), and the closing price is greater than the upper Bollinger Band, and the current row is not the last row in the DataFrame, an entry signal is generated. This means a potential short position is considered. The entry signal details (timestamp, opening price) are appended to the `sell\_stats` list. Stop-loss and target-profit values are calculated based on the opening price.

- \*\*Position Management\*\*: If in a position (`position` is True), several conditions are checked to determine whether to exit the position or not:

- If the high price exceeds the stop-loss value, an exit signal is generated, and the position is closed. The exit details are added to the `buy\_stats` list.

- If the low price goes below the target-profit value, a target-hit signal is generated, and the position is closed.

- If the closing price goes below the lower Bollinger Band, an exit signal is generated, and the position is closed.

- If the current row is the last row in the DataFrame, an exit signal is generated for the last bar.

- \*\*Signal Indication\*\*: The appropriate signal columns ("Sell\_Signal" or "Buy\_Signal") in the DataFrame are updated to indicate the presence of a signal (1) when an entry or exit signal is generated.

4. \*\*Return Signals and DataFrame\*\*: The function returns three values: the `sell\_stats` list (containing entry signals), the `buy\_stats` list (containing exit signals), and the modified DataFrame `df` with the added signal information.

In summary, the `signal\_generation` function analyzes the market conditions, generates signals for potential trades, and records these signals along with relevant details in the `sell\_stats` and `buy\_stats` lists. It also updates the DataFrame with signal information for visualization and analysis purposes.

**- metrics:**

The `metrics` function is responsible for calculating and presenting various trading metrics based on the trade signals generated during the mean-reverting strategy backtesting. Here's a breakdown of what this function does:

1. \*\*Variables Initialization\*\*: Several variables are initialized to keep track of different aspects of the trading signals, including `time\_in\_trade`, `points`, `stop`, `target`, `exit\_signal`, `entry\_signal`, and `last\_bar`.

2. \*\*Counting Signal Types\*\*: Two loops iterate through the `sell\_stats` and `buy\_stats` lists to count the occurrence of different types of trade signals. The number of entry signals, exit signals, target-hit signals, stop-hit signals, and last bar exit signals are counted and stored in their respective variables.

3. \*\*Trade Durations and Points Calculation\*\*: Another loop pairs the entry and exit signals to calculate the time duration for each trade (`time\_in\_trade`) and the points earned or lost for each trade (`points`). The points are calculated as the difference between the entry price and the exit price, multiplied by the trading quantity.

4. \*\*Metric Calculation and DataFrame Creation\*\*: Various trading metrics are calculated based on the collected data. These metrics include the total number of trades, the number of winning trades, the number of losing trades, the maximum loss, average gain, average loss, total profit and loss (PnL), average time in trade, and counts of different types of exit signals. These metrics are stored in a Pandas DataFrame with appropriate index labels.

5. \*\*Data Presentation\*\*: The calculated metrics are displayed in tabular format using the Pandas DataFrame. Additionally, a line plot is created to visualize the cumulative points (PnL) over the course of the trades.

In summary, the `metrics` function takes the generated trade signals as input, calculates various trading metrics based on these signals, and presents the results in both tabular and graphical forms. It provides insights into the performance of the mean-reverting trading strategy during the backtesting period.

- After generating the signals, I have used the backtesting module in python to generate trade data and after building a function, have used it to get the following metrics.

**- graph:**

The `graph` function is responsible for creating a candlestick chart that visualizes the market data along with the generated trading signals, specifically buy and sell signals. Here's a breakdown of what this function does:

1. \*\*DataFrame Preparation\*\*: The function sets the "CreatedOn" column as the index of the input DataFrame `df`. It also renames the columns of `df` to standard names for the open, high, low, and close prices, as well as the Bollinger Bands.

2. \*\*Data Slicing\*\*: Two new DataFrames are created from `df`:

- `df1` contains only the columns related to open, high, low, and close prices.

- `bb` contains only the columns related to the Bollinger Bands.

3. \*\*Addplot Configuration\*\*: The function configures additional plot elements (`addplot`) to overlay on the candlestick chart. These elements include:

- Bollinger Bands represented by the lines.

- Buy signals represented by green triangles above the close prices where `Buy\_Signal` is 1.

- Sell signals represented by red triangles below the close prices where `Sell\_Signal` is 1.

4. \*\*Plotting\*\*: The candlestick chart is plotted using the `mpf.plot` function. The `df1` DataFrame provides the candlestick data, and the `addplot` parameter includes the additional elements configured in the `apdict` list.

- `type='candle'` specifies that a candlestick chart should be plotted.

- `figscale=1.6` adjusts the figure scale for better visualization.

In summary, the `graph` function visualizes the market data using a candlestick chart and overlays Bollinger Bands as well as buy and sell signals on the chart. This visualization helps traders and analysts to better understand the strategy's performance and behavior in relation to the price movements and signal generation points.

I have written the code in iPython notebook format. It is recommended that the code be opened in Jupyter notebook or VS code for the best visualizations and for better visibility and cleaner formatting.

It is highly recommended that the reader refer to the code for a better and indepth understanding of the strategy.

I have taken the help of ChatGPT to generate this readme file.

The average time taken to run the code is around 33-35 seconds out of which 28+ seconds is used to read the data.