Python for Finance: Mean Reversion Algorithm

Algorithmic trading programs for financial applications are exceptionally powerful tools and can be created and implemented utilizing the Python programming language. In this report, the framework for a simple mean reversion algorithmic strategy will be detailed and discussed, followed by a more complex modified mean reversion algorithm that will be designed and deployed using the Quantiopian.com virtual environment.

I was originally tasked to create an algorithm that employed the purchasing and short-selling of the S&P 500 based upon the % change of returns from 6-months in the past in relation to current prices. This simple strategy is seeking to capitalize upon probable changes in the market using historical inferences, that the S&P 500 will return to a sort of average trend by short-selling when prices are relatively high and purchasing when oversold or when returns have been underperforming the historical market average.

This strategy in the simplest form, chooses a past date 6-months from the present and calculates the % change of return and then acts upon that data accordingly, also known as a mean reversion strategy. There are more comprehensive mean reversion strategies that I believe are a better indicator of market trends and should outperform the initial strategy significantly by increasing the complexity and number of inputs; therefore, I have decided to design and create a more complex mean reversion algorithm. However, I will first describe the complete process and provide sample code in which to perform the original algorithmic strategy before I describe the more detailed mean reversion algorithm.

By downloading 'daily' S&P 500 data from Yahoo Finance, we begin by reading and summarizing the data and then calculate daily returns. Next, the algorithm will be designed to buy and hold equities for 6-months if the average returns are less than 15% over a 6-month period, or (short) sell and then buy after 6-months if returns are greater than 20% over a 6-month period.

- Original Parameters:
  - Download 'daily' S&P 500 data from Yahoo Finance
  - Read S&P 500 .csv data
  - Summarize data *(completed with the .describe() function)*
  - Compute daily returns *(completed with the pct_change() function)*
- Example Constraints:
  - If returns are < 15%, then buy and hold for 6 months
  - If returns are > 20%, then sell and buy after 6 months

In order to complete this algorithm, we must first gather the data from Yahoo Finance. Once this step is complete, we begin our importation process. First, we import the libraries/ packages used [**pandas** *(for data manipulation)*, **NumPy** *(for enhanced mathematical operations)*, **matplotlib** (*for data visualization),* and **backtrader** *(for backtesting)*] and then load the S&P 500 data into the python script. This can be accomplished through multiple methods; however, for this case we will use a pandas dataframe (**df**) to import the file to better manipulate the information. The code I used to accomplish this task is below:

```
import matplotlib.pyplot as plt

from matplotlib import style

import numpy as np

import pandas as pd

import backtrader as bt

df = pd.read_csv(r'C:\Users\dynamo\Desktop\AI - ML - Python Master\S&P
Algo\data\^GSPC.csv', index_col=1, parse_dates=True).sort_index()
```

Once the information is uploaded into the Python script, the data can be analyzed and manipulated accordingly using the example code below:

```
df['pct_change'] = df.Close.pct_change()

print(df)

print(df.describe())
```

This code allows for the viewing of the opening price, close price, adjusted close, daily high, volume, mean, standard deviation, and daily percentage change. From here, simple code that defines the window length of historical time-series data that we want to analyze *(6-months)* will be created using the pandas dataframe. Next, functions that looks back to the beginning of the defined time-series data and compares the percentage change in price would be implemented. The mathematical logic could be implemented utilizing an if (*if returns are < 15%, then buy and hold for 6 months*) and elif (*if returns are > 20%, then sell and buy after 6 months*) statements to determine if the security should be purchased, sold or no action taken if the constraints are not met. If these constraints are met, then a function to order or sell a target number of shares will be written. Then a backtesting function using backtester (or another package) can be used to track its performance in the form of retuns. However, a scheduling function (the frequency in which it should run) and other inputs should be determined and implemented for the algorithm to run properly, this would include the starting equity and the use of leverage if desired.

When undergoing my research for the design of this portion of the algorithm, I found the Quantopian.com virtual environment that is very powerful for the development and deployment of financial algorithms and began to discover more complex strategies and see how they work. It was at this point that I was able to design some alternative algorithms and backtest them with relative ease due to the rich Quantopian environment. This allowed me to quickly change inputs and modify designs to enhance returns. After running various simplistic algorithms and seeing the inferior (and often negative returns,) I decided that a more complex algo would be justified in order to enhance potential returns and further advance my knowledge and skillset during this process.

What is Quantopian and why should we use it? From Quantopian.com:

> Quantopian is a crowd-sourced quantitative investment firm. We inspire talented people from around the world to write investment algorithms. Quantopian provides capital, education, data, a research environment, and a development platform to algorithm authors (quants). We offer license agreements for algorithms that fit our investment strategy, and the licensing authors are paid based on their strategy's individual performance. We provide everything a quant needs to create a strategy and profit from it.

Quantopian is a platform that provides access to free online deployable environment, free data sources, has outstanding visualization, backtesting, real-time testing, built-in benchmarking, built-in functions to streamline development, quick algorithmic customization, ability to have algorithm funded, excellent educational resources, tutorials, and extensive (clear) documentation to use it. Basically, its an online environment that facilitates the development and implementation of financial algorithms beginning from the most rudimentary concepts to some of the most advanced, creating an all-in-one solution available for free.

Financial algorithms are heavily reliant on the data sources they utilize, Quantopian includes various free data sources that can be used for backtesting and real-time trading with both paper money and personal brokerage account connectivity for actual trading. From Quantipian.com: "for paper trading and real-money trading, we get a real-time feed of trades from Nanex's NxCore product. Those trades are bundled into one-minute bars and fed to the trading algorithms. Paper trading data is provided on a 15-minute delay. Real-money trading is processed without delay." With Yahoo Finance and Google API access for free financial information deprecated, Quantopian's free data sources prove an invaluable asset for algorithmic development.

Now that the Quantopian environment has been described, I will breakdown the design, purpose, components, and performance of a more complex mean reversion algorithm deployed within the Quantopian virtual environment.

What is mean reversion? From Investopedia.com:

> Mean reversion is financial theory suggesting that asset prices and returns eventually return back to the long-run mean or average of the entire dataset. This mean or average can be the historical average of the price or return, or another relevant average such as the growth in the economy or the average return of an industry.

This mean reversion algorithm is designed to rank stocks from the "QTradableStocksUS" data-feed (supplied from Quantopian) by the top and bottom 5% performing securities over a 10-day period, then purchase underperforming stocks and sell overperforming stocks in anticipation of a mean reversion. The purpose of this algorithm is to generate maximum returns by optimizing the algorithm and modifying inputs, and then backtesting to calculate changes in performance.

The **components of this algorithm** within the Quantopian environment are as follows:

Imports

- Quantopian for pipeline
- Packages
- Specialized Packages
- Datasets (QTradableStocksUS)

Related code to be deployed:

```
import quantopian.algorithm as algo

import quantopian.optimize as opt

from quantopian.pipeline import Pipeline

from quantopian.pipeline.data.builtin import USEquityPricing

from quantopian.pipeline.factors import Returns

from quantopian.pipeline.filters import QTradableStocksUS
```

Inputs

- Equities: universe = QTradableStocksUS()
- Weighting: MAX_POSITION_CONCENTRATION = 0.01 [1% maximum investment in any given security]
- Leverage (2 times): MAX_GROSS_EXPOSURE = 2.0
- Time / Lookback period: RETURNS_LOOKBACK_DAYS = 10

Initialize

- Attach Pipeline
- Scheduling (trades, data recording, rebalancing, etc.)

Define Pipeline (Factors)

- Asset Universe
- Basic Data Sets
- Create any Built-in Factors (Ex. Simple-Moving Average)
- Create any Custom-Factors
- 'return' Pipeline (Define the columns and any screen which we want our pipeline to return, this becomes the data that our algorithm will use to make trading decisions.)

Run Pipeline (before trading starts)

- Run pipeline_output to get the latest data for each security.
- The data is returned in a 2D pandas dataframe. Rows are the security objects. Columns are what was defined in the pipeline definition.
- Placed in the context object so it's available to other functions and methods (quasi global.)

Define Trade

- This is a scheduled function to execute all buys and sells.
- Here is where you can filter, sort, and do whatever you want with that data.
- Anything that could have been done in pipeline can be done with the dataframe that it returns.

Record Trading Data & Logging

- Record the number of positions held each day.

- Plot variables at the end of each day.
- Add logging of intended orders (timing, symbols and order target size).
- This helps us monitor that the algo is behaving correctly and was able to achieve its target positions.

Why Pipeline? Many trading algorithms have the following structure: (from Quantopian.com)

1. For each asset in a known (large) set, compute N scalar values for the asset based on a trailing window of data. This kind of calculation is called a **cross-sectional trailing-window** computation. A simple example of a cross-sectional trailing-window computation is "close-to-close daily returns", which has the form: *Every day, fetch the last two days of close prices for all assets. For each asset, calculate the percent change between the asset's previous close price and its current close price.*
2. Select a smaller tradeable set of assets based on the values computed in (1).
3. Calculate desired portfolio weights on the set of assets selected in (2).
4. Place orders to move the algorithm's current portfolio allocations to the desired weights computed in (3).

There are several technical challenges with doing this robustly. These include:

- efficiently querying large sets of assets
- performing computations on large sets of assets
- handling adjustments (splits and dividends)
- asset delistings

Pipeline exists to solve these challenges by providing a uniform API for expressing computations on a diverse collection of datasets and to make it easy to define and execute cross-sectional trailing-window computations.

Now that the design, purpose and components have been described, I will discuss the inputs used and the overall performance of the algorithm.

After some trial-and-error, I found that a 10-day lookback window provided the most optimal results, as I was generating negative returns in almost all other scenarios. I believe this is due to the recency in which it is implemented, as it seems to provide a reasonable measure of the short-term trend. I chose to employ the use of leverage for the algorithm to magnify potential gains. Due to the risk profile increasing with leverage, some other constraints should be put in place to mitigate some level of risk. So, in this case, I chose to weight the portfolio to have only a 1% maximum position in any one security at a given time, and to rebalance the portfolio every day 90 minutes after the market opens. Also, I input a dollar-neutral strategy to mitigate risk as well, and I implemented a z-score normalization of the returns to better rank the stocks. Ultimately, I chose to use only the top and bottom 5% performing stocks, as these probably have the greatest potential for a mean reversion, modifying it to this input drastically changed my results. Finally, I used a built-in function "MaximizeAlpha" to seek securities with the highest alpha rating and to optimize the portfolio accordingly while complying with the weighting and dollar-neutral constraints. **Overall, this algorithm over the last 12 months has generated returns in excess of 23% while the S&P 500 generated only 8.42%.**

Full Mean Reversion algorithmic code for deployment on the Quantopian.com virtual environment

```
import quantopian.algorithm as algo

import quantopian.optimize as opt

from quantopian.pipeline import Pipeline

from quantopian.pipeline.data.builtin import USEquityPricing

from quantopian.pipeline.factors import Returns

from quantopian.pipeline.filters import QTradableStocksUS


MAX_GROSS_EXPOSURE = 2.0


MAX_POSITION_CONCENTRATION = 0.01


RETURNS_LOOKBACK_DAYS = 10


def initialize(context):


    algo.schedule_function(

        rebalance,

        algo.date_rules.week_start(days_offset=0),
```

```
        algo.time_rules.market_open(hours=1, minutes=30)
    )

    algo.attach_pipeline(make_pipeline(context), 'mean_reversion')

def make_pipeline(context):

    universe = QTradableStocksUS()

    recent_returns = Returns(
        window_length=RETURNS_LOOKBACK_DAYS,
        mask=universe
    )

    recent_returns_zscore = recent_returns.zscore()

    low_returns = recent_returns_zscore.percentile_between(0,5)
    high_returns = recent_returns_zscore.percentile_between(95,100)

    securities_to_trade = (low_returns | high_returns)

    pipe = Pipeline(
```

```
        columns={
            'recent_returns_zscore': recent_returns_zscore
        },
        screen=securities_to_trade
    )

    return pipe

def before_trading_start(context, data):

    context.output = algo.pipeline_output('mean_reversion')

    context.recent_returns_zscore = context.output['recent_returns_zscore']


def rebalance(context, data):

    objective = opt.MaximizeAlpha(-context.recent_returns_zscore)

    max_gross_exposure = opt.MaxGrossExposure(MAX_GROSS_EXPOSURE)

    max_position_concentration = opt.PositionConcentration.with_equal_bounds(
```

```
    -MAX_POSITION_CONCENTRATION,
    MAX_POSITION_CONCENTRATION
)

dollar_neutral = opt.DollarNeutral()

constraints = [
    max_gross_exposure,
    max_position_concentration,
    dollar_neutral,
]

algo.order_optimal_portfolio(objective, constraints)
```