ALGORITHMIC TRADING USING PYTHON

Introduction :

Algorithmic trading (also known as algo trading or automated trading) refers to the use of computer programs and advanced mathematical models to execute trades in financial markets. Instead of relying on human decision-making, algo trading relies on pre-programmed instructions to analyze market data and make trading decisions.

The algorithms used in algo trading can be designed to perform a wide range of tasks, such as identifying patterns in market data, executing trades based on specific criteria, and managing risk. By automating the trading process, algo trading can reduce the time and effort required for traders to monitor markets and execute trades manually.

Method :

In this project we put whole algorithmic trading project into practice, from back testing the strategy to carrying out automatic, real-time trading. The project's main components are listed below:

- Strategy: I opted for a time series momentum strategy, which essentially assumes that an asset class that has executed satisfactorily or poorly will keep on doing so (cf. Moskowitz, Tobias, Yao Hua Ooi, and Lasse Heje Pedersen (2012): "
- Trading framework: I went with Oanda because you can trade different contracts for variations (CFDs), which effectively let you place directed bets on a range of investments (including currencies, stock indices, and commodities).
- Data: Oanda will provide us with all of our historical and real-time data..
- **Software**: We use Python in combination with the other libraries and packages.

Process :

- To access the Oanda API programmatically, you need to install the relevant Python package

```
pip install oandapy
```

- Create

```
[oanda]
account_id = YOUR_ACCOUNT_ID
access_token = YOU_ACCESS_TOKEN
```

- Replace

```
In [1]:
import configparser  # 1
import oandapy as opy  # 2

config = configparser.ConfigParser()  # 3
config.read('oanda.cfg')  # 4

oanda = opy.API(environment='practice',
                access_token=config['oanda']['access_token'])  # 5
```

Back testing:

Everything required to begin the back testing of the momentum strategy has already been set up. Retrieving the data and converting it to a pandas Data Frame object is the first step in the back testing process.

```
In [2]:
import pandas as pd  # 6

data = oanda.get_history(instrument='EUR_USD',  # our instrument
                         start='2016-12-08',  # start data
                         end='2016-12-10',  # end date
                         granularity='M1')  # minute bars  # 7

df = pd.DataFrame(data['candles']).set_index('time')  # 8

df.index = pd.DatetimeIndex(df.index)  # 9

df.info() # 10
```

```
<class 'pandas.core.frame.DataFrame'>
DatetimeIndex: 2658 entries, 2016-12-08 00:00:00 to 2016-12-09 21:59:00
Data columns (total 10 columns):
closeAsk    2658 non-null float64
closeBid    2658 non-null float64
complete    2658 non-null bool
highAsk     2658 non-null float64
highBid     2658 non-null float64
lowAsk      2658 non-null float64
lowBid      2658 non-null float64
openAsk     2658 non-null float64
openBid     2658 non-null float64
volume      2658 non-null int64
dtypes: bool(1), float64(8), int64(1)
memory usage: 210.3 KB
```

Secondly, we formalize the momentum strategy by instructing Python to compute the placement in the device using the mean log yield over the previous 45, 90, and 150 min bands.

```
In [3]:
import numpy as np  # 11

df['returns'] = np.log(df['closeAsk'] / df['closeAsk'].shift(1))  # 12

cols = []  # 13

for momentum in [15, 30, 60, 120]:  # 14
    col = 'position_%s' % momentum  # 15
    df[col] = np.sign(df['returns'].rolling(momentum).mean())  # 16
    cols.append(col)  # 17
```

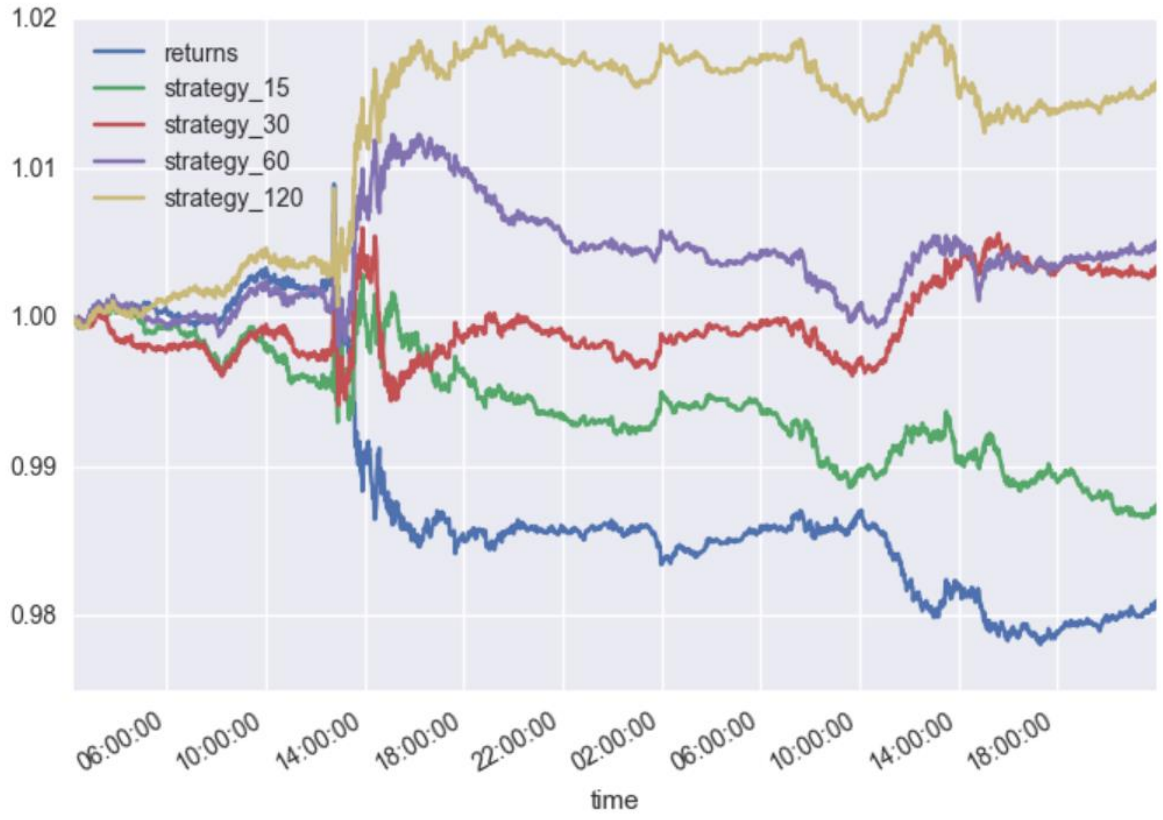
```
In [4]:
%matplotlib inline
import seaborn as sns; sns.set()  # 18

strats = ['returns']  # 19

for col in cols:  # 20
    strat = 'strategy_%s' % col.split('_')[1]  # 21
    df[strat] = df[col].shift(1) * df['returns']  # 22
    strats.append(strat)  # 23

df[strats].dropna().cumsum().apply(np.exp).plot()  # 24
```

```
Out[4]:
<matplotlib.axes._subplots.AxesSubplot at 0x11a9c6a20>
```



Automated Trading:

```
In [5]:
class MomentumTrader(opy.Streamer):  # 25
    def __init__(self, momentum, *args, **kwargs):  # 26
        opy.Streamer.__init__(self, *args, **kwargs)  # 27
        self.ticks = 0  # 28
        self.position = 0  # 29
        self.df = pd.DataFrame()  # 30
        self.momentum = momentum  # 31
        self.units = 100000  # 32
    def create_order(self, side, units):  # 33
        order = oanda.create_order(config['oanda']['account_id'],
            instrument='EUR_USD', units=units, side=side,
            type='market')  # 34
        print('\n', order)  # 35
    def on_success(self, data):  # 36
        self.ticks += 1  # 37
        # print(self.ticks, end=', ')
        # appends the new tick data to the DataFrame object
        self.df = self.df.append(pd.DataFrame(data['tick'],
                                 index=[data['tick']['time']]))  # 38
        # transforms the time information to a DatetimeIndex object
        self.df.index = pd.DatetimeIndex(self.df['time'])  # 39
        # resamples the data set to a new, homogeneous interval
        dfr = self.df.resample('5s').last()  # 40
        # calculates the log returns
        dfr['returns'] = np.log(dfr['ask'] / dfr['ask'].shift(1))  # 41
        # derives the positioning according to the momentum strategy
        dfr['position'] = np.sign(dfr['returns'].rolling(
                                        self.momentum).mean())  # 42
        if dfr['position'].ix[-1] == 1:  # 43
            # go long
            if self.position == 0:  # 44
                self.create_order('buy', self.units)  # 45
```

```
        elif self.position == -1:  # 46
            self.create_order('buy', self.units * 2)  # 47
        self.position = 1  # 48
    elif dfr['position'].ix[-1] == -1:  # 49
        # go short
        if self.position == 0:  # 50
            self.create_order('sell', self.units)  # 51
        elif self.position == 1: # 52
            self.create_order('sell', self.units * 2)  # 53
        self.position = -1  # 54
    if self.ticks == 250:  # 55
        # close out the position
        if self.position == 1:  # 56
            self.create_order('sell', self.units)  # 57
        elif self.position == -1:  # 58
            self.create_order('buy', self.units)  # 59
        self.disconnect()  # 60
```

Conclusion:

Backtesting a momentum strategy, automating trading based on a momentum strategy definition, and other project phases like obtaining data for backtesting, are all detailed in the paper. The code provided offers a place to start when exploring a variety of options, like trading with alternate instruments, utilizing alternative algorithmic trading algorithms, etc.

To sum up, algorithmic trading has grown in importance as a tool for investors and traders looking to maximize their performance in the financial markets. Algo trading uses sophisticated mathematical models and computer algorithms to execute trades more quickly, with less emotion involved in decision-making, and with better risk management. A thorough grasp of the underlying market dynamics and diligent testing and monitoring of traders' algorithms are essential for ensuring the efficacy of algo trading. Additionally, it's critical for regulators to keep up with innovations in algorithmic trading and to make sure that the right protections are in place for investors.

References:

- Seth, S. (2023, April 5). *Basics of Algorithmic Trading: Concepts and examples*. Investopedia. Retrieved April 20, 2023, from https://www.investopedia.com/articles/active-trading/101014/basics-algorithmic-trading-concepts-and-examples.asp
- Hilpisch, Y. (2017, January 18). *Algorithmic trading in less than 100 lines of Python Code*. O'Reilly Media. Retrieved April 20, 2023, from https://www.oreilly.com/content/algorithmic-trading-in-less-than-100-lines-of-python-code/