EXCELLENCE. KNOWLEDGE. EXTENSIVE.

Pair Trading

Statistical Arbitrage on Cash Stocks

By QuantInsti™

Asia's pioneer Algorithmic Trading Research and Training Institute

About the Author

Jonathan Moreno Narváez
Master of Science in Chemical Engineering, Universidad de los Andes, Bogotá, Colombia

Jonathan has a strong knowledge of mathematical programming and has worked as a process optimization engineer for 3 years. He started to get involved in trading as a hobby, especially in algorithmic trading due to his passion for math but eventually, it became his full-time job. Jonathan enrolled for Executive Programme in Algorithmic Trading (EPAT™) in November 2016 and found his space in the world on quantitative analysis in finance. Currently, he is taking several courses online in subjects related to Artificial Intelligence and its applications in finance and is about to start an online portal in Financial Engineering to share his experience as a Quant Trader.

Contents

About the Author ... 2

Pair Trading – Statistical Arbitrage on Cash Stocks ... 4

Strategy ... 4

Code Details and In-Sample Backtesting ... 4

Analyzing Model Output ... 10

Chemicals ... 13

Financials ... 14

Entertainment ... 15

Consumer Goods ... 16

Industrial Goods ... 17

Monte Carlo Analysis ... 18

Chemicals ... 20

Financials ... 21

Entertainment ... 22

Consumer Goods ... 23

Industrial Goods ... 24

All Portfolios ... 25

Conclusions ... 29

Future work ... 29

References ... 29

Annex ... 29

Backtest.py ... 31

Data.py ... 35

Event.py ... 41

Execution.py ... 45

Performance.py ... 48

Plot_performance_JOMN.py ... 53

Portfolio.py ... 55

Strategy.py ... 62

Contact Us ... 64

Pair Trading – Statistical Arbitrage on Cash Stocks

Objective

The objective of this project is to model a Statistical Arbitrage trading strategy and quantitatively analyze the modelling results. Motivation relies on diversifying investment throughout five sectors, aka: Technology, Financial, Entertainment, Consumer Goods and Industrial Goods. Within these sectors, stocks were selected according with their move in tandem because prices are affected by same market events. However, noise might make them temporally deviate from the usual pattern and a trader could take advantage of this apparent deviation with the expectation that the stocks will eventually return to their long-term relationship.

Within each sector, stocks were selected based on high liquidity, small bid/ask spread and ability to short the stock. Once the stock universe was defined, pairs were formed. Every day as we want to enter a position, all the pairs in the universe were evaluated and the top pairs are selected per some criteria.

	Sectors/Stocks	1	2	3	4	5
1	Chemicals	DOW	PX	AASH	CE	HUN
2	Financials	PNC	JPM	BAC	C	HSBC
3	Entertainment	FOX	FOXA	DIS	TWX	CMCSA
4	Consumer Goods	CLX	CL	PG	PEP	COKE
5	Industrial Goods	FLS	HON	ROK	CR	BA

Strategy

As the universe of pairs was already defined, the logic of the strategy is: if any pair ratio (ratio of closing prices of stock pair) diverges from a certain threshold, we short the stock that is expensive and buy the one that is cheap. Once they converge to the mean, we close the position and profit from the reversal.

The strategy triggers new orders when the pair ratio diverge from its mean. To ensure the convenience of trading at this point, pair must be cointegrated. If the pair ratio is cointegrated, the ratio is mean revering and the greater the dispersion from its mean, the higher the probability of a reversal, which makes the trade more attractive. This analysis allows to determine the stability of the long-term relationship. Spread time series is tested for stationarity by the augmented Dickey Fuller (ADF) test. In other words, if pair stocks are cointegrated, it suggests that the mean and variance of this correlation remains content over time. There is, however, a major issue which make this simple strategy difficult to implement in practice: long term relationship can break down; spread can move from one equilibrium to another.

Code Details and In-Sample Backtesting

The code is a modification of code provided by Michael L. Halls-Moore as part of his book *Successful Algorithmic Trading*. For reference about the backtesting system implemented on python, go to Part VI from this reference for further explanation.

Backtesting was done for each pair for the period of 1-Jan-2009 to 31-Dec-2014.

First at all, proper packages are imported:

```
#!/usr/bin/python
# -*- coding: utf-8 -*-
# Pair_Trading_Mean_Reverting.py

from __future__ import print_function

import datetime

import numpy as np
import pandas as pd
import statsmodels.api as sm
import statsmodels.tsa.stattools as ts
import matplotlib.pyplot as plt
import itertools

from strategy import Strategy
from event import SignalEvent
from backtest import Backtest
from data import HistoricCSVDataHandler
from portfolio import Portfolio
from execution import SimulatedExecutionHandler
from plot_performance_JOMN import plot_performance
```

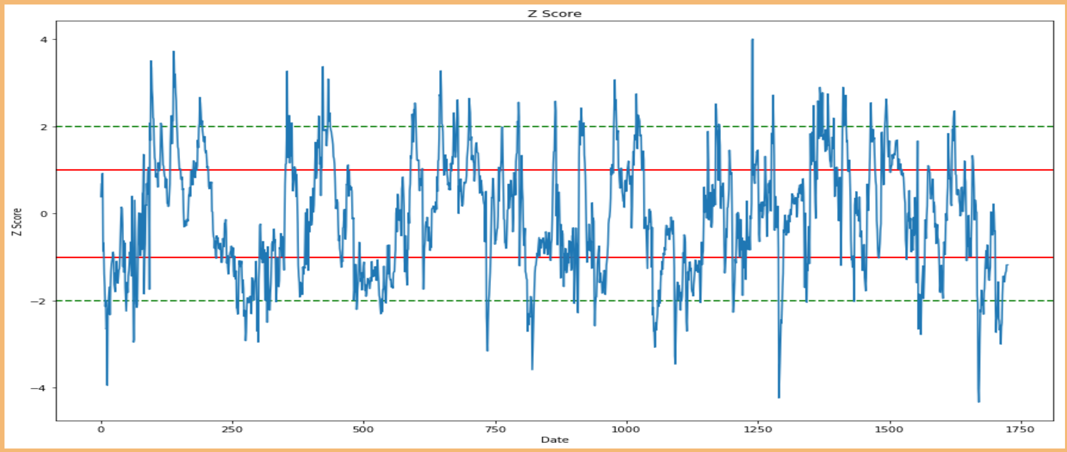
Second, an indicator was built, aka z-score, for every pair. This indicator is the number of standard deviations that pair ratio has diverge from its mean:

$$z_{i,j} = \frac{R_{i,j} - \mu_{i,j}}{\sigma_{i,j}}$$

Where $R_{i,j}$ is the price ratio of stock i and j, $\mu_{i,j}$ is the mean of the ratio and $\sigma_{i,j}$ is the standard deviation of same ratio.

Once Z-score is greater or lower a certain threshold, the fist condition required for sending an order is fulfilled. Then, a rolling Augmented Dickey Fuller test is calculated. From this calculation, the p-value is extracted and if it is less than the alpha, it means that the price ratio series is stationary and the second condition is met. In other words, every day while backtesting, following conditions must be checked:

- Entry and Exit criteria are satisfied. Z-score is above or below dot-green lines for entry and inside red lines for close positions.
- While any trade is initiated or previous trade continued, that pair must satisfy ADF test daily.



The strategy was coded with Object Oriented Programming approach. Z-score was calculated using the following parameters:

- Moving average and Standard deviation of price ratio: 40 days
- ADF test window: 40 days
- Z-score Low: 1.0
- Z-score Low: 2.0
- Initial Capital: $USD 100,000

```
class IntradayOLSMRStrategy(Strategy):
    """
    Uses ordinary least squares (OLS) to perform a rolling linear
    regression to determine the hedge ratio between a pair of equities.
    The z-score of the residuals time series is then calculated in a
    rolling fashion and if it exceeds an interval of thresholds
    (defaulting to [1.0, 2.0]) then a long/short signal pair are generated
    (for the high threshold) or an exit signal pair are generated (for the
    low threshold).
    """

    def __init__(
        self, bars, events, pair, ols_window=40,
        zscore_low=1, zscore_high=2
    ):
        """
        Initialises the stat arb strategy.

        Parameters:
        bars - The DataHandler object that provides bar information
        events - The Event Queue object.
        """
        self.bars = bars
        self.symbol_list = self.bars.symbol_list
        self.events = events
        self.ols_window = ols_window
        self.zscore_low = zscore_low
        self.zscore_high = zscore_high

        self.pair = pair
```

```
        self.datetime = datetime.datetime.utcnow()

        self.long_market = False
        self.short_market = False

    def calculate_xy_signals(self, zscore_last,cadf):
        """
        Calculates the actual x, y signal pairings
        to be sent to the signal generator.

        Parameters
        zscore_last - The current zscore to test against
        """
        y_signal = None
        x_signal = None
        p0 = self.pair[0]
        p1 = self.pair[1]
        dt = self.datetime
        hr = abs(self.hedge_ratio)
        global cadf_array

        cadf_array = cadf_array.append(pd.DataFrame([cadf],columns=['p-
value']),ignore_index=True)

        # If we're long the market and below the
        # negative of the high zscore threshold
        if zscore_last <= -self.zscore_high and not self.long_market and cadf<=0.05:
            self.long_market = True
            y_signal = SignalEvent(1, p0, dt, 'LONG', 1.0)
            x_signal = SignalEvent(1, p1, dt, 'SHORT', hr)

        # If we're long the market and between the
#         absolute value of the low zscore threshold
        if (abs(zscore_last) <= self.zscore_low and self.long_market and cadf<=0.05) or
cadf>0.05:
            self.long_market = False
            y_signal = SignalEvent(1, p0, dt, 'EXIT', 1.0)
            x_signal = SignalEvent(1, p1, dt, 'EXIT', 1.0)

        # If we're short the market and above
        # the high zscore threshold
        if zscore_last >= self.zscore_high and not self.short_market and cadf<=0.05:
            self.short_market = True
            y_signal = SignalEvent(1, p0, dt, 'SHORT', 1.0)
            x_signal = SignalEvent(1, p1, dt, 'LONG', hr)

        # If we're short the market and between the
        # absolute value of the low zscore threshold
        if (abs(zscore_last) <= self.zscore_low and self.short_market and cadf<=0.05) or
cadf>0.05:
            self.short_market = False
            y_signal = SignalEvent(1, p0, dt, 'EXIT', 1.0)
            x_signal = SignalEvent(1, p1, dt, 'EXIT', 1.0)
```

```
        return y_signal, x_signal
    def calculate_signals_for_pairs(self):
        """
        Generates a new set of signals based on the mean reversion
        strategy.

        Calculates the hedge ratio between the pair of tickers.
        We use OLS for this, althought we should ideall use CADF.
        """
        global zcore_last_array

        # Obtain the latest window of values for each
        # component of the pair of tickers
        y = self.bars.get_latest_bars_values(
            self.pair[0], "adj_close", N=self.ols_window
        )
        x = self.bars.get_latest_bars_values(
            self.pair[1], "adj_close", N=self.ols_window
        )

        if y is not None and x is not None:
            # Check that all window periods are available
            if len(y) >= self.ols_window and len(x) >= self.ols_window:
                # Calculate the current hedge ratio using  OLS
                self.hedge_ratio = sm.OLS(y, x).fit().params[0]

                # Calculate the current z-score of the residuals
                spread = y - self.hedge_ratio * x
                zscore_last = ((spread - spread.mean())/spread.std())[-1]

                zcore_last_array =
zcore_last_array.append(pd.DataFrame([zscore_last],columns=['Z Score']),ignore_index=True)

                z_score = ((spread - spread.mean())/spread.std())

                cadf = ts.adfuller(z_score)

                y_signal, x_signal = self.calculate_xy_signals(zscore_last,cadf[1])
                if y_signal is not None and x_signal is not None:
                    self.events.put(y_signal)
                    self.events.put(x_signal)

    def calculate_signals(self, event):
        """
        Calculate the SignalEvents based on market data.
        """
        if event.type == 'MARKET':
            self.calculate_signals_for_pairs()


def get_symb_pairs(symbol_list):
    '''self.symbol_list is a list stocks symbols
```

```
        This function takes in a list of symbols and
        returns a list of unique pairs of symbols
    '''
    symbPairs = []

    symbList = symbol_list

    for subset in itertools.combinations(symbList, 2):
        symbPairs.append(subset)

    return symbPairs

if __name__ == "__main__":
    csv_dir = 'C:/Users/Jonathan/OneDrive/Algo Trading/Cursos/Quantinsti/Final
Project/Project/data'  # CHANGE THIS!
    symbol_list_all = [['DOW','PX','ASH','CE','HUN'],
                       ['PNC','JPM','BAC','C','HSBC'],
                       ['FOX','FOXA','DIS','TWX','CMCSA'],
                       ['CLX','CL','PG','PEP','COKE'],
                       ['FLS','HON','ROK','CR','BA']]
    initial_capital = 100000.0
    zcore_last_array = pd.DataFrame([(float('NaN'))],columns=['Z Score'])
    cadf_array = pd.DataFrame([(float('NaN'))],columns=['p-value'])
    heartbeat = 0.0
    start_date = datetime.datetime(2008, 1, 1, 0, 0, 0)

    rows = len(symbol_list_all)

    for j in range(0,rows):
        symbol_list = symbol_list_all[j]
        PAIRS = get_symb_pairs(symbol_list)
        Pairs_Len = len(PAIRS)
        for i in range(0,Pairs_Len):
            pair = PAIRS[i]
            print(pair)

            backtest = Backtest(csv_dir, symbol_list, initial_capital, heartbeat,
                start_date, HistoricCSVDataHandler, SimulatedExecutionHandler,
                Portfolio, IntradayOLSMRStrategy,pair)

            csv_file = "equity_"+pair[0]+"_"+pair[1]+".csv"

            backtest.simulate_trading(csv_file)
            print(csv_file)
            plot_performance(csv_file)
```

Analyzing Model Output

Once the strategy was backtested for the 50 pairs (10 pairs formed for each sector), results were consolidated per sector as states the following code. Without losing generality, equally weighted returns for each pair within each sector portfolio was assumed.

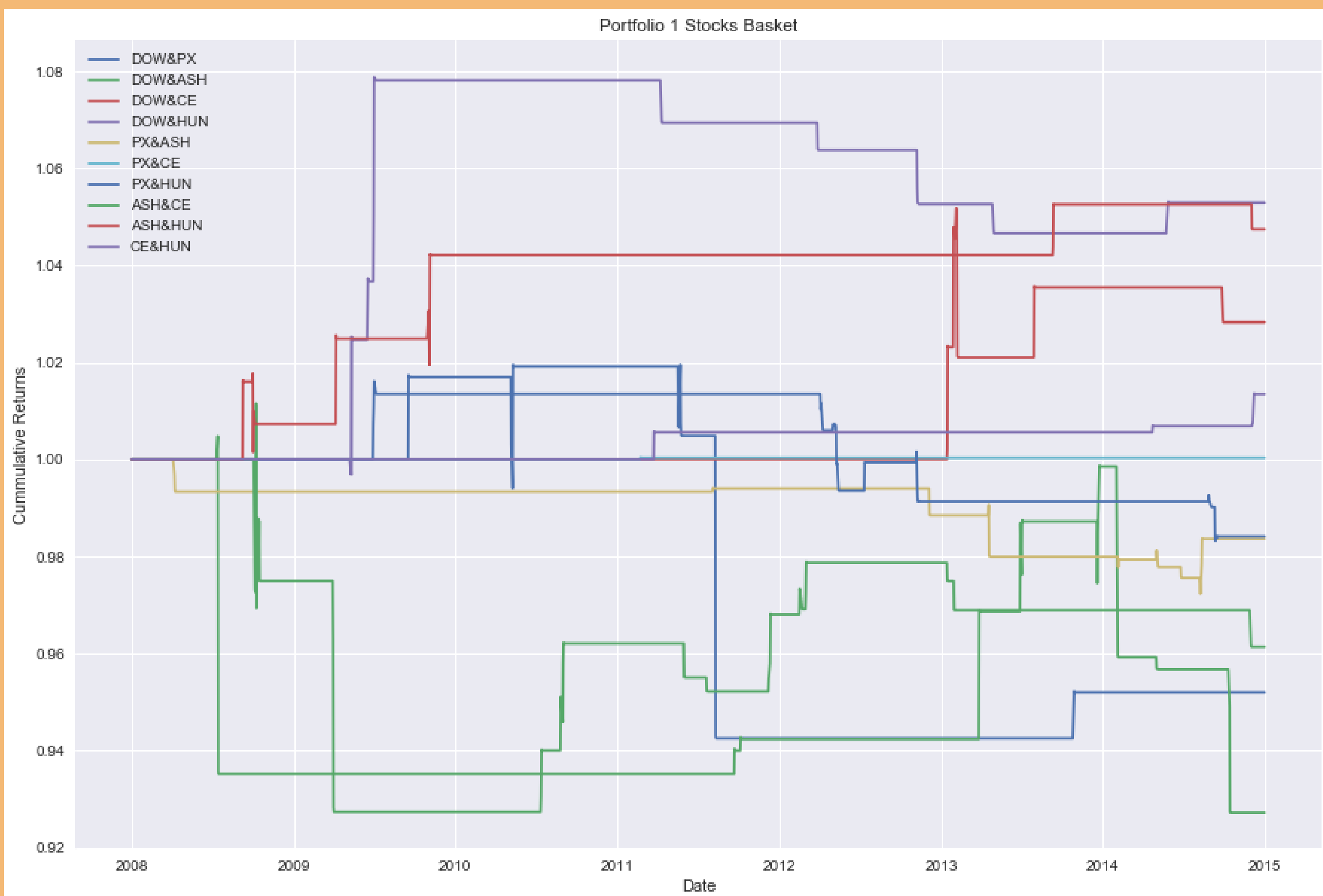
```
# -*- coding: utf-8 -*-
"""
Created on Thu May 04 10:35:15 2017

@author: Jonathan Moreno Narváez
"""
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sbn
import itertools

#%% Statistics
def Stats(returns,pnl):
    #Total Return
    total_return=pnl[-1]
    print("Total Return", "%0.2f%%" % ((total_return - 1.0) * 100.0))

    #CAGR

    days = (pnl.index[-1]-pnl.index[1]).days
    CAGR = ((pnl[-1]/pnl[1])**(252.0/days))-1.0
    print("CAGR","%0.2f%%" % (CAGR*100.0))
    # Standard deviation
    Std = np.std(returns)
    print("Standard Deviation Ret", "%0.2f%%" % (Std*100.0))
    # Sharpe Ratio
    Sh_R = np.sqrt(252) * (np.mean(returns)) / np.std(returns)
    print("Sharpe Ratio", "%0.2f" % Sh_R)
    # Maximum Drawdown
    # Calculate the cumulative returns curve
    # and set up the High Water Mark
    hwm = [0]

    # Create the drawdown and duration series
    idx = pnl.index
    drawdown = pd.Series(index = idx)
    duration = pd.Series(index = idx)

    # Loop over the index range
    for t in range(1, len(idx)):
        hwm.append(max(hwm[t-1], pnl[t]))
```

```
        drawdown[t]= (hwm[t]-pnl[t])
        duration[t]= (0 if drawdown[t] == 0 else duration[t-1]+1)
    #    drawdown, drawdown.max(), duration.max()
    Data_Cum['drawdown'] = drawdown
    print("Max Drawdown", "%0.2f%%" % (drawdown.max() * 100.0))
    print("Drawdown Duration", "%d days" % duration.max())

#%% Several Stocks

#list of stocks in portfolio
symbol_list_all = [['DOW','PX','ASH','CE','HUN'],
                   ['PNC','JPM','BAC','C','HSBC'],
                   ['FOX','FOXA','DIS','TWX','CMCSA'],
                   ['CLX','CL','PG','PEP','COKE'],
                   ['FLS','HON','ROK','CR','BA']]

def get_symb_pairs(symbol_list):
    '''self.symbol_list is a list stocks symbols
       This function takes in a list of symbols and
       returns a list of unique pairs of symbols
    '''
    symbPairs = []

    symbList = symbol_list

    for subset in itertools.combinations(symbList, 2):
        symbPairs.append(subset)

    return symbPairs

#read returns from csv files of pairs-stocks in the portfolio
rows = len(symbol_list_all)

csv_dir = 'C:/Users/Jonathan/OneDrive/Algo Trading/Cursos/Quantinsti/Final
Project/Project/results'  # CHANGE THIS!

weights_MC = np.zeros((4+10-1,rows))

for j in range(0,rows):
    Data = pd.DataFrame()
    symbol_list = symbol_list_all[j]
    PAIRS = get_symb_pairs(symbol_list)
    Pairs_Len = len(PAIRS)
    for i in range(0,Pairs_Len):
        pair = PAIRS[i]
        print(pair)
        pair_stk =
pd.read_csv(csv_dir+'/equity_'+str(pair[0])+'_'+str(pair[1])+'.csv',index_col=0)
        #daily returns
        Data[str(pair[0])+'_'+str(pair[1])] = pair_stk['returns']

    Data.index = pd.to_datetime(Data.index)
```

```
    Data.dropna().to_csv(csv_dir+'/Returns_'+str(j+1)+'.csv')

    #calculate mean daily return and covariance of daily returns
    mean_daily_returns = Data.dropna().mean()
    cov_matrix = Data.dropna().cov()

    #set array holding portfolio weights of each stock
    weights = np.asarray([0.1,0.1,0.1,0.1,0.1,0.1,0.1,0.1,0.1,0.1])

    #calculate annualised portfolio return
    portfolio_return = round(np.sum(mean_daily_returns * weights) * 252,4)
    #calculate annualised portfolio volatility
    portfolio_std_dev = round(np.sqrt(np.dot(weights.T,np.dot(cov_matrix, weights))) *
np.sqrt(252),4)
    print('Portfolio expected annualised return is {}% and volatility is
{}%').format(portfolio_return*100,portfolio_std_dev*100)

    Data = Data.dropna()

    Data['ret_pot'] = (Data * weights).sum(axis=1)
    Data_Cum = (1.0+Data).cumprod()

    returns = Data['ret_pot']
    pnl = Data_Cum['ret_pot']

    Stats(returns,pnl)

    fig1 = plt.figure(figsize=(15,10))
    plt.plot(Data_Cum.index,Data_Cum[[0]],label=list(Data_Cum)[0])
    plt.plot(Data_Cum.index,Data_Cum[[1]],label=list(Data_Cum)[1])
    plt.plot(Data_Cum.index,Data_Cum[[2]],label=list(Data_Cum)[2])
    plt.plot(Data_Cum.index,Data_Cum[[3]],label=list(Data_Cum)[3])
    plt.plot(Data_Cum.index,Data_Cum[[4]],label=list(Data_Cum)[4])
    plt.plot(Data_Cum.index,Data_Cum[[5]],label=list(Data_Cum)[5])
    plt.plot(Data_Cum.index,Data_Cum[[6]],label=list(Data_Cum)[6])
    plt.plot(Data_Cum.index,Data_Cum[[7]],label=list(Data_Cum)[7])
    plt.plot(Data_Cum.index,Data_Cum[[8]],label=list(Data_Cum)[8])
    plt.plot(Data_Cum.index,Data_Cum[[9]],label=list(Data_Cum)[9])
    plt.plot(Data_Cum.index,Data_Cum[[10]],label=list(Data_Cum)[10],lw=3,color='k')
    plt.xlabel('Date')
    plt.ylabel('Cummulative Returns')
    plt.title('Portfolio '+str(j+1)+' Stocks Basket')
    plt.legend(loc=2)
    plt.show()
```

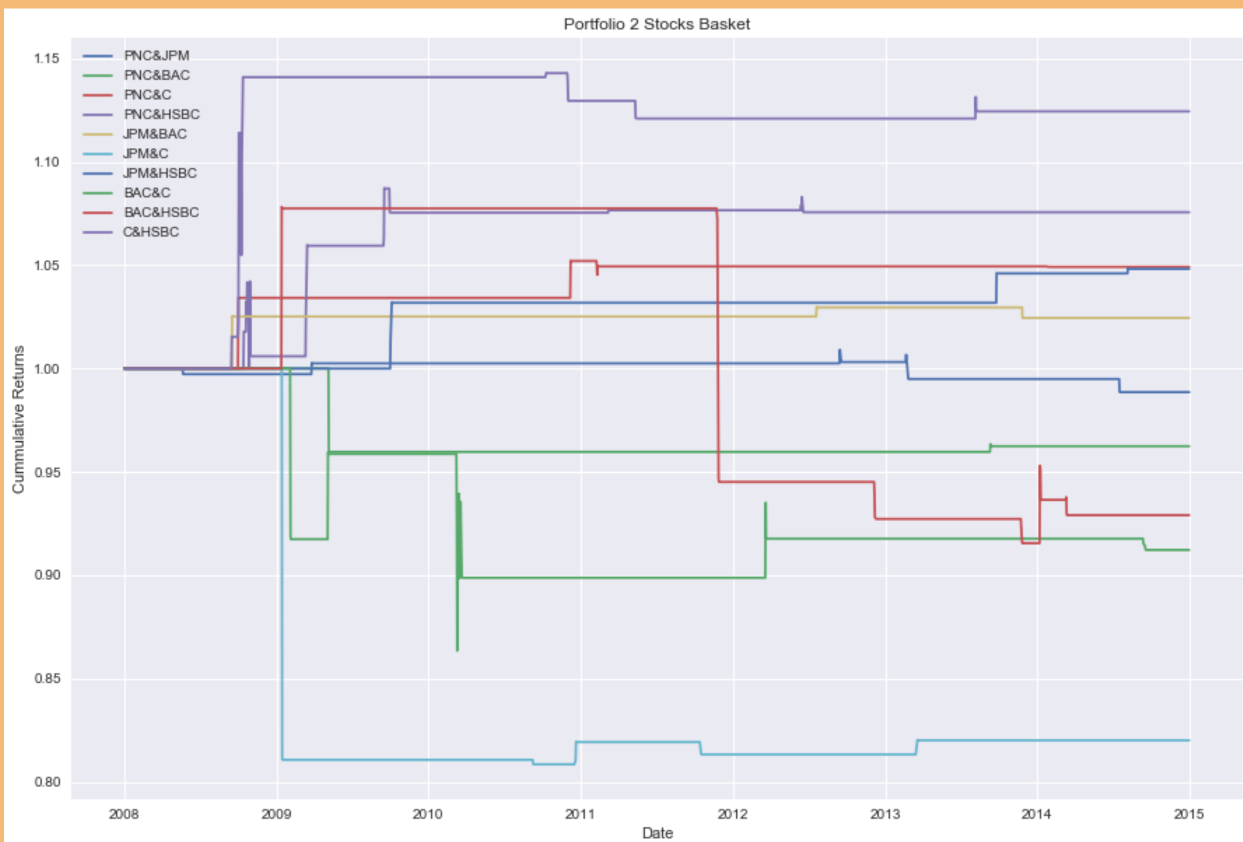
Chemicals

	DOW_PX	DOW_ASH	DOW_CE	DOW_HUN	PX_ASH	PX_CE	PX_HUN	ASH_CE	ASH_HUN	CE_HUN
Total Return	-4.80%	-7.28%	2.83%	5.29%	-1.64%	0.03%	-1.58%	-3.86%	4.74%	1.35%
Sharpe Ratio	-0.24	-0.29	0.25	0.44	-0.37	0.29	-0.31	-0.16	0.44	0.6
Max Drawdown	7.71%	7.76%	3.07%	3.22%	2.76%	0.01%	3.28%	8.41%	1.62%	0.02%
Max Drawdown Lev 2	15.42%	15.52%	6.14%	6.44%	5.52%	0.02%	6.56%	16.82%	3.24%	0.04%
Max Drawdown Lev 3	23.13%	23.28%	9.21%	9.66%	8.28%	0.03%	9.84%	25.23%	4.86%	0.06%
Drawdown Duration (days)	1170	1630	482	1385	1699	971	1385	1568	965	772
CAGR	-0.48%	-0.74%	0.28%	0.51%	-0.16%	0.00%	-0.16%	-0.39%	0.46%	0.13%
Standard Deviation Ret	0.18%	0.22%	0.10%	0.11%	0.04%	0.00%	0.05%	0.20%	0.10%	0.02%
Success Ratio	60.00%	55.56%	60.00%	50.00%	25.00%	100.00%	40.00%	46.15%	66.67%	100.00%
Average Profit_Average Loss	0.38	0.48	1.9	2.79	0.35	inf	0.58	0.64	4.47	inf



Financials

	PNC_JPM	PNC_BAC	PNC_C	PNC_HSBC	JPM_BAC	JPM_C	JPM_HSBC	BAC_C	BAC_HSBC	C_HSBC
Total Return	-1.14%	-3.76%	4.92%	7.56%	2.45%	-17.99%	4.82%	-8.78%	-7.09%	12.44%
Sharpe Ratio	-0.28	-0.35	0.47	0.37	0.35	-0.35	0.67	-0.19	-0.16	0.4
Max Drawdown	2.05%	4.03%	0.68%	4.14%	0.54%	19.15%	0.04%	13.65%	16.27%	5.91%
Max Drawdown Lev 2	4.10%	8.06%	1.36%	8.28%	1.08%	38.30%	0.08%	27.30%	32.54%	11.82%
Max Drawdown Lev 3	6.15%	12.09%	2.04%	12.42%	1.62%	57.45%	0.12%	40.95%	48.81%	17.73%
Drawdown Duration (days)	874	1425	1023	1332	967	1502	999	1489	1502	1064
CAGR	-0.11%	-0.38%	0.47%	0.72%	0.24%	-1.94%	0.47%	-0.90%	-0.72%	1.16%
Standard Deviation Ret	0.04%	0.10%	0.09%	0.19%	0.06%	0.45%	0.06%	0.38%	0.36%	0.28%
Success Ratio	40.00%	50.00%	50.00%	62.50%	66.67%	40.00%	100.00%	33.33%	33.33%	71.43%
Average Profit_Average Loss	0.34	0.07	18.98	2.65	5.96	0.11	inf	0.44	0.62	7.33

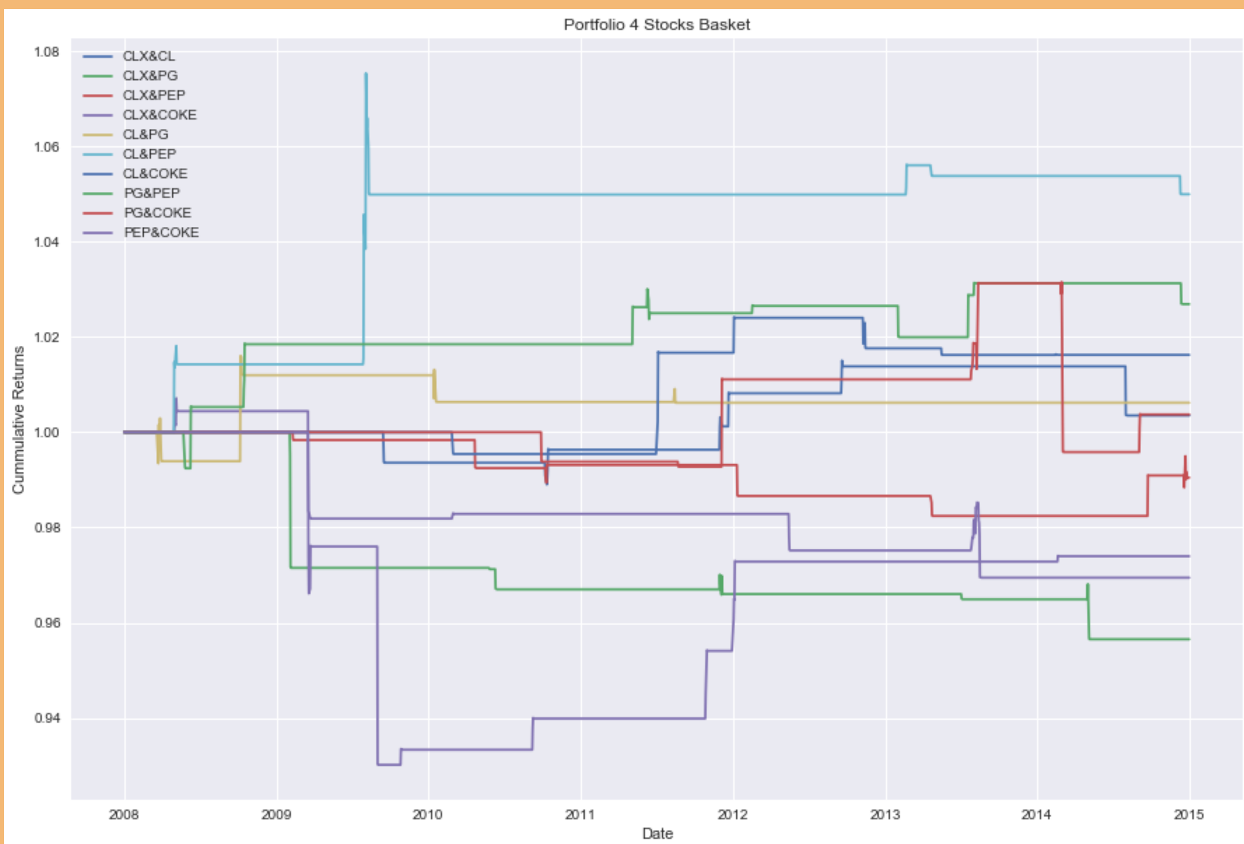


Entertainment

	FOX_FOXA	FOX_DIS	FOX_TWX	FOX_CMCSA	FOXA_DIS	FOXA_TWX	FOXA_CMCSA	DIS_TWX	DIS_CMCSA	TWX_CMCSA
Total Return	0.28%	0.13%	-1.09%	1.72%	17.48%	-6.52%	0.40%	0.66%	0.90%	0.42%
Sharpe Ratio	0.07	0.02	-0.15	0.16	0.54	-1.09	0.07	0.07	0.1	0.09
Max Drawdown	1.43%	5.57%	1.91%	3.66%	1.99%	6.63%	1.55%	4.33%	2.83%	1.49%
Max Drawdown Lev 2	2.86%	11.14%	3.82%	7.32%	3.98%	13.26%	3.10%	8.66%	5.66%	2.98%
Max Drawdown Lev 3	4.29%	16.71%	5.73%	10.98%	5.97%	19.89%	4.65%	12.99%	8.49%	4.47%
Drawdown Duration (days)	1550	1208	1277	1014	499	1207	1015	1148	1272	1235
CAGR	0.03%	0.01%	-0.11%	0.17%	1.60%	-0.66%	0.04%	0.07%	0.09%	0.04%
Standard Deviation Ret	0.04%	0.11%	0.06%	0.10%	0.28%	0.06%	0.05%	0.09%	0.08%	0.04%
Success Ratio	54.55%	50.00%	33.33%	75.00%	66.67%	0.00%	66.67%	53.85%	70.00%	83.33%
Average Profit_Average Loss	1.16	1.04	0.52	6.31	17.25	0	1.18	1.25	1.32	1.64

Consumer Goods

	CLX_CL	CLX_PG	CLX_PEP	CLX_COKE	CL_PG	CL_PEP	CL_COKE	PG_PEP	PG_COKE	PEP_COKE
Total Return	0.35%	-4.35%	-0.95%	-2.60%	0.62%	5.00%	1.62%	2.69%	0.37%	-3.06%
Sharpe Ratio	0.08	-0.59	-0.22	-0.15	0.09	0.37	0.31	0.44	0.04	-0.54
Max Drawdown	1.15%	4.35%	1.76%	7.70%	0.99%	2.55%	0.80%	1.01%	3.57%	3.06%
Max Drawdown Lev 2	2.30%	8.70%	3.52%	15.40%	1.98%	5.10%	1.60%	2.02%	7.14%	6.12%
Max Drawdown Lev 3	3.45%	13.05%	5.28%	23.10%	2.97%	7.65%	2.40%	3.03%	10.71%	9.18%
Drawdown Duration (days)	574	1490	1485	1677	1570	1363	753	641	409	1459
CAGR	0.03%	-0.44%	-0.09%	-0.26%	0.06%	0.48%	0.16%	0.26%	0.04%	-0.31%
Standard Deviation Ret	0.04%	0.07%	0.04%	0.14%	0.07%	0.12%	0.05%	0.06%	0.10%	0.05%
Success Ratio	66.67%	0.00%	28.57%	75.00%	20.00%	60.00%	50.00%	60.00%	57.14%	50.00%
Average Profit_Average Loss	1.22	0	0.49	0.67	1.41	9.57	2.34	2.38	1.11	0.27



Industrial Goods

	FLS_HON	FLS_ROK	FLS_CR	FLS_BA	HON_ROK	HON_CR	HON_BA	ROK_CR	ROK_BA	CR_BA
Total Return	0.09%	3.92%	1.41%	6.20%	3.37%	-0.99%	2.48%	-1.82%	-1.37%	-1.06%
Sharpe Ratio	0.02	0.55	0.13	0.46	0.36	-0.39	0.25	-0.47	-0.37	-0.16
Max Drawdown	1.91%	0.37%	2.64%	2.44%	2.30%	1.34%	2.85%	1.82%	1.84%	2.00%
Max Drawdown Lev 2	3.82%	0.74%	5.28%	4.88%	4.60%	2.68%	5.70%	3.64%	3.68%	4.00%
Max Drawdown Lev 3	5.73%	1.11%	7.92%	7.32%	6.90%	4.02%	8.55%	5.46%	5.52%	6.00%
Drawdown Duration (days)	602	1198	1038	561	886	784	680	1660	1149	1685
CAGR	0.01%	0.38%	0.14%	0.59%	0.33%	-0.10%	0.24%	-0.18%	-0.14%	-0.10%
Standard Deviation Ret	0.06%	0.06%	0.11%	0.12%	0.08%	0.02%	0.09%	0.03%	0.03%	0.06%
Success Ratio	33.33%	80.00%	50.00%	85.71%	54.55%	50.00%	50.00%	0.00%	50.00%	66.67%
Average Profit_Average Loss	1.04	12.55	1.3	36.13	2.24	0.22	2.22	0	0.1	0.48

Monte Carlo Analysis

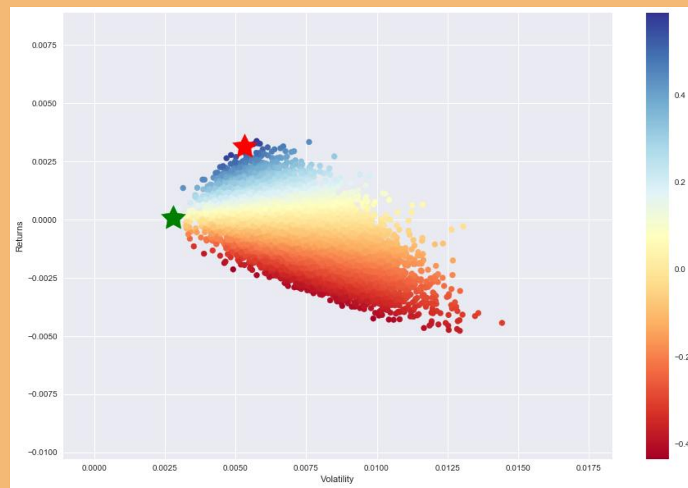
To get a better estimation about returns weight within each portfolio, a Monte Carlo analysis was made to select a better combination which resulted in a better Sharpe Ratio.

```
    #%% Portfolios Highlighted
    '''
    Two portfolios that we may like to highlight as being “special” are:
        1) the portfolio with the highest Sharpe Ratio (i.e. the highest risk adjusted
returns);
        2) The “minimum variance portfolio” which is the portfolio with the lowest
volatility.
    '''
    #set number of runs of random portfolio weights
    num_portfolios = 30000

    #set up array to hold results
    #We have increased the size of the array to hold the weight values for each stock
    results = np.zeros((4+Pairs_Len-1,num_portfolios))

    for i in xrange(num_portfolios):
        #select random weights for portfolio holdings
        weights = np.array(np.random.random(10))
        #rebalance weights to sum to 1
        weights /= np.sum(weights)

        #calculate portfolio return and volatility
        portfolio_return = np.sum(mean_daily_returns * weights) * 252
        portfolio_std_dev = np.sqrt(np.dot(weights.T,np.dot(cov_matrix, weights))) *
np.sqrt(252)

        #store results in results array
        results[0,i] = portfolio_return
        results[1,i] = portfolio_std_dev
        #store Sharpe Ratio (return / volatility) - risk free rate element excluded for
simplicity
        results[2,i] = results[0,i] / results[1,i]
        #iterate through the weight vector and add data to results array
        for k in range(len(weights)):
            results[k+3,i] = weights[k]

    #convert results array to Pandas DataFrame
    results_frame =
pd.DataFrame(results.T,columns=['ret','stdev','sharpe',Data[[0]],Data[[1]],Data[[2]],Data[
[3]],Data[[4]],Data[[5]],Data[[6]],Data[[7]],Data[[8]],Data[[9]]])

    #locate position of portfolio with highest Sharpe Ratio
    max_sharpe_port = results_frame.iloc[results_frame['sharpe'].idxmax()]
    #locate positon of portfolio with minimum standard deviation
    min_vol_port = results_frame.iloc[results_frame['stdev'].idxmin()]
```

```
    #create scatter plot coloured by Sharpe Ratio
    fig2 = plt.figure(figsize=(15,10))

plt.scatter(results_frame.stdev,results_frame.ret,c=results_frame.sharpe,cmap='RdYlBu')
    plt.xlabel('Volatility')
    plt.ylabel('Returns')
    plt.colorbar()
    #plot red star to highlight position of portfolio with highest Sharpe Ratio
    plt.scatter(max_sharpe_port[1],max_sharpe_port[0],marker=(5,1,0),color='r',s=1000)
    #plot green star to highlight position of minimum variance portfolio
    plt.scatter(min_vol_port[1],min_vol_port[0],marker=(5,1,0),color='g',s=1000)

    print(max_sharpe_port)
#    print(min_vol_port)
    #%% Portfolio with highest Sharpe Ratio
    weights_MC[:,j]=max_sharpe_port
    #set array holding portfolio weights of each stock
    weights = weights_MC[3:,j]

    #calculate annualised portfolio return
    portfolio_return = round(np.sum(mean_daily_returns * weights) * 252,4)
    #calculate annualised portfolio volatility
    portfolio_std_dev = round(np.sqrt(np.dot(weights.T,np.dot(cov_matrix, weights))) *
np.sqrt(252),4)
    print('Portfolio expected annualised return is {}% and volatility is
{}%').format(portfolio_return*100,portfolio_std_dev*100)

    Data = Data.dropna()

    Data['ret_pot'] = (Data.drop('ret_pot', axis=1) * weights).sum(axis=1)
    Data_Cum = (1.0+Data).cumprod()

    returns = Data['ret_pot']
    pnl = Data_Cum['ret_pot']
    Stats(returns,pnl)

    fig1 = plt.figure(figsize=(15,10))
    plt.plot(Data_Cum.index,Data_Cum[[0]],label=list(Data_Cum)[0])
    plt.plot(Data_Cum.index,Data_Cum[[1]],label=list(Data_Cum)[1])
    plt.plot(Data_Cum.index,Data_Cum[[2]],label=list(Data_Cum)[2])
    plt.plot(Data_Cum.index,Data_Cum[[3]],label=list(Data_Cum)[3])
    plt.plot(Data_Cum.index,Data_Cum[[4]],label=list(Data_Cum)[4])
    plt.plot(Data_Cum.index,Data_Cum[[5]],label=list(Data_Cum)[5])
    plt.plot(Data_Cum.index,Data_Cum[[6]],label=list(Data_Cum)[6])
    plt.plot(Data_Cum.index,Data_Cum[[7]],label=list(Data_Cum)[7])
    plt.plot(Data_Cum.index,Data_Cum[[8]],label=list(Data_Cum)[8])
    plt.plot(Data_Cum.index,Data_Cum[[9]],label=list(Data_Cum)[9])
    plt.plot(Data_Cum.index,Data_Cum[[10]],label=list(Data_Cum)[10],lw=3,color='k')
    plt.xlabel('Date')
    plt.ylabel('Cummulative Returns')
    plt.title('Portfolio '+str(j+1)+' Stocks Basket')
    plt.legend(loc=2)
    plt.show()
```

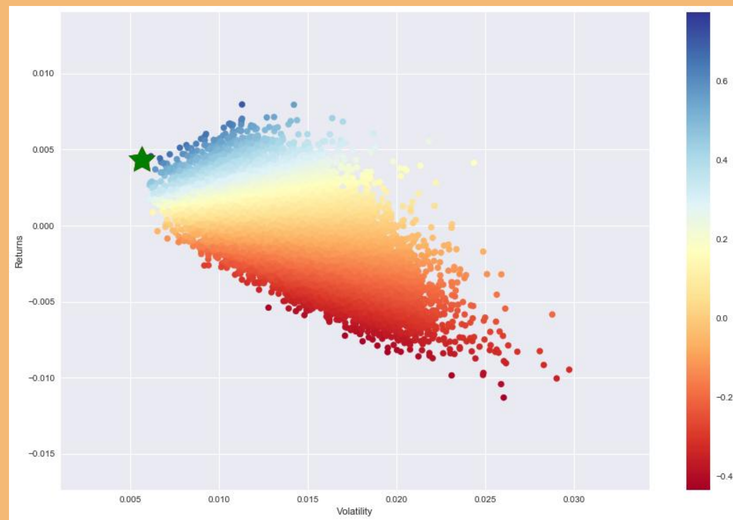
Chemicals

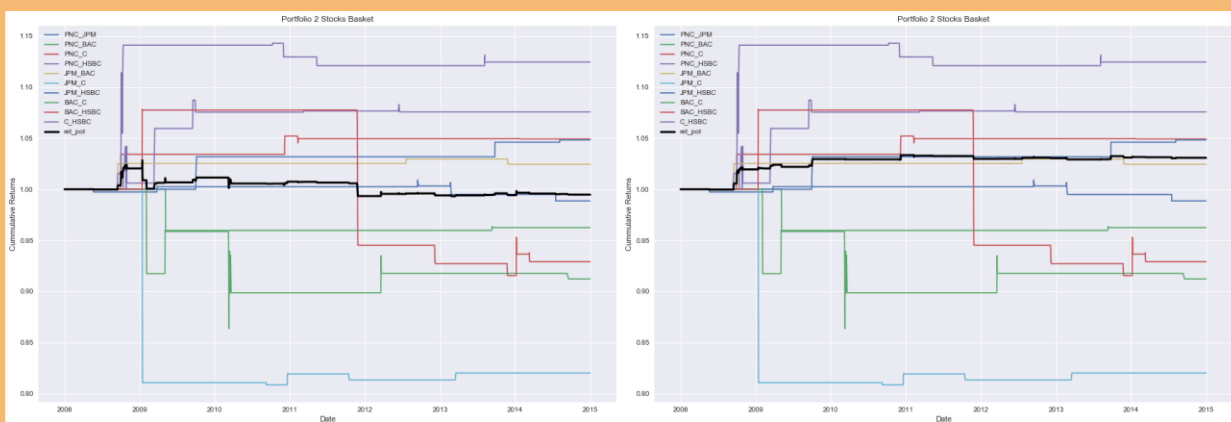
Parameters	Equally Weighted	Monte Carlo
Annualized Return	-0.06%	0.31%
Volatility	0.63%	0.53%
Total Return	-0.44%	2.21%
CAGR	-0.04%	0.22%
Standard Deviation Ret	0.04%	0.03%
Sharpe Ratio	-0.10	0.59
Max Drawdown	1.35%	0.83%
Drawdown Duration	949 days	949 days



Financials

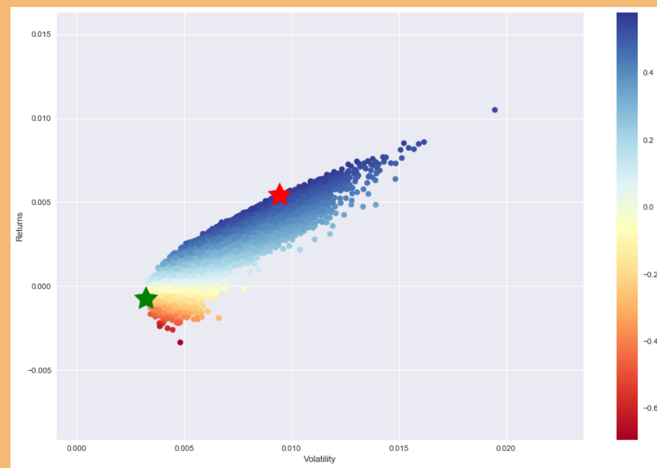
Parameters	Equally Weighted	Monte Carlo
Annualized Return	-0.07%	0.43%
Volatility	1.24%	0.56%
Total Return	-0.52%	3.08%
CAGR	-0.05%	0.30%
Standard Deviation Ret	0.08%	0.04%
Sharpe Ratio	-0.05	0.77
Max Drawdown	3.52%	0.43%
Drawdown Duration	1502 days	1023 days





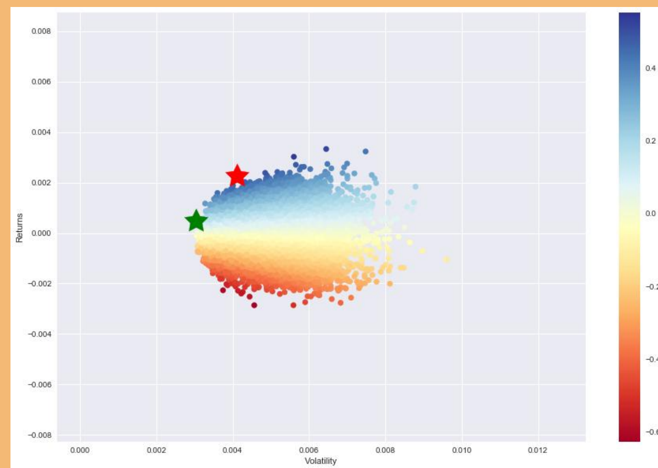
Entertainment

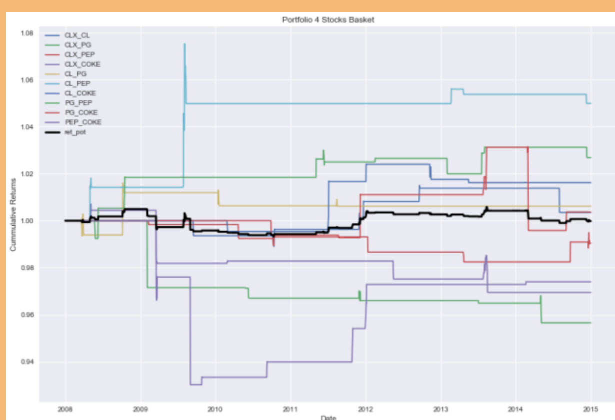
Parameters	Equally Weighted	Monte Carlo
Annualized Return	0.2%	0.55%
Volatility	0.58%	0.94%
Total Return	1.38%	3.86%
CAGR	0.14%	0.37%
Standard Deviation Ret	0.04%	0.06%
Sharpe Ratio	0.34	0.58
Max Drawdown	1.15%	0.97%
Drawdown Duration	1053 days	519 days

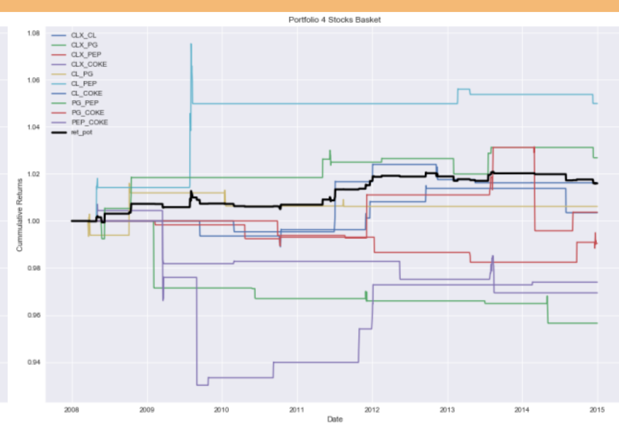


Consumer Goods

Parameters	Equally Weighted	Monte Carlo
Annualized Return	-0.0%	0.23%
Volatility	0.43%	0.41%
Total Return	-0.01%	1.60%
CAGR	-0.00%	0.16%
Standard Deviation Ret	0.03%	0.03%
Sharpe Ratio	-0.00	0.55
Max Drawdown	1.17%	0.77%
Drawdown Duration	1210 days	483 days

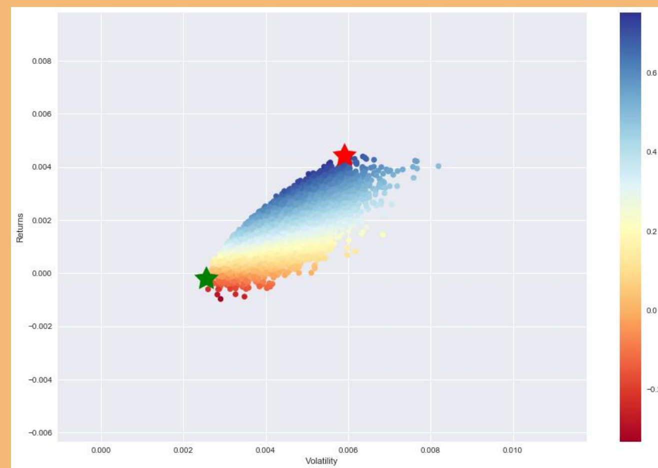


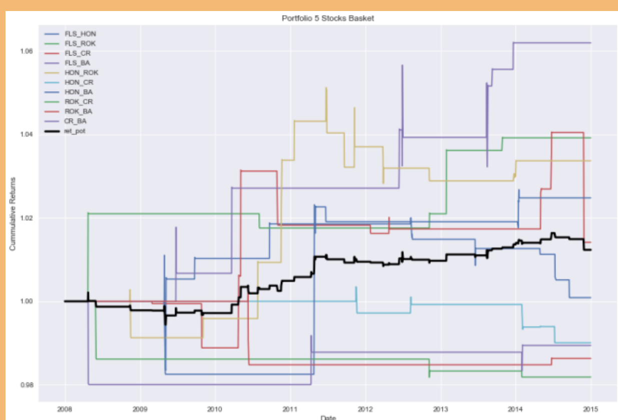


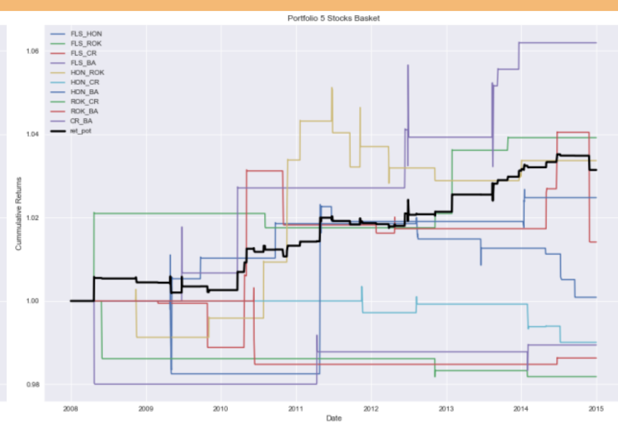


Industrial Goods

Parameters	Equally Weighted	Monte Carlo
Annualized Return	0.18%	0.44%
Volatility	0.38%	0.59%
Total Return	1.23%	3.14%
CAGR	0.12%	0.31%
Standard Deviation Ret	0.02%	0.04%
Sharpe Ratio	0.46	0.75
Max Drawdown	0.77%	0.42%
Drawdown Duration	511 days	255 days







All Portfolios

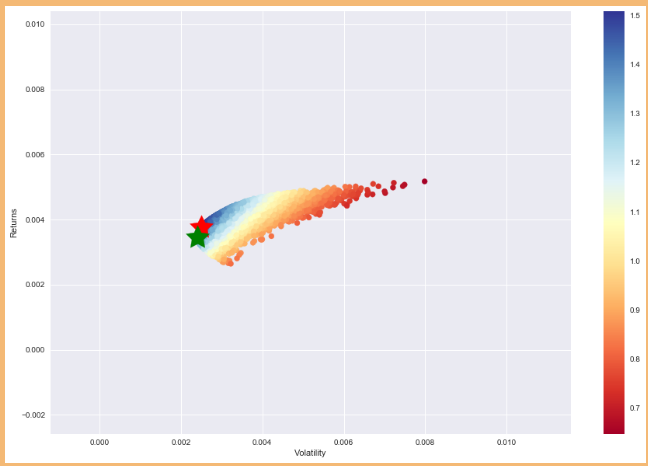
Here, all 5 portfolios with their weights, estimated via Monte Carlo to get higher Sharpe Ratio, were considered to form a diversifying portfolio, looking for better Sharpe Ratio.

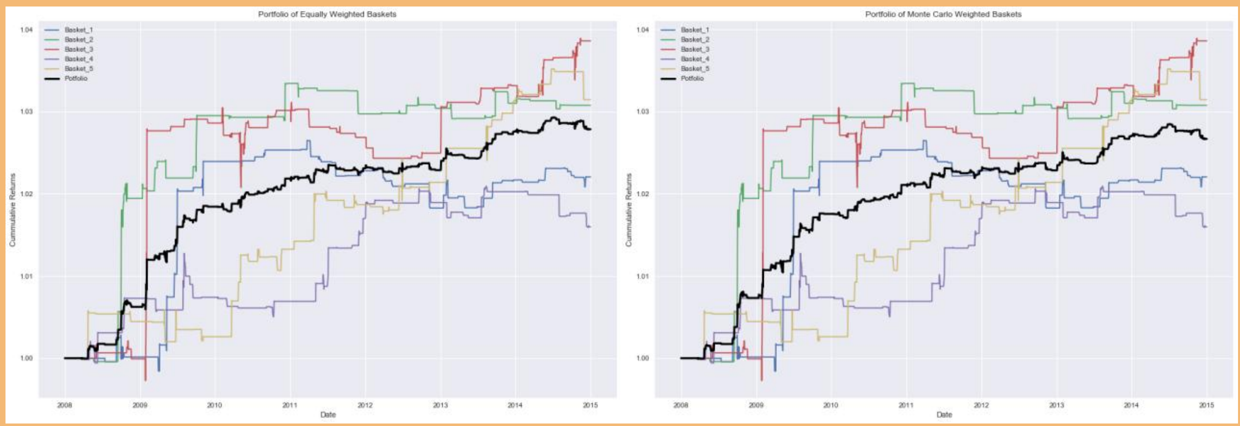
```
#%% Global Portfolio

Data_Port = pd.DataFrame()

for j in range(0,rows):
    Data = pd.read_csv(csv_dir+'/Returns_'+str(j+1)+'.csv',index_col=0)
    weights = weights_MC[3:,j]

    Data['ret_pot'] = (Data * weights).sum(axis=1)
    Data_Port['Basket_'+str(j+1)] = Data['ret_pot']

weights_Port = np.asarray([0.2,0.2,0.2,0.2,0.2])

#calculate mean daily return and covariance of daily returns
mean_daily_returns = Data_Port.mean()
cov_matrix = Data_Port.cov()

#calculate annualised portfolio return
portfolio_return = round(np.sum(mean_daily_returns * weights_Port) * 252,4)
#calculate annualised portfolio volatility
portfolio_std_dev = round(np.sqrt(np.dot(weights_Port.T,np.dot(cov_matrix, weights_Port)))
* np.sqrt(252),4)
print('Portfolio expected annualised return is {}% and volatility is
{}%').format(portfolio_return*100,portfolio_std_dev*100)

Data_Port['Potfolio'] = (Data_Port * weights_Port).sum(axis=1)
Data_Cum_Port = (1.0+Data_Port).cumprod()

#print(Data_Cum_Port.tail())
Data_Cum_Port.index = pd.to_datetime(Data_Cum_Port.index)

returns = Data_Port['Potfolio']
pnl = Data_Cum_Port['Potfolio']

Stats(returns,pnl)

fig1 = plt.figure(figsize=(15,10))
plt.plot(Data_Cum_Port.index,Data_Cum_Port[[0]],label=list(Data_Cum_Port)[0])
plt.plot(Data_Cum_Port.index,Data_Cum_Port[[1]],label=list(Data_Cum_Port)[1])
plt.plot(Data_Cum_Port.index,Data_Cum_Port[[2]],label=list(Data_Cum_Port)[2])
plt.plot(Data_Cum_Port.index,Data_Cum_Port[[3]],label=list(Data_Cum_Port)[3])
plt.plot(Data_Cum_Port.index,Data_Cum_Port[[4]],label=list(Data_Cum_Port)[4])
plt.plot(Data_Cum_Port.index,Data_Cum_Port[[5]],label=list(Data_Cum_Port)[5],lw=3,color='k
')
plt.xlabel('Date')
plt.ylabel('Cummulative Returns')
plt.title('Portfolio of Equally Weighted Baskets')
```

```
plt.legend(loc=2)
plt.show()

#%% Portfolios Highlighted
'''
Two portfolios that we may like to highlight as being “special” are:
    1) the portfolio with the highest Sharpe Ratio (i.e. the highest risk adjusted
returns);
    2) The “minimum variance portfolio” which is the portfolio with the lowest volatility.
'''
#set number of runs of random portfolio weights
num_portfolios = 30000

#set up array to hold results
#We have increased the size of the array to hold the weight values for each stock
results = np.zeros((4+rows-1,num_portfolios))

for i in xrange(num_portfolios):
    #select random weights for portfolio holdings
    weights = np.array(np.random.random(5))
    #rebalance weights to sum to 1
    weights /= np.sum(weights)

    #calculate portfolio return and volatility
    portfolio_return = np.sum(mean_daily_returns * weights) * 252
    portfolio_std_dev = np.sqrt(np.dot(weights.T,np.dot(cov_matrix, weights))) *
np.sqrt(252)

    #store results in results array
    results[0,i] = portfolio_return
    results[1,i] = portfolio_std_dev
    #store Sharpe Ratio (return / volatility) - risk free rate element excluded for
simplicity
    results[2,i] = results[0,i] / results[1,i]
    #iterate through the weight vector and add data to results array
    for k in range(len(weights)):
        results[k+3,i] = weights[k]

#convert results array to Pandas DataFrame
results_frame =
pd.DataFrame(results.T,columns=['ret','stdev','sharpe',Data_Cum_Port[[0]],Data_Cum_Port[[1
]],Data_Cum_Port[[2]],Data_Cum_Port[[3]],Data_Cum_Port[[4]]])

#locate position of portfolio with highest Sharpe Ratio
max_sharpe_port = results_frame.iloc[results_frame['sharpe'].idxmax()]
#locate positon of portfolio with minimum standard deviation
min_vol_port = results_frame.iloc[results_frame['stdev'].idxmin()]

#create scatter plot coloured by Sharpe Ratio
fig2 = plt.figure(figsize=(15,10))
plt.scatter(results_frame.stdev,results_frame.ret,c=results_frame.sharpe,cmap='RdYlBu')
plt.xlabel('Volatility')
plt.ylabel('Returns')
```

```
plt.colorbar()
#plot red star to highlight position of portfolio with highest Sharpe Ratio
plt.scatter(max_sharpe_port[1],max_sharpe_port[0],marker=(5,1,0),color='r',s=1000)
#plot green star to highlight position of minimum variance portfolio
plt.scatter(min_vol_port[1],min_vol_port[0],marker=(5,1,0),color='g',s=1000)

print(max_sharpe_port)
#print(min_vol_port)

#set array holding portfolio weights of each stock
weights_Port = np.array(max_sharpe_port[3:])

#calculate annualised portfolio return
portfolio_return = round(np.sum(mean_daily_returns * weights_Port) * 252,4)
#calculate annualised portfolio volatility
portfolio_std_dev = round(np.sqrt(np.dot(weights_Port.T,np.dot(cov_matrix, weights_Port)))
* np.sqrt(252),4)
print('Portfolio expected annualised return is {}% and volatility is
{}%').format(portfolio_return*100,portfolio_std_dev*100)

Data_Port['Potfolio'] = (Data_Port.drop('Potfolio', axis=1) * weights_Port).sum(axis=1)
Data_Cum_Port = (1.0+Data_Port).cumprod()

#print(Data_Cum_Port.tail())
Data_Cum_Port.index = pd.to_datetime(Data_Cum_Port.index)

returns = Data_Port['Potfolio']
pnl = Data_Cum_Port['Potfolio']

Stats(returns,pnl)

fig1 = plt.figure(figsize=(15,10))
plt.plot(Data_Cum_Port.index,Data_Cum_Port[[0]],label=list(Data_Cum_Port)[0])
plt.plot(Data_Cum_Port.index,Data_Cum_Port[[1]],label=list(Data_Cum_Port)[1])
plt.plot(Data_Cum_Port.index,Data_Cum_Port[[2]],label=list(Data_Cum_Port)[2])
plt.plot(Data_Cum_Port.index,Data_Cum_Port[[3]],label=list(Data_Cum_Port)[3])
plt.plot(Data_Cum_Port.index,Data_Cum_Port[[4]],label=list(Data_Cum_Port)[4])
plt.plot(Data_Cum_Port.index,Data_Cum_Port[[5]],label=list(Data_Cum_Port)[5],lw=3,color='k
')
plt.xlabel('Date')
plt.ylabel('Cummulative Returns')
plt.title('Portfolio of Monte Carlo Weighted Baskets')
plt.legend(loc=2)
plt.show()
```

Parameters	Equally Weighted	Monte Carlo
Annualized Return	0.45%	0.43%
Volatility	0.31%	0.29%
Total Return	3.19%	2.67%
CAGR	0.31%	0.26%
Standard Deviation Ret	0.02%	0.02%
Sharpe Ratio	1.45	1.51
Max Drawdown Lev 1	0.3%	0.18%
Max Drawdown Lev 2	0.6%	0.36%
Max Drawdown Lev 3	0.9%	0.54%
Max Drawdown Lev 10	3.0%	1.8%
Max Drawdown Lev 30	9.0%	5.4%
Max Drawdown Lev 30	15.0%	9.0%
Drawdown Duration	249 days	130 days





Conclusions

Beyond the arbitrary selection of parameters, from backtesting made pair by pair, a wide range of results were obtained. Returns throughout analyzed horizon ranged from -18% to 18%. But once, aggregation among returns of pairs in the same sector is made, the returns were less volatile and made a positive profit, scarce though. With the right parameter selection per pair or per sector basket, having a close look on data snooping, it is likely to have slightly better results than those presented.

The consideration of all portfolios shows the success of diversifying within different sectors and with the right weighting across portfolios, a consistent algorithm could be obtained. Furthermore, the Monte Carlo modeling was effective enough to get an insight of correct weights among pairs and sector portfolios for final aggregation.

Future work

For future work, there are a lot of analysis to carry out. Here are a few listed:

- Capital allocation could be analyzed based on a minimum variance portfolio criteria.
- Optimization could be performed with minimum 1 year data for out-of-sample test. Notice that there is no difference in the parameters of any pair. It may be worth analyze the performance of optimized parameters pair by pair and make comparisons among them.
- Walk Forward Analysis could be done to asses a proper risk management system.
- Machine Learning potential could be analyzed in regards with ongoing optimization to get a more reliable estimation of strategy setting while it is running and gathering new information from markets.
- Verify the potential of filter out most false signals trough noise time series removal, i.e. Kalman filter.

References

- Successful Algorithmic Trading. Michael L. Halls-Moore. A Step-By-Step guide to Quantitative Strategies.

Annex

Supporting backtesting files are described as follows:

- Backtest.py: It encapsulates the settings and components for carrying out an event-driven backtest.
- Data.py: It is where data is handled. Its goal is to output a generated set of bars (OHLCVI) for each symbol as requested.
- Event.py: Is the base class providing an interface for all subsequent events, that will trigger further events in the trading infrastructure.
- Execution.py: It is where ExecutionHandler abstract class is defined. It can be used to subclass simulated brokerages or live brokerages, with identical interfaces.
- Performance.py: It is where all performance parameters are calculated based on backtesting results.
- Plot_performance_JOMN.py: It is where the performance plot is setting up for visualization.

- Portfolio.py: Definition of Portfolio class which handles the position and market values of all instruments at a resolution of a bar. It is where all positions are made, updated and closed. It is where is made the risk management of the strategy and post-backtesting statistics are calculated.
- Strategy.py: Definition of Strategy as an abstract base class proving an interface for all subsequent (inherited) strategy handling objects.

Backtest.py

```python
#!/usr/bin/python
# -*- coding: utf-8 -*-

# backtest.py

from __future__ import print_function

import datetime
import pprint
try:
    import Queue as queue
except ImportError:
    import queue
import time


class Backtest(object):
    """
    Enscapsulates the settings and components for carrying out
    an event-driven backtest.
    """

    def __init__(
        self, csv_dir, symbol_list, initial_capital,
        heartbeat, start_date, data_handler,
#        execution_handler, portfolio, strategy
        execution_handler, portfolio, strategy,pair
    ):
        """
        Initialises the backtest.

        Parameters:
        csv_dir - The hard root to the CSV data directory.
        symbol_list - The list of symbol strings.
        intial_capital - The starting capital for the portfolio.
        heartbeat - Backtest "heartbeat" in seconds
        start_date - The start datetime of the strategy.
        data_handler - (Class) Handles the market data feed.
        execution_handler - (Class) Handles the orders/fills for trades.
        portfolio - (Class) Keeps track of portfolio current and prior positions.
        strategy - (Class) Generates signals based on market data.
        """
        self.csv_dir = csv_dir
        self.symbol_list = symbol_list
        self.initial_capital = initial_capital
        self.heartbeat = heartbeat
        self.start_date = start_date

        self.pair = pair

        self.data_handler_cls = data_handler
        self.execution_handler_cls = execution_handler
        self.portfolio_cls = portfolio
        self.strategy_cls = strategy
```

```
        self.events = queue.Queue()

        self.signals = 0
        self.orders = 0
        self.fills = 0
        self.num_strats = 1

#         self._generate_trading_instances()
        self._generate_trading_instances(pair)

#     def _generate_trading_instances(self):
    def _generate_trading_instances(self,pair):
        """
        Generates the trading instance objects from
        their class types.
        """
        print(
            "Creating DataHandler, Strategy, Portfolio and ExecutionHandler"
        )
        self.data_handler = self.data_handler_cls(self.events, self.csv_dir,
self.symbol_list)
#         self.strategy = self.strategy_cls(self.data_handler, self.events)
        self.strategy = self.strategy_cls(self.data_handler, self.events, pair)
#         self.portfolio = self.portfolio_cls(self.data_handler, self.events,
self.start_date,
#                                             self.initial_capital)
        self.portfolio = self.portfolio_cls(self.data_handler, self.events,
self.start_date,pair,
                                            self.initial_capital)
        self.execution_handler = self.execution_handler_cls(self.events)

    def _run_backtest(self):
        """
        Executes the backtest.
        """
        i = 0
        while True:
            i += 1
#             print(i)
            # Update the market bars
            if self.data_handler.continue_backtest == True:
                self.data_handler.update_bars()
            else:
                break

            # Handle the events
            while True:
                try:
                    event = self.events.get(False)
                except queue.Empty:
                    break
                else:
                    if event is not None:
                        if event.type == 'MARKET':
                            self.strategy.calculate_signals(event)
```

```
                            self.portfolio.update_timeindex(event)

                        elif event.type == 'SIGNAL':
                            self.signals += 1
#                            print(self.signals)
                            self.portfolio.update_signal(event)

                        elif event.type == 'ORDER':
                            self.orders += 1
                            self.execution_handler.execute_order(event)

                        elif event.type == 'FILL':
                            self.fills += 1
                            self.portfolio.update_fill(event)

            time.sleep(self.heartbeat)

#    def _output_performance(self):
    def _output_performance(self,csv_file):
        """
        Outputs the strategy performance from the backtest.
        """
        self.portfolio.create_equity_curve_dataframe()

        print("Creating summary stats...")
#        stats = self.portfolio.output_summary_stats()
        stats = self.portfolio.output_summary_stats(csv_file)

        print("Creating equity curve...")
        print(self.portfolio.equity_curve.tail(10))
        pprint.pprint(stats)

        print("Signals: %s" % self.signals)
        print("Orders: %s" % self.orders)
        print("Fills: %s" % self.fills)

#    def simulate_trading(self):
    def simulate_trading(self,csv_file):
        """
        Simulates the backtest and outputs portfolio performance.
        """
        self._run_backtest()
#        self._output_performance()
        self._output_performance(csv_file)
```

Data.py

```python
#!/usr/bin/python
# -*- coding: utf-8 -*-

# data.py

from __future__ import print_function

from abc import ABCMeta, abstractmethod
import datetime
import os, os.path

import numpy as np
import pandas as pd

from event import MarketEvent


class DataHandler(object):
    """
    DataHandler is an abstract base class providing an interface for
    all subsequent (inherited) data handlers (both live and historic).

    The goal of a (derived) DataHandler object is to output a generated
    set of bars (OHLCVI) for each symbol requested.

    This will replicate how a live strategy would function as current
    market data would be sent "down the pipe". Thus a historic and live
    system will be treated identically by the rest of the backtesting suite.
    """

    __metaclass__ = ABCMeta

    @abstractmethod
    def get_latest_bar(self, symbol):
        """
        Returns the last bar updated.
        """
        raise NotImplementedError("Should implement get_latest_bar()")

    @abstractmethod
    def get_latest_bars(self, symbol, N=1):
        """
        Returns the last N bars updated.
        """
        raise NotImplementedError("Should implement get_latest_bars()")

    @abstractmethod
    def get_latest_bar_datetime(self, symbol):
        """
        Returns a Python datetime object for the last bar.
        """
        raise NotImplementedError("Should implement get_latest_bar_datetime()")

    @abstractmethod
    def get_latest_bar_value(self, symbol, val_type):
```

```
        """
        Returns one of the Open, High, Low, Close, Volume or OI
        from the last bar.
        """
        raise NotImplementedError("Should implement get_latest_bar_value()")

    @abstractmethod
    def get_latest_bars_values(self, symbol, val_type, N=1):
        """
        Returns the last N bar values from the
        latest_symbol list, or N-k if less available.
        """
        raise NotImplementedError("Should implement get_latest_bars_values()")

    @abstractmethod
    def update_bars(self):
        """
        Pushes the latest bars to the bars_queue for each symbol
        in a tuple OHLCVI format: (datetime, open, high, low,
        close, volume, open interest).
        """
        raise NotImplementedError("Should implement update_bars()")


class HistoricCSVDataHandler(DataHandler):
    """
    HistoricCSVDataHandler is designed to read CSV files for
    each requested symbol from disk and provide an interface
    to obtain the "latest" bar in a manner identical to a live
    trading interface.
    """

    def __init__(self, events, csv_dir, symbol_list):
        """
        Initialises the historic data handler by requesting
        the location of the CSV files and a list of symbols.

        It will be assumed that all files are of the form
        'symbol.csv', where symbol is a string in the list.

        Parameters:
        events - The Event Queue.
        csv_dir - Absolute directory path to the CSV files.
        symbol_list - A list of symbol strings.
        """
        self.events = events
        self.csv_dir = csv_dir
        self.symbol_list = symbol_list

        self.symbol_data = {}
        self.latest_symbol_data = {}
        self.continue_backtest = True
        self.bar_index = 0

        self._open_convert_csv_files()
```

```
    def _open_convert_csv_files(self):
        """
        Opens the CSV files from the data directory, converting
        them into pandas DataFrames within a symbol dictionary.

        For this handler it will be assumed that the data is
        taken from Yahoo. Thus its format will be respected.
        """
        comb_index = None
        for s in self.symbol_list:
            # Load the CSV file with no header information, indexed on date
            self.symbol_data[s] = pd.io.parsers.read_csv(
                os.path.join(self.csv_dir, '%s.csv' % s),
                header=0, index_col=0, parse_dates=True,
                names=[
                    'datetime', 'open', 'high',
                    'low', 'close', 'volume', 'adj_close'
                ]
            ).sort()

            # Combine the index to pad forward values
            if comb_index is None:
                comb_index = self.symbol_data[s].index
            else:
                comb_index.union(self.symbol_data[s].index)

            # Set the latest symbol_data to None
            self.latest_symbol_data[s] = []

        for s in self.symbol_list:
            self.symbol_data[s] = self.symbol_data[s].reindex(
                index=comb_index, method='pad'
            )
            self.symbol_data[s]["returns"] =
self.symbol_data[s]["adj_close"].pct_change()
            self.symbol_data[s] = self.symbol_data[s].iterrows()

    def _get_new_bar(self, symbol):
        """
        Returns the latest bar from the data feed.
        """
        for b in self.symbol_data[symbol]:
            yield b

    def get_latest_bar(self, symbol):
        """
        Returns the last bar from the latest_symbol list.
        """
        try:
            bars_list = self.latest_symbol_data[symbol]
        except KeyError:
            print("That symbol is not available in the historical data set.")
            raise
        else:
```

```
            return bars_list[-1]

    def get_latest_bars(self, symbol, N=1):
        """
        Returns the last N bars from the latest_symbol list,
        or N-k if less available.
        """
        try:
            bars_list = self.latest_symbol_data[symbol]
        except KeyError:
            print("That symbol is not available in the historical data set.")
            raise
        else:
            return bars_list[-N:]

    def get_latest_bar_datetime(self, symbol):
        """
        Returns a Python datetime object for the last bar.
        """
        try:
            bars_list = self.latest_symbol_data[symbol]
        except KeyError:
            print("That symbol is not available in the historical data set.")
            raise
        else:
            return bars_list[-1][0]

    def get_latest_bar_value(self, symbol, val_type):
        """
        Returns one of the Open, High, Low, Close, Volume or OI
        values from the pandas Bar series object.
        """
        try:
            bars_list = self.latest_symbol_data[symbol]
        except KeyError:
            print("That symbol is not available in the historical data set.")
            raise
        else:
            return getattr(bars_list[-1][1], val_type)

    def get_latest_bars_values(self, symbol, val_type, N=1):
        """
        Returns the last N bar values from the
        latest_symbol list, or N-k if less available.
        """
        try:
            bars_list = self.get_latest_bars(symbol, N)
        except KeyError:
            print("That symbol is not available in the historical data set.")
            raise
        else:
            return np.array([getattr(b[1], val_type) for b in bars_list])

    def update_bars(self):
        """
```

```
        Pushes the latest bar to the latest_symbol_data structure
        for all symbols in the symbol list.
        """
        for s in self.symbol_list:
            try:
                bar = next(self._get_new_bar(s))
            except StopIteration:
                self.continue_backtest = False
            else:
                if bar is not None:
                    self.latest_symbol_data[s].append(bar)
        self.events.put(MarketEvent())
```

Event.py

```python
#!/usr/bin/python
# -*- coding: utf-8 -*-

# event.py

from __future__ import print_function


class Event(object):
    """
    Event is base class providing an interface for all subsequent
    (inherited) events, that will trigger further events in the
    trading infrastructure.
    """
    pass


class MarketEvent(Event):
    """
    Handles the event of receiving a new market update with
    corresponding bars.
    """

    def __init__(self):
        """
        Initialises the MarketEvent.
        """
        self.type = 'MARKET'


class SignalEvent(Event):
    """
    Handles the event of sending a Signal from a Strategy object.
    This is received by a Portfolio object and acted upon.
    """

    def __init__(self, strategy_id, symbol, datetime, signal_type, strength):
        """
        Initialises the SignalEvent.

        Parameters:
        strategy_id - The unique ID of the strategy sending the signal.
        symbol - The ticker symbol, e.g. 'GOOG'.
        datetime - The timestamp at which the signal was generated.
        signal_type - 'LONG' or 'SHORT'.
        strength - An adjustment factor "suggestion" used to scale
            quantity at the portfolio level. Useful for pairs strategies.
        """
        self.strategy_id = strategy_id
        self.type = 'SIGNAL'
        self.symbol = symbol
        self.datetime = datetime
        self.signal_type = signal_type
        self.strength = strength
```

```python
class OrderEvent(Event):
    """
    Handles the event of sending an Order to an execution system.
    The order contains a symbol (e.g. GOOG), a type (market or limit),
    quantity and a direction.
    """

    def __init__(self, symbol, order_type, quantity, direction):
        """
        Initialises the order type, setting whether it is
        a Market order ('MKT') or Limit order ('LMT'), has
        a quantity (integral) and its direction ('BUY' or
        'SELL').

        TODO: Must handle error checking here to obtain
        rational orders (i.e. no negative quantities etc).

        Parameters:
        symbol - The instrument to trade.
        order_type - 'MKT' or 'LMT' for Market or Limit.
        quantity - Non-negative integer for quantity.
        direction - 'BUY' or 'SELL' for long or short.
        """
        self.type = 'ORDER'
        self.symbol = symbol
        self.order_type = order_type
        self.quantity = quantity
        self.direction = direction

    def print_order(self):
        """
        Outputs the values within the Order.
        """
        print(
            "Order: Symbol=%s, Type=%s, Quantity=%s, Direction=%s" %
            (self.symbol, self.order_type, self.quantity, self.direction)
        )


class FillEvent(Event):
    """
    Encapsulates the notion of a Filled Order, as returned
    from a brokerage. Stores the quantity of an instrument
    actually filled and at what price. In addition, stores
    the commission of the trade from the brokerage.

    TODO: Currently does not support filling positions at
    different prices. This will be simulated by averaging
    the cost.
    """

    def __init__(self, timeindex, symbol, exchange, quantity,
                 direction, fill_cost, commission=None):
        """
```

```
        Initialises the FillEvent object. Sets the symbol, exchange,
        quantity, direction, cost of fill and an optional
        commission.

        If commission is not provided, the Fill object will
        calculate it based on the trade size and Interactive
        Brokers fees.

        Parameters:
        timeindex - The bar-resolution when the order was filled.
        symbol - The instrument which was filled.
        exchange - The exchange where the order was filled.
        quantity - The filled quantity.
        direction - The direction of fill ('BUY' or 'SELL')
        fill_cost - The holdings value in dollars.
        commission - An optional commission sent from IB.
        """
        self.type = 'FILL'
        self.timeindex = timeindex
        self.symbol = symbol
        self.exchange = exchange
        self.quantity = quantity
        self.direction = direction
        self.fill_cost = fill_cost

        # Calculate commission
        if commission is None:
            self.commission = self.calculate_ib_commission()
        else:
            self.commission = commission

    def calculate_ib_commission(self):
        """
        Calculates the fees of trading based on an Interactive
        Brokers fee structure for API, in USD.

        This does not include exchange or ECN fees.

        Based on "US API Directed Orders":
        https://www.interactivebrokers.com/en/index.php?f=commission&p=stocks2
        """
        full_cost = 1.3
        if self.quantity <= 500:
            full_cost = max(1.3, 0.013 * self.quantity)
        else: # Greater than 500
            full_cost = max(1.3, 0.008 * self.quantity)
        return full_cost
```

Execution.py

```python
#!/usr/bin/python
# -*- coding: utf-8 -*-

# execution.py

from __future__ import print_function

from abc import ABCMeta, abstractmethod
import datetime
try:
    import Queue as queue
except ImportError:
    import queue

from event import FillEvent, OrderEvent


class ExecutionHandler(object):
    """
    The ExecutionHandler abstract class handles the interaction
    between a set of order objects generated by a Portfolio and
    the ultimate set of Fill objects that actually occur in the
    market.

    The handlers can be used to subclass simulated brokerages
    or live brokerages, with identical interfaces. This allows
    strategies to be backtested in a very similar manner to the
    live trading engine.
    """

    __metaclass__ = ABCMeta

    @abstractmethod
    def execute_order(self, event):
        """
        Takes an Order event and executes it, producing
        a Fill event that gets placed onto the Events queue.

        Parameters:
        event - Contains an Event object with order information.
        """
        raise NotImplementedError("Should implement execute_order()")


class SimulatedExecutionHandler(ExecutionHandler):
    """
    The simulated execution handler simply converts all order
    objects into their equivalent fill objects automatically
    without latency, slippage or fill-ratio issues.

    This allows a straightforward "first go" test of any strategy,
    before implementation with a more sophisticated execution
    handler.
    """
```

```python
    def __init__(self, events):
        """
        Initialises the handler, setting the event queues
        up internally.

        Parameters:
        events - The Queue of Event objects.
        """
        self.events = events

    def execute_order(self, event):
        """
        Simply converts Order objects into Fill objects naively,
        i.e. without any latency, slippage or fill ratio problems.

        Parameters:
        event - Contains an Event object with order information.
        """
        if event.type == 'ORDER':
            fill_event = FillEvent(
                datetime.datetime.utcnow(), event.symbol,
                'ARCA', event.quantity, event.direction, None
            )
            self.events.put(fill_event)
```

Performance.py

```python
#!/usr/bin/python
# -*- coding: utf-8 -*-

# performance.py

from __future__ import print_function

import numpy as np
import pandas as pd


def create_sharpe_ratio(returns, periods=252):
    """
    Create the Sharpe ratio for the strategy, based on a
    benchmark of zero (i.e. no risk-free rate information).

    Parameters:
    returns - A pandas Series representing period percentage returns.
    periods - Daily (252), Hourly (252*6.5), Minutely(252*6.5*60) etc.
    """
    return np.sqrt(periods) * (np.mean(returns)) / np.std(returns)

def create_CAGR(pnl):
    """
    Compound Annual Growth Rates (CAGR)

    Parameters:
    pnl - A pandas Series representing period percentage returns.

    Returns:
    CAGR - Compound Annual Growth Rates
    """

    # days
    days = (pnl.index[-1]-pnl.index[1]).days

    #CAGR = ((pnl['equity_curve'][-1]/pnl['equity_curve'][0])**(252/days))-1
#     print(pnl)
#     print(pnl.index[-1])
#     print(pnl.index[1])
#     print(days)
#     print(pnl[-1])
#     print(pnl[1])
    CAGR = ((pnl[-1]/pnl[1])**(252.0/days))-1.0

    return CAGR

def Success_Ratio(pnl,comission):
    """
    Success ratio for the strategy.

    Parameters:
    pnl - A pandas Series representing period percentage returns.
    comission - Information about entry and exit trades dates-.
```

```
    Returns:
    Sucess_Ratio - Success ratio
    """
#    returns_n = returns[(returns.T != 0).any()]
#
#    tot = float(len(returns_n))
#
#    pos = float(returns_n[returns_n >= 0.0].count())

    Trades_Dates = comission[comission > 0.0]
    Num_Trades = Trades_Dates.count()
#    print(Num_Trades)

    PNL = pnl.pct_change()

    pnl_trades = pd.DataFrame()

    for i in range(0,Num_Trades/2):
        start_date=Trades_Dates.index[i*2]
        end_date=Trades_Dates.index[i*2+1]
        PNL_trade_ = PNL.loc[(PNL.index >= start_date) & (PNL.index <= end_date)]
#        PNL_Trade = (1.0+PNL_trade_).cumprod()
        PNL_Trade = (1.0+PNL_trade_).prod()-1.0
        pnl_trades = pnl_trades.append([PNL_Trade], ignore_index = True)
#        print(PNL_Trade)
#    print(PNL[PNL != 0.0])
#    print(PNL[PNL < 0.0].count())

#    print(pnl_trades)

    pos = pnl_trades[pnl_trades > 0.0].count()
#    print(pnl_trades[pnl_trades > 0.0].count())

    neg = pnl_trades[pnl_trades < 0.0].count()
#    print(pnl_trades[pnl_trades < 0.0].count())

    if pnl_trades[pnl_trades < 0.0].empty and pnl_trades[pnl_trades > 0.0].empty:
        Success_Ratio = 0
    else:
        try:
            Success_Ratio = pos/(pos+neg)
        except ZeroDivisionError:
            Success_Ratio = float('Inf')

    return Success_Ratio

def Average_Profit_Average_Loss(pnl,comission):
    """
    Average_Profit_Average_Loss for the strategy.

    Parameters:
    pnl - A pandas Series representing period percentage returns.
    comission - Information about entry and exit trades dates-.

    Returns:
```

```
    Average_Profit_Average_Loss - Relation Average Profit vs Average Loss
    """

#    Num_Trades = comission[comission > 0.0].count()
#    print(Num_Trades)
#    print(comission[comission > 0.0])

    Trades_Dates = comission[comission > 0.0]
    Num_Trades = Trades_Dates.count()
#    print(Num_Trades)

    PNL = pnl.pct_change()

    pnl_trades = pd.DataFrame()

    for i in range(0,Num_Trades/2):
        start_date=Trades_Dates.index[i*2]
        end_date=Trades_Dates.index[i*2+1]
        PNL_trade_ = PNL.loc[(PNL.index >= start_date) & (PNL.index <= end_date)]
#        PNL_Trade = (1.0+PNL_trade_).cumprod()
        PNL_Trade = (1.0+PNL_trade_).prod()-1.0
        pnl_trades = pnl_trades.append([PNL_Trade], ignore_index = True)

#    print(pnl_trades)

    pos = pnl_trades[pnl_trades > 0.0].sum()

    neg = pnl_trades[pnl_trades < 0.0].sum()

    if pnl_trades[pnl_trades < 0.0].empty and pnl_trades[pnl_trades > 0.0].empty:
        Average_Profit_Average_Loss = 0
    else:
        try:
            Average_Profit_Average_Loss = pos/np.abs(neg)
        except ZeroDivisionError:
            Average_Profit_Average_Loss = float('Inf')

    return Average_Profit_Average_Loss

def Standard_Deviation(returns):
    """
    Standard Deviation

    Parameters:
    returns - A pandas Series representing period percentage returns.

    Returns:
    Std_Dev - Standard deviation of returns
    """

    Std_Dev = np.std(returns)

    return Std_Dev
```

```python
def create_drawdowns(pnl):
    """
    Calculate the largest peak-to-trough drawdown of the PnL curve
    as well as the duration of the drawdown. Requires that the
    pnl_returns is a pandas Series.

    Parameters:
    pnl - A pandas Series representing period percentage returns.

    Returns:
    drawdown, duration - Highest peak-to-trough drawdown and duration.
    """

    # Calculate the cumulative returns curve
    # and set up the High Water Mark
    hwm = [0]

    # Create the drawdown and duration series
    idx = pnl.index
    drawdown = pd.Series(index = idx)
    duration = pd.Series(index = idx)

    # Loop over the index range
    for t in range(1, len(idx)):
        hwm.append(max(hwm[t-1], pnl[t]))
        drawdown[t]= (hwm[t]-pnl[t])
        duration[t]= (0 if drawdown[t] == 0 else duration[t-1]+1)
    return drawdown, drawdown.max(), duration.max()
```

Plot_performance_JOMN.py

```python
#!/usr/bin/python
# -*- coding: utf-8 -*-

# plot_performance.py

import os.path

import numpy as np
import matplotlib.pyplot as plt
import pandas as pd


def plot_performance(csv_file):
    data = pd.io.parsers.read_csv(
#         "equity.csv", header=0,
        csv_file, header=0,
        parse_dates=True, index_col=0
    ).sort()

    # Plot three charts: Equity curve,
    # period returns, drawdowns
    fig = plt.figure(figsize=(15,10))
    # Set the outer colour to white
    fig.patch.set_facecolor('white')

    # Plot the equity curve
    ax1 = fig.add_subplot(311, ylabel='Portfolio value, %')
    data['equity_curve'].plot(ax=ax1, color="blue", lw=2.)
    plt.grid(True)

    # Plot the returns
    ax2 = fig.add_subplot(312, ylabel='Period returns, %')
    data['returns'].plot(ax=ax2, color="black", lw=2.)
    plt.grid(True)

    # Plot the returns
    ax3 = fig.add_subplot(313, ylabel='Drawdowns, %')
    data['drawdown'].plot(ax=ax3, color="red", lw=2.)
    plt.grid(True)

    # Plot the figure
    plt.show()
```

Portfolio.py

```
#!/usr/bin/python
# -*- coding: utf-8 -*-

# portfolio.py

from __future__ import print_function

import csv
import datetime
from math import floor
try:
    import Queue as queue
except ImportError:
    import queue

import numpy as np
import pandas as pd
from data import HistoricCSVDataHandler
from event import FillEvent, OrderEvent
from performance import
create_sharpe_ratio,create_CAGR,Success_Ratio,Average_Profit_Average_Loss,Standard_De
viation,create_drawdowns


class Portfolio(object):
    """
    The Portfolio class handles the positions and market
    value of all instruments at a resolution of a "bar",
    i.e. secondly, minutely, 5-min, 30-min, 60 min or EOD.

    The positions DataFrame stores a time-index of the
    quantity of positions held.

    The holdings DataFrame stores the cash and total market
    holdings value of each symbol for a particular
    time-index, as well as the percentage change in
    portfolio total across bars.
    """

#    def __init__(self, bars, events, start_date, initial_capital=100000.0):
    def __init__(self, bars, events, start_date, pair, initial_capital=100000.0):
        """
        Initialises the portfolio with bars and an event queue.
        Also includes a starting datetime index and initial capital
        (USD unless otherwise stated).

        Parameters:
        bars - The DataHandler object with current market data.
        events - The Event Queue object.
        start_date - The start date (bar) of the portfolio.
        initial_capital - The starting capital in USD.
        """
        self.bars = bars
        self.events = events
        self.symbol_list = self.bars.symbol_list
```

```
        self.start_date = start_date
        self.initial_capital = initial_capital

        self.pair = pair

        self.all_positions = self.construct_all_positions()
        self.current_positions = dict( (k,v) for k, v in [(s, 0) for s in
self.symbol_list] )

        self.all_holdings = self.construct_all_holdings()
        self.current_holdings = self.construct_current_holdings()

    def construct_all_positions(self):
        """
        Constructs the positions list using the start_date
        to determine when the time index will begin.
        """
        d = dict( (k,v) for k, v in [(s, 0) for s in self.symbol_list] )
        d['datetime'] = self.start_date
        return [d]

    def construct_all_holdings(self):
        """
        Constructs the holdings list using the start_date
        to determine when the time index will begin.
        """
        d = dict( (k,v) for k, v in [(s, 0.0) for s in self.symbol_list] )
        d['datetime'] = self.start_date
        d['cash'] = self.initial_capital
        d['commission'] = 0.0
        d['total'] = self.initial_capital
        return [d]

    def construct_current_holdings(self):
        """
        This constructs the dictionary which will hold the instantaneous
        value of the portfolio across all symbols.
        """
        d = dict( (k,v) for k, v in [(s, 0.0) for s in self.symbol_list] )
        d['cash'] = self.initial_capital
        d['commission'] = 0.0
        d['total'] = self.initial_capital
        return d

    def update_timeindex(self, event):
        """
        Adds a new record to the positions matrix for the current
        market data bar. This reflects the PREVIOUS bar, i.e. all
        current market data at this stage is known (OHLCV).

        Makes use of a MarketEvent from the events queue.
        """
        latest_datetime = self.bars.get_latest_bar_datetime(self.symbol_list[0])

        # Update positions
```

```
        # ================
        dp = dict( (k,v) for k, v in [(s, 0) for s in self.symbol_list] )
        dp['datetime'] = latest_datetime

        for s in self.symbol_list:
            dp[s] = self.current_positions[s]

        # Append the current positions
        self.all_positions.append(dp)

        # Update holdings
        # ===============
        dh = dict( (k,v) for k, v in [(s, 0) for s in self.symbol_list] )
        dh['datetime'] = latest_datetime
        dh['cash'] = self.current_holdings['cash']
        dh['commission'] = self.current_holdings['commission']
        dh['total'] = self.current_holdings['cash']

        for s in self.symbol_list:
            # Approximation to the real value
            market_value = self.current_positions[s] * \
                self.bars.get_latest_bar_value(s, "adj_close")
            dh[s] = market_value
            dh['total'] += market_value

        # Append the current holdings
        self.all_holdings.append(dh)

    # ======================
    # FILL/POSITION HANDLING
    # ======================

    def update_positions_from_fill(self, fill):
        """
        Takes a Fill object and updates the position matrix to
        reflect the new position.

        Parameters:
        fill - The Fill object to update the positions with.
        """
        # Check whether the fill is a buy or sell
        fill_dir = 0
        if fill.direction == 'BUY':
            fill_dir = 1
        if fill.direction == 'SELL':
            fill_dir = -1

        # Update positions list with new quantities
        self.current_positions[fill.symbol] += fill_dir*fill.quantity

    def update_holdings_from_fill(self, fill):
        """
        Takes a Fill object and updates the holdings matrix to
        reflect the holdings value.
```

```
        Parameters:
        fill - The Fill object to update the holdings with.
        """
        # Check whether the fill is a buy or sell
        fill_dir = 0
        if fill.direction == 'BUY':
            fill_dir = 1
        if fill.direction == 'SELL':
            fill_dir = -1

        # Update holdings list with new quantities
        fill_cost = self.bars.get_latest_bar_value(
            fill.symbol, "adj_close"
        )
        cost = fill_dir * fill_cost * fill.quantity
        self.current_holdings[fill.symbol] += cost
        self.current_holdings['commission'] += fill.commission
        self.current_holdings['cash'] -= (cost + fill.commission)
        self.current_holdings['total'] -= (cost + fill.commission)

    def update_fill(self, event):
        """
        Updates the portfolio current positions and holdings
        from a FillEvent.
        """
        if event.type == 'FILL':
            self.update_positions_from_fill(event)
            self.update_holdings_from_fill(event)

    def generate_naive_order(self, signal):
        """
        Simply files an Order object as a constant quantity
        sizing of the signal object, without risk management or
        position sizing considerations.

        Parameters:
        signal - The tuple containing Signal information.
        """
        order = None

        symbol = signal.symbol
        direction = signal.signal_type
        strength = signal.strength

#        p0 = self.pair[0]
#        p1 = self.pair[1]
#
#        p0_price = self.bars.get_latest_bar_value(p0,"adj_close")
#        p1_price = self.bars.get_latest_bar_value(p1,"adj_close")

        symbol_price = self.bars.get_latest_bar_value(symbol,"adj_close")

#        print('Price of %s at %f' %(p0,p0_price))
#        print('Price of %s at %f' %(p1,p1_price))
```

```
#        mkt_quantity = np.floor((self.initial_capital /(p0_price+p1_price))/2.0)
        mkt_quantity = np.floor((self.initial_capital /(symbol_price*strength))/2.0)
#        print('Price of %s at %f -> %i - Strength %f'
%(symbol,symbol_price,mkt_quantity,strength))
#        mkt_quantity = 100
#        mkt_quantity = mkt_quantity*2
#        mkt_quantity = mkt_quantity*3
        cur_quantity = self.current_positions[symbol]
        order_type = 'MKT'

        if direction == 'LONG' and cur_quantity == 0:
            order = OrderEvent(symbol, order_type, mkt_quantity, 'BUY')
        if direction == 'SHORT' and cur_quantity == 0:
            order = OrderEvent(symbol, order_type, mkt_quantity, 'SELL')
        if direction == 'EXIT' and cur_quantity > 0:
            order = OrderEvent(symbol, order_type, abs(cur_quantity), 'SELL')
        if direction == 'EXIT' and cur_quantity < 0:
            order = OrderEvent(symbol, order_type, abs(cur_quantity), 'BUY')

#        print(
#            "Order: Symbol=%s, Direction=%s, Quantity=%s" %
#            (symbol, direction, cur_quantity)
#        )

        return order

    def update_signal(self, event):
        """
        Acts on a SignalEvent to generate new orders
        based on the portfolio logic.
        """
        if event.type == 'SIGNAL':
            order_event = self.generate_naive_order(event)
            self.events.put(order_event)

    # ========================
    # POST-BACKTEST STATISTICS
    # ========================

    def create_equity_curve_dataframe(self):
        """
        Creates a pandas DataFrame from the all_holdings
        list of dictionaries.
        """
        curve = pd.DataFrame(self.all_holdings)
        curve.set_index('datetime', inplace=True)
        curve['returns'] = curve['total'].pct_change()
        curve['equity_curve'] = (1.0+curve['returns']).cumprod()
        self.equity_curve = curve

#    def output_summary_stats(self):
    def output_summary_stats(self,csv_file):
        """
        Creates a list of summary statistics for the portfolio.
        """
```

```
        total_return = self.equity_curve['equity_curve'][-1]
        returns = self.equity_curve['returns']
        pnl = self.equity_curve['equity_curve']

#         returns
#         print(returns)

        sharpe_ratio = create_sharpe_ratio(returns)
        drawdown, max_dd, dd_duration = create_drawdowns(pnl)
        self.equity_curve['drawdown'] = drawdown
        CAGR = create_CAGR(pnl)
        Std_Dev = Standard_Deviation(returns)

#         print(pd.DataFrame(self.all_positions))
#         print(pnl)
        curve = pd.DataFrame(self.all_holdings)
        curve.set_index('datetime', inplace=True)
#         print(curve)
        comission = (curve['commission']/curve['commission'].shift()-1).dropna()
#         comission = (curve.pct_change()).dropna()
#         print(comission)
#         Num_Trades = comission[comission > 0.0].count()
#         print(Num_Trades)
#         print(comission[comission > 0.0])

        Succ_Ratio = Success_Ratio(pnl,comission)
        Aver_Profit_Aver_Loss=Average_Profit_Average_Loss(pnl,comission)

        stats = [("Total Return", "%0.2f%%" % ((total_return - 1.0) * 100.0)),
                 ("Sharpe Ratio", "%0.2f" % sharpe_ratio),
                 ("Max Drawdown", "%0.2f%%" % (max_dd * 100.0)),
                 ("Drawdown Duration (days)", "%d" % dd_duration),
                 ("CAGR","%0.2f%%" % (CAGR*100.0)),
                 ("Standard Deviation Ret", "%0.2f%%" % (Std_Dev*100.0)),
                 ("Success Ratio", "%0.2f%%" % (Succ_Ratio*100)),
                 ("Average Profit_Average Loss", "%0.2f" % (Aver_Profit_Aver_Loss))]

        with open("stats_"+csv_file,'wb') as resultFile:
            wr = csv.writer(resultFile, dialect='excel')
            wr.writerows(stats)

#         self.equity_curve.to_csv('equity.csv')
        self.equity_curve.to_csv(csv_file)
        return stats
```

Strategy.py

```python
#!/usr/bin/python
# -*- coding: utf-8 -*-

# strategy.py

from __future__ import print_function

from abc import ABCMeta, abstractmethod
import datetime
try:
    import Queue as queue
except ImportError:
    import queue

import numpy as np
import pandas as pd

from event import SignalEvent


class Strategy(object):
    """
    Strategy is an abstract base class providing an interface for
    all subsequent (inherited) strategy handling objects.

    The goal of a (derived) Strategy object is to generate Signal
    objects for particular symbols based on the inputs of Bars
    (OHLCV) generated by a DataHandler object.

    This is designed to work both with historic and live data as
    the Strategy object is agnostic to where the data came from,
    since it obtains the bar tuples from a queue object.
    """

    __metaclass__ = ABCMeta

    @abstractmethod
    def calculate_signals(self):
        """
        Provides the mechanisms to calculate the list of signals.
        """
        raise NotImplementedError("Should implement calculate_signals()")
```

Contact Us

QuantInsti Quantitative Learning Pvt. Ltd.

India: A-309, Boomerang, Chandivali Farm Road, Powai, Mumbai, India - 400072 Contact:

+91-22- 61691400, +91 9920448877

Singapore: 30 Cecil Street, #19-08, Prudential Tower, Singapore – 049712

+65-90578301

Website: www.quantinsti.com

Email: contact@quantinsti.com

>>>Follow Us