Dynamic Common Trends (DCT)

ABSTRACT

Our strategy screens for cointegrated pairs within S&P 500 with a superior filter methodology for signal extraction. The strategy is composed of three steps: First, the cointegration test identifies long-horizon relations of a stock pair using the two-step Engle and Granger method within each sector. Second, the strategy is executed based on a unique dynamic hedging ratio using Kalman Filter. Last, the trade is monitored through the Hurst exponent daily to ensure the significance of the time series. Compared to a simple OLS-regression pairs trading, the relationship in pairs is estimated dynamically and the advanced filter methodology significantly improves the speed and power of the market signal extraction. Over a 5-year test period, the strategy achieved 7.48% in CGAR with an absolute return of 60% and a Sharpe Ratio of 3.41.

I BACKGROUND

In the current financial landscape, the markets are perceived to react based on actions (or lack of actions) taken by the Federal Reserve. With the Fed's purchase of assets through quantitative easing, interest rates have been on a downwards trend and in much of the developed world are near 0. As a result of the low interest rates, many retail and institutional investors are looking for higher yields than those offered in the bond markets and are turning to the stock market instead. Based on research, in months with large balance sheet increases by the Fed, the US equity market has tended to outperform.[1] With the great financial crisis of 2008 to Covid-19 today, the Fed has taken an increasingly active approach in the economy. As only one example, almost 24% of all dollars were created in the past year.[2] From February 1st, 2008, to February 1st, 2021, the Fed balance sheet has gone from approximately $871 billion to $7.4 trillion dollars.[3] We believe this influx of dollars into the economy in addition to the Fed's large balance sheet is potentially creating inflated valuations in the stock market. As a result, we want to create a trading strategy that is neither bullish nor bearish, as the extent to which the Fed is able to continue to support the markets in the future is unknown. As a consequence, we take a market-neutral approach through our pairs-trading model. We wanted to run a long and short book and can do both with our cointegrated pairs. Through our approach, we recognize the law of one price over the long-term and have implemented the trading strategy by exploiting discrepancies in the short term. That is, there are short-term discrepancies that we exploit through arbitrage opportunities.

II ECONOMIC HYPOTHESIS & NOVELTY

Living under this near-zero interest rate environment, asset prices are detached from their long-term fundamentals and we hold the assumption that this detachment short-term and the asset prices will eventually reveal

[1] Hassett, Kevin and Kevin A Hassett. 08/25/2014. "A Fed-Blown Bubble?" *National Review (New York)* 66 (15): 8.
[2] https://www.thestreet.com/mishtalk/economics/23-6-of-all-us-dollars-were-created-in-the-last-year
[3] https://www.federalreserve.gov/monetarypolicy/bst_recenttrends.htm

their fundamental values. Our strategy is aimed at maximizing the portfolio return by extracting trading opportunities from these temporary market divergences.

Pairs trading takes advantage of the symmetry of transient price deviations between two assets to capture Alpha returns on both assets, with the central assumption that the linear combination between the paired assets is mean reverting. Pairs trading is also a market-neutral trading strategy. Under the law of one price, differences in the prices of paired assets will quickly shrink and the response of price to value causes the individual risk of paired assets to be directly converted into individual return. This yield is independent of the market.

While someone might doubt the novelty of the strategy, compared to the simple OLS or cointegration test, we add an advanced filter methodology that significantly improves the speed and power of the market signal extraction.

II IMPLEMENTATION

A. Data and Universe Selection

This strategy is developed based on the quantitative research platforms, QuantConnect, using Python. The universe we selected is the S&P 500 individual stocks for its liquidity and execution purposes. Compared to the MSCI, stocks within S&P 500 have greater liquidity: with less restriction and more availability for shorting and bulk buying, while compared to ETFs, though having fewer stabilities, the S&P 500 produces more stationary pairs that enhance the power of this strategy. The time horizon we chose is from Nov. 1st, 2016 till Feb. 1st, 2021, which is after President Trump's admission and includes the market impact of COVID-19.

We break down the S&P 500 into 11 sectors: consumer discretionary, communication services, energy, financials, healthcare, industrials, information technology, materials, real estates, consumer staples, and utility. Instead of analyzing all possible pairs within the S&P 500, the sector approach helps develop a stronger fundamental basis for the cointegrated pairs and provides flexibility for the analysts to implement their macro views into the sectors' weights and adjust accordingly for their investment constraints. However, for the purpose of unbiased backtesting, we assume the constant sectors' weights as the sectors' weights for S&P as of Nov. 1st, 2016, the starting date for our testing period.

B. Determine the Sectors' Weight

For the sectors' weight, our approach is a simple approximation to the industry weight of the S&P 500 for the purpose of unbiased backtesting. However, this strategy can also be very flexible in terms of integrating analyst's insight into the sectors' weight selection.

C. Cointegrated Pairs Screener

For each sector, we begin by screening our universe of assets for cointegrated securities using the two-step Engle and Granger method; by definition, two non-stationary stochastic time series processes – in this case, the price series of assets can be used to form a linear combination, such that the linear combination is stationary, which implies the process to be mean reverting, i.e. the error corrects itself. To verify that the price spread is stationary and mean reverting, we employ two statistical tests:

1. The Augmented Dickey-Fuller test, which is a test for the presence of a unit root in time series data; when a time series process lacks a unit root, this ensures that the process is stationary and mean-reverting.

2. The Hurst Exponent test is a statistical measure that classifies the long-term memory of a time series process; if the Hurst Exponent is less than one half, the time series process is mean reverting.

Take the information technology sector as an example, within the 65 stocks (i.e. 2145 possible pairs) within the Information technology sector, after analyzing 1067 rows of daily market close data, we identify 88 possible pairs (*Fig. 1 Code Snippet: All identified pairs within S&P Info Tech; Fig. 2 Time series price data for all stocks within the Information Technology Sector, 2016 - 2021*). For example, ACN and CRM is identified as a stable pair that passed both tests as shown in the time series chart of their price data (*Fig. 3*) and the correlation chart (*Fig. 4*).

D. Dynamic Hedging Execution Algorithm

After testing that the price spread is stationary and mean-reverting, the next step in our process is to estimate the speed and half-life of the mean reverting process. The half-life can be defined as the time for the expected value of the price spread to reach the intermediate price between the current value and the long run mean, which the mean reverting process is converging to in the long run. The half-life is a measure of the mean-reversion process; therefore, it is an alternative parameter to the reversion speed, as there is a direct relation between the half-life and the speed. The price spread follows an arithmetic mean-reverting process also known as an Arithmetic Ornstein-Uhlenbeck process. The speed of the mean-reversion process can be estimated from historical data, allowing for a straightforward calculation of the estimated half-life.

After calculating the half-life, the next step is to normalize the price spread to produce a metric that can be used to identify price spreads that have diverged from their long run mean. We augment the normalization process by using a rolling Z-score with a window that is equal to the half-life of the price spread; using a rolling Z-score enables us to measure the anomalousness of the price spread data within the half-life window. This method increases the efficacy of identifying positive or negative divergence from the mean price spread.

The type of divergence (positive or negative) will indicate which security to buy and sell; if the spread has positively diverged from its mean, this indicates that we want to sell/short asset Y and buy asset X. If the spread has negatively diverged from its mean, this indicates that we want to buy asset Y and sell/short asset X. The standard Z-score we used is 1.414 based on Fibonacci root. Still take ACN and CRM as our example: Fig. 5 shows their changes in the rolling Z-score. Whenever the Z-score passes the green line (i.e. the threshold), the trade will be

executed and closely monitored daily both by the Hurst component and the rolling Z-Score window (trade will be closed when the Z-Score converges back to zero.)

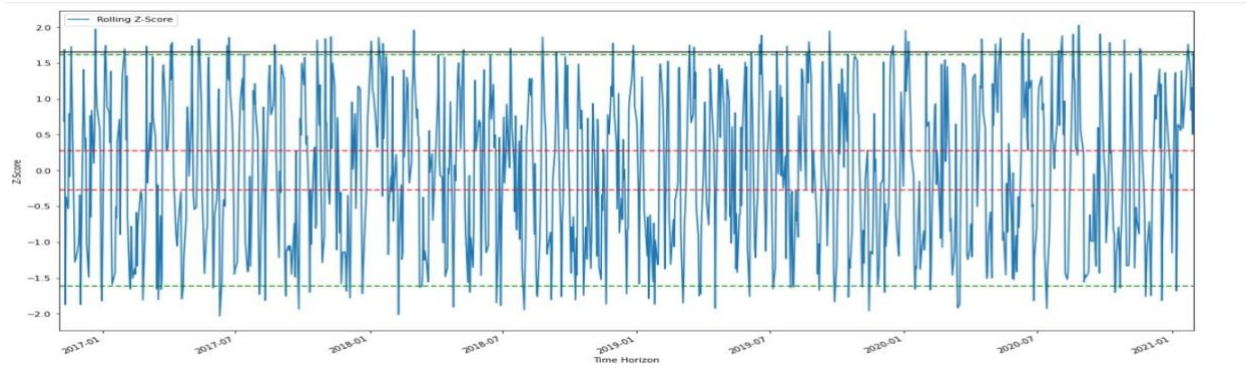

Figure 5: Rolling Z-Score of ACN and CRM

To ensure market neutrality, we utilize a dynamic hedging ratio, which is determined by a Kalman filtering process with a Bayesian approach. The theoretical basis is developed from a state space model, with the general premise being that a set of states evolve over time, but the observations of those states contain statistical noise, meaning that it is not possible to directly observe the "true" states. Therefore, the goal of the state space model is to infer information about the states, given the observations, as new information arrives, hence the Bayesian approach. The Kalman filtering approach, also known as linear quadratic estimation, tends to be more accurate than those based on a single measurement alone; this is achieved by estimating a joint probability distribution over the variables for each timeframe. The graph below (*Fig. 6*) shows the changes of the price spread with Dynamic hedging ratio for *ACN and CRM.*

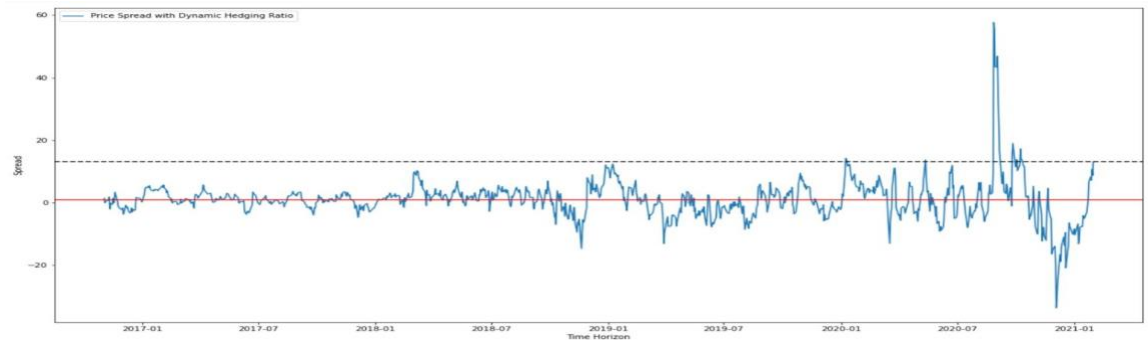

Fig. 6 Price Spread with Dynamic Hedging Ratio, ACN and CRM

E. Sector Portfolio and Total Portfolio Results Calculation

After the dynamic hedging execution algorithm is executed within each sector, the portfolio return is calculated by the weighted sum of the return from each sector. Please refer to Part V for the backtesting and analysis discussion.

III RISK ANALYSIS:

*A. **Execution Risk:** Is the short trade always available for all of our stocks?*

One of the most significant risks of any pairs trading strategies is the availability of short selling for any particular stocks. When the market is volatile or with limited buyers holding that securities, we might not be able to find the lender in the market with the amount we need. The absence of shorting shares or insufficiency to meet the hedge ratio will result in the ineffectiveness of the pairs strategy and broken price spread.

This execution risk also depends on two factors: the universe we choose and scalability of the fund. For our universe, S&P 500, it is a relatively liquid market with sufficient short selling shares in most cases. During the backtesting of our model, only one pair was not able to source sufficient shares due to a lack of borrow availability. With respect to scalability, of course as the scale of the strategy is increased, one would expect to see diminishing returns from arbitrage opportunities (assuming an inverse relationship between the dollars attributed to the strategy and the number of securities that diverge from their mean spread.)

***Model Risk:** A risk that occurs when the actual return does not perform as the model predicts.*

This can be due to anything from inaccurate research to flawed logic or calculations. For example, pairs trading is based on the assumption that the two stocks will return to their corresponding relationship after any divergence, but this does not mean that they will continue to be correlated in the future, and it is difficult to tell how the two stocks will necessarily be correlated. For example, an event at Long Term Capital Management (LTCM) known as the "Collapse" was caused by the model risk.

IV LIQUIDITY AND CAPITAL CONSIDERATION

Since we're looking at S&P 500 stocks, liquidity shouldn't be a major concern. The overall market is the most liquid in the world. Because of this universe choice and the assumption of a mega hedge fund with connections to various brokers, as well as the large amount of stocks we will be holding, our strategy can be deployed at least for $10 million with a minimum impact on the market, i.e. our VWAP will still be within expectation.

Our strategy can also be extended to any other universe, but there might be liquidity risk associated with that: for example, while on the NYSE there are dedicated order specialists who would have to buy/sell from their own inventory if no one else was buying/selling that stock, the NASDAQ doesn't have that, so we would have to wait for another customer to be on the other side of the trade. However, since these are some of the biggest companies, we don't expect large risks from this. Moreover, if we expanded into less liquid/international markets this may pose a bigger challenge of executing long/short orders.

V ANALYSIS OF STRATEGY

We conducted the backtesting on the quant research platform, Quantconnect, using Python. We set the time horizon from Nov. 1st, 2016 to Feb. 1, 2021 with $10 million initial investment. Fig. 7 shows the code snippet of the execution. Within the 11 sectors, the Communications sector has the highest absolute return of 74%, while the Utilities sector has the lowest absolute return of 23%. For example, below shows the equity return curve for the information technology sector, which is the sector with the highest weight in S&P 500. Over a 5-year horizon, the strategy achieved an average 7.48% in CGAR per sector and a Sharpe Ratio of 3.41 if assuming the same initial sector weight as the S&P 500's market capitalization. With an initial sector allocation approximated to the S&P 500 industry weight, the total portfolio absolute return is 60% since Nov. 2016. If we assume an equal-weighted initial capital allocation per section, the total return is 56%. The detailed table of weight and return of each sector is on the table (Fig. 8).

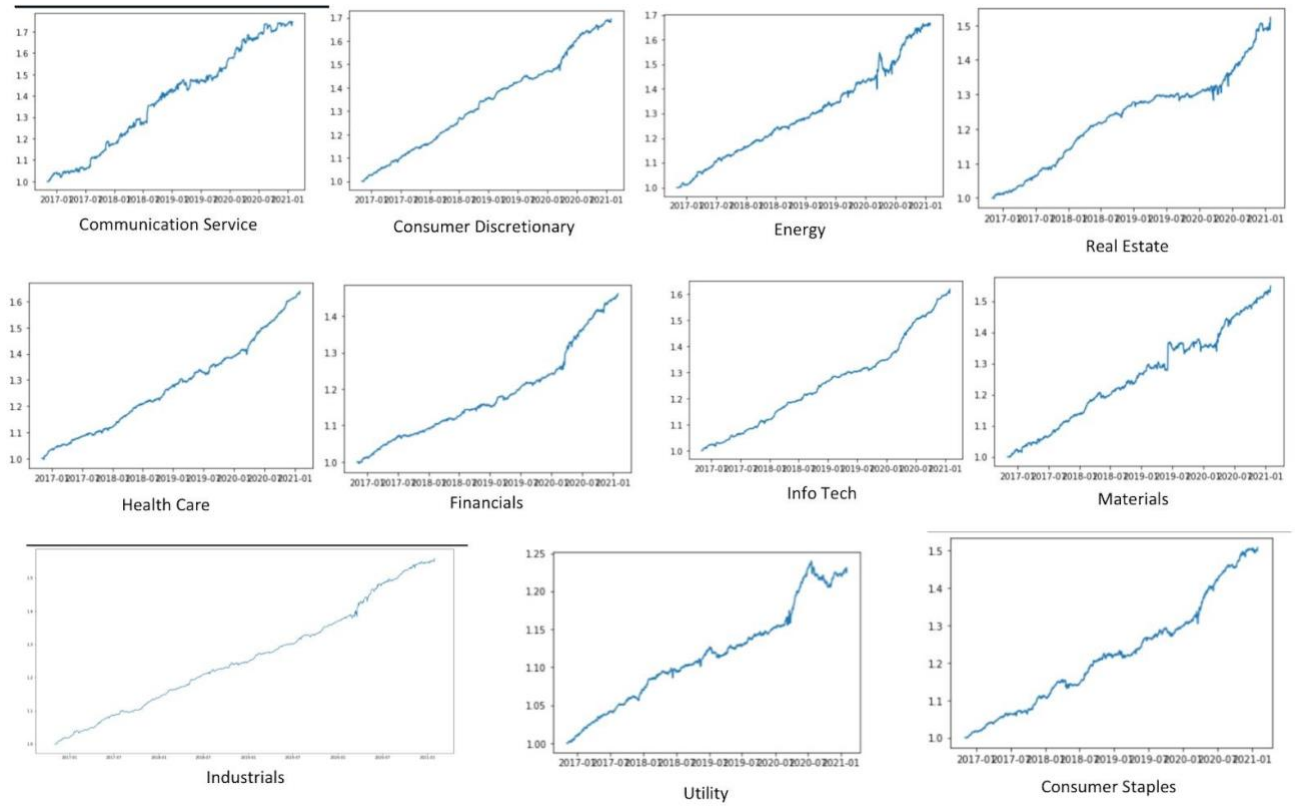

Fig. 8 Equity return per sector, from Nov. 1, 2016 to Feb. 1, 2021

We also adjusted the absolute return of the portfolio by the transaction cost. Bowen, Hutchinson and Sullivan (2010) discussed the estimated transaction cost of pairs trade to be around 2% - 5%. [4] Therefore, with a conservative 3% of the transaction cost discounted from the annual CGAR of the strategy, our final CGAR of this strategy is around 4.93%.

[4] Bowen, David, et al. "High-Frequency Equity Pairs Trading:Transaction Costs, Speed of Execution, and Patterns in Returns." *The Journal of Trading*, vol. 5, no. 3, 2010, pp. 31–38., doi:10.3905/jot.2010.5.3.031.

VI APPENDIX

```
We found 88.0 pairs.
[('ACN S6HA7SVNXLYD', 'CRM SZQUJUA9SVOL', 0.025948294
 ('ACN S6HA7SVNXLYD', 'ORCL R735QTJ8XC9X', 0.04226283
 ('ADI R735QTJ8XC9X', 'KEYS VV9PLPU5EI5H', 0.03145135
 ('ADI R735QTJ8XC9X', 'ORCL R735QTJ8XC9X', 0.04213970
 ('ADS S5CWVYVQSODH', 'CEY S67FQV7Z62CL', 0.045873030
 ('ADS S5CWVYVQSODH', 'FTNT UHQJ0EDGU6HX', 0.02514488
 ('ADS S5CWVYVQSODH', 'GPN S1VV43FB2UZP', 0.019770393
 ('ADSK R735QTJ8XC9X', 'AMD R735QTJ8XC9X', 0.00920793
```

Fig. 1 Code Snippet: part of identified pairs (total 88 pairs) within S&P Info Tech, Nov. 1 2016 - Feb. 1 2021

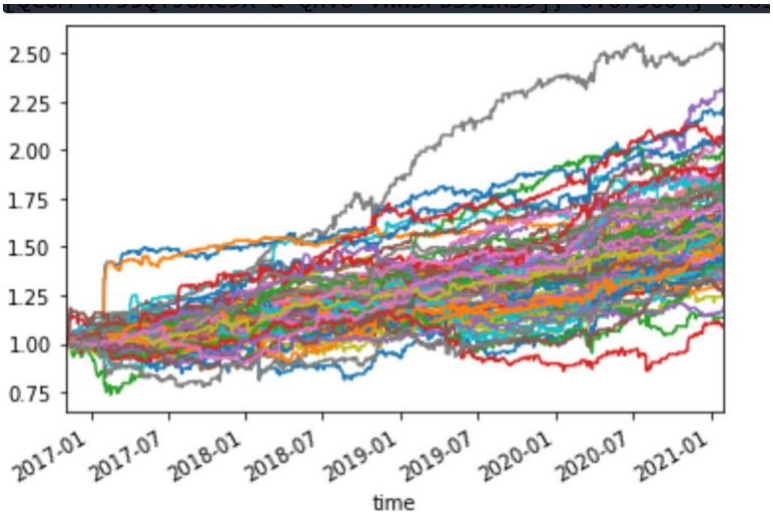

Fig. 2 Time series price data for all stocks within the Information Technology Sector, 2016 - 2021

Fig. 3 Time Series Data of ACN and CRM

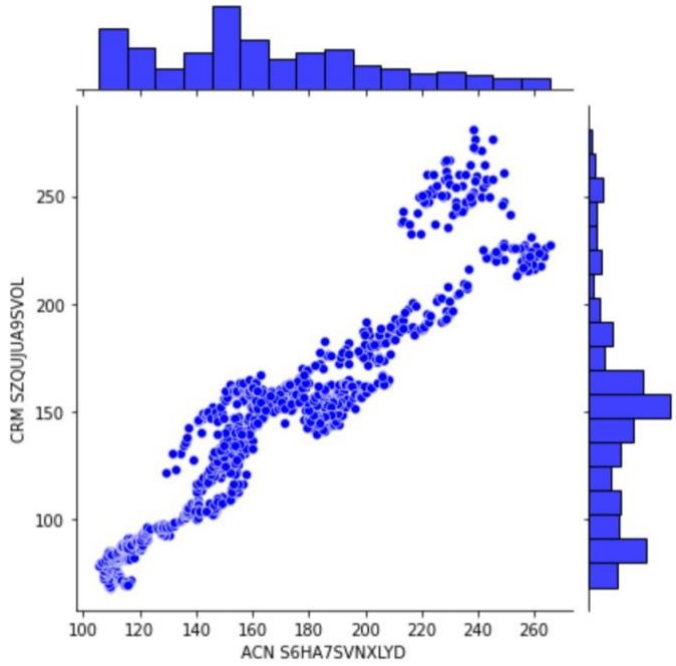

Fig. 4 correlation plot between ACM and CRM

```
# trading logic
entryZscore = 1.414
exitZscore = 0

#set up num units long
df1['long entry'] = ((df1.zScore < - entryZscore) & ( df1.zScore.shift(1) > - entryZscore))
df1['long exit'] = ((df1.zScore > - exitZscore) & (df1.zScore.shift(1) < - exitZscore))
df1['num units long'] = np.nan
df1.loc[df1['long entry'],'num units long'] = 1
df1.loc[df1['long exit'],'num units long'] = 0
df1['num units long'][0] = 0
df1['num units long'] = df1['num units long'].fillna(method='pad')

#set up num units short
df1['short entry'] = ((df1.zScore > entryZscore) & (df1.zScore.shift(1) < entryZscore))
df1['short exit'] = ((df1.zScore < exitZscore) & (df1.zScore.shift(1) > exitZscore))
df1.loc[df1['short entry'],'num units short'] = -1
df1.loc[df1['short exit'],'num units short'] = 0
df1['num units short'][0] = 0
df1['num units short'] = df1['num units short'].fillna(method='pad')

df1['numUnits'] = df1['num units long'] + df1['num units short']
df1['spread pct ch'] = (df1['spread'] - df1['spread'].shift(1)) / ((df1['x'].shift(1) * abs(df1['hr'].shift(1))) + df1['y'].shift(1))
df1['port rets'] = df1['spread pct ch'] * df1['numUnits'].shift(1)

df1['cum rets'] = df1['port rets'].cumsum()
df1['cum rets'] = df1['cum rets'] + 1
```

Fig. 7 Code Snippet: Trading Logic

Sector	Sharpe Ratio	CGAR	Absolute Return
Consumer Disc.	4.48	8.95%	69%
Energy	2.29	8.64%	66%
Financial	3.64	6.36%	46%
Healthcare	4.83	8.37%	64%
Industrials	4.43	7.49%	56%
Info. Tech.	4.98	8.14%	62%
Materials	2.45	7.37%	55%
Real Estate	2.82	7.08%	52%
Consumer Staples	3.16	6.91%	51%
Utility	2.32	3.44%	23%
Communications	2.17	9.48%	74%
Sector Average	**3.415454545**	**7.48%**	**56%**

Fig 9 Table of Backtesting Results Per Sector

Sector	Absolute Return	Sector Weight (S&P Market Cap Weight Approx.)	Sector Weight (equal weighted)
Consumer Disc.	69%	12.70%	9.09%
Energy	66%	2.33%	9.09%
Financial	46%	10.34%	9.09%
Healthcare	64%	13.44%	9.09%
Industrials	56%	8.47%	9.09%
Info. Tech.	62%	27.60%	9.09%
Materials	55%	2.64%	9.09%
Real Estate	52%	2.41%	9.09%
Consumer Staples	51%	6.55%	9.09%
Utility	23%	2.73%	9.09%
Communications	74%	11%	9.09%
Checker		100%	100%
Portfolio Absolute Return		**60%**	**56%**

Fig 9 Table of Backtesting Results: Calculation of Total Portfolio Return

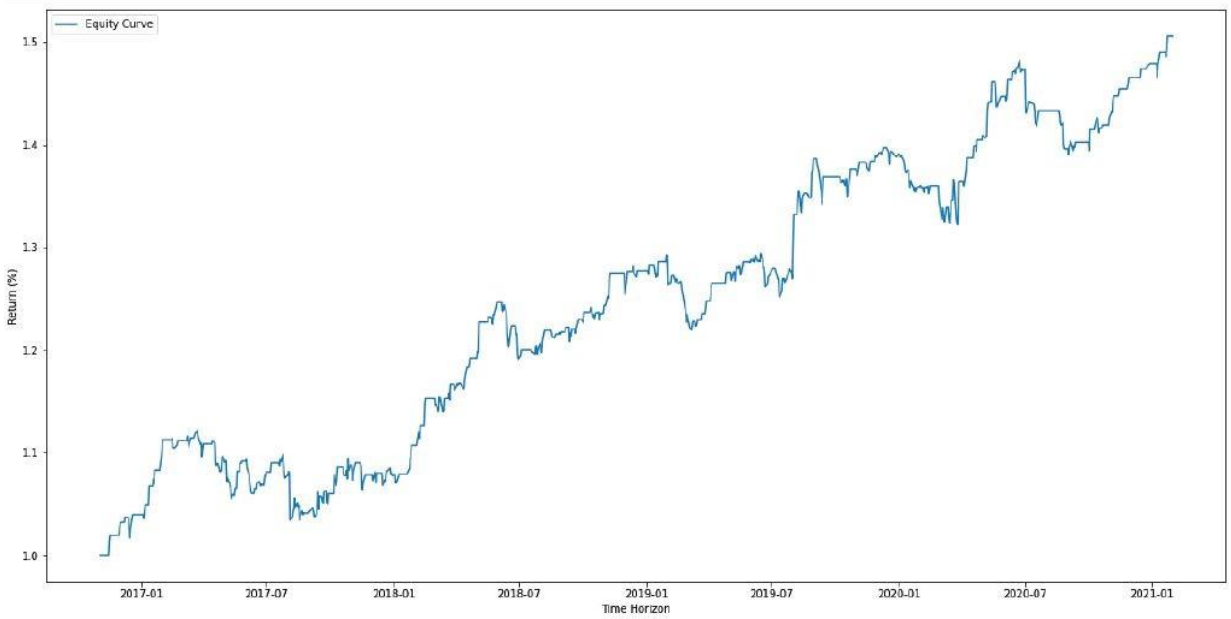

Fig. 10 Individual Pair Equity Curve if ACN and CRM's pair trade

```
# https://quantpedia.com/Screener/Details/12
/*
Read Me:

Data Structure:
pairs (list)
hisotry_price: "ticker" deque

*/

import clr
clr.AddReference("System")
clr.AddReference("QuantConnect.Algorithm")
clr.AddReference("QuantConnect.Common")

from System import *
from QuantConnect import *
from QuantConnect.Algorithm import *
import numpy as np
import pandas as pd
from scipy import stats
from math import floor
from datetime import timedelta
from collections import deque
import itertools as it # iteration tools for it.combine
from decimal import Decimal

class DynamicPairsTradingAlgorithm(QCAlgorithm):
    # Algorithm reference needs to include Initialize and OnDatq
    def Initialize(self):
        # initialize the price data
        # train historical price
        self.SetStartDate(2016,1,1)
        self.SetEndDate(2021,1,1)
        self.SetCash(100000)
        tickers = ["A", "ABBV","ABC", "ABMD", "ABT", "AGN", "ALGN", "ALXN", "AMGN", "ANTM", "BAX", "BDX", "BIIB", "BMY",
"BSX", "CAH", "CELG", "CERN", "CI", "CNC", "COO", "CVS", "DGX", "DHR", "DVA", "EW", "GILD", "HCA", "HOLX", "HSIC",
"HUM", "IDXX", "ILMN", "INCY", "IQV", "ISRG", "JNJ", "LH", "LLY", "MCK", "MDT", "MRK", "MTD", "MYL", "NKTR", "PFE",
"PKI", "PRGO", "REGN", "RMD", "SYK", "TFX", "TMO", "UHS", "UNH", "VAR", "VRTX", "WAT", "WCG", "XRAY", "ZBH", "ZTS"]
        self.threshold = 2
        self.symbols = []
        for i in tickers:
            self.symbols.append(self.AddEquity(i, Resolution.Daily).Symbol)

        self.pairs = {}
        ## TODO: what is our training/formation period?
        self.formation_period = 252

        self.history_price = {} #data structure = {"ticker":deque}

        # data cleaning + load the data
        for symbol in self.symbols:
            # pull out hisotrical data
            hist = self.History([symbol], self.formation_period+1, Resolution.Daily)
            if hist.empty:
                self.symbols.remove(symbol) # remove it if there is not enough data
            else: #double ended queue
                # select how long we wil train
                self.history_price[str(symbol)] = deque(maxlen=self.formation_period)
                for tuple in hist.loc[str(symbol)].itertuples():
                    self.history_price[str(symbol)].append(float(tuple.close)) # select only the close price data
                if len(self.history_price[str(symbol)]) < self.formation_period: # if not enough data then delete
                    self.symbols.remove(symbol)
                    self.history_price.pop(str(symbol))

        # generate pairs
```

```
        self.symbol_pairs = list(it.combinations(self.symbols, 2))
        # this will give all combination of symbols

        # Add the benchmark
        self.AddEquity("SPY", Resolution.Daily)

        # Schedule the events: trigger the code to run at specific times of day
        # this event will start at the firs tradable day of SPY each month, and at the time of SPY trade

        ##TODO: is our algorithm gonna be trading daily??
        ##Reference: https://www.quantconnect.com/docs/algorithm-reference/scheduled-events

        self.Schedule.On(self.DateRules.MonthStart("SPY"), self.TimeRules.AfterMarketOpen("SPY"), self.DynamicHedging)
        self.count = 0
        self.sorted_pairs = None


    def OnData(self, Slice):
        # Update the price series everyday
        # Slice means time slice: time and value of a particular time
        # Slice["IBM"] will give a TradeBar of IBM
        ##Reference: https://www.quantconnect.com/docs/algorithm-reference/handling-data
        for symbol in self.symbols:
            # if the total dataset contains the symbol key in our ticker, we will append the daily data to the hisotry price
            if Slice.Bars.ContainsKey(symbol) and str(symbol) in self.history_price:
                self.history_price[str(symbol)].append(float(Slice[symbol].Close))
        if self.sorted_pairs is None:
            return


        for i in self.sorted_pairs:
            # calculate the spread of two price series
            spread = np.array(self.history_price[str(i[0])]) - np.array(self.history_price[str(i[1])])
            mean = np.mean(spread)
            std = np.std(spread)
            ratio = self.Portfolio[i[0]].Price / self.Portfolio[i[1]].Price
            # long-short position is opened when pair prices have diverged by two standard deviations
            if spread[-1] > mean + self.threshold * std:
                if not self.Portfolio[i[0]].Invested and not self.Portfolio[i[1]].Invested:
                    quantity = int(self.CalculateOrderQuantity(i[0], 0.2))
                    self.Sell(i[0], quantity)
                    self.Buy(i[1],  floor(ratio*quantity))

            elif spread[-1] < mean - self.threshold * std:
                quantity = int(self.CalculateOrderQuantity(i[0], 0.2))
                if not self.Portfolio[i[0]].Invested and not self.Portfolio[i[1]].Invested:
                    self.Sell(i[1], quantity)
                    self.Buy(i[0], floor(ratio*quantity))

            # the position is closed when prices revert back
            elif self.Portfolio[i[0]].Invested and self.Portfolio[i[1]].Invested:
                    self.Liquidate(i[0])
                    self.Liquidate(i[1])


    def DynamicHedging(self):
        # schedule the event to fire every half year to select pairs with the smallest historical distance
        if self.count % 6 == 0:
            distances = {}
            for i in self.symbol_pairs:
                distances[i] = Pair(i[0], i[1], self.history_price[str(i[0])],  self.history_price[str(i[1])]).distance()
                self.sorted_pairs = sorted(distances, key = lambda x: distances[x])[:4]
        self.count += 1

class Pair:
    def __init__(self,a,b):
```

```
        self.a = a
        self.b = b
        self.name = str(a) + ':' + str(b)
        self.df = pd.concat([a.df,b.df],axis = 1).dropna()
        self.num_bar = self.df.shape[0]
        self.cor = self.df.corr().ix[0][1]
        self.error = 0
        self.last_error = 0
        self.a_price = []
        self.a_date = []
        self.b_price = []
        self.b_date = []

    def cor_update(self):
        self.cor = self.df.corr().ix[0][1]

    def price_record(self,data_a,data_b):
        self.a_price.append(float(data_a.Close))
        self.a_date.append(data_a.EndTime)
        self.b_price.append(float(data_b.Close))
        self.b_date.append(data_b.EndTime)

    def df_update(self):
        new_df = pd.DataFrame({str(self.a):self.a_price,str(self.b):self.b_price},index = [self.a_date]).dropna()
        self.df = pd.concat([self.df,new_df])
        self.df = self.df.tail(self.num_bar)
        for i in [self.a_price,self.a_date,self.b_price,self.b_date]:
            i = []

    def distance(self):
        # calculate the sum of squared deviations between two normalized price series
        norm_a = np.array(self.price_a)/self.price_a[0]
        norm_b = np.array(self.price_b)/self.price_b[0]
        return sum((norm_a - norm_b)**2)

    def zscore(series):
        return (series - series.mean()) / np.std(series)

    def hurst(ts):
        lags = range(2, 100)
        tau = [sqrt(std(subtract(ts[lag:], ts[:-lag]))) for lag in lags]
        poly = polyfit(log(lags), log(tau), 1)
        return poly[0]*2.0

    def check_for_stationarity(X, cutoff=0.05): #this will return a boolean
    # H_0 in adfuller is unit root exists (non-stationary)
    # We must observe significant p-value to convince ourselves that the series is stationary
        pvalue = adfuller(X)[1]
        if pvalue < cutoff:
            print ('The series ' + X.name +' is likely stationary.')
            return True
        else:
            print ('The series ' + X.name +' is likely non-stationary.')
            return False

    def find_cointegrated_pairs(data, significance=0.05):
        n = data.shape[1]
        score_matrix = np.zeros((n, n))
        pvalue_matrix = np.ones((n, n))
        keys = data.keys()
        pairs = []
        for i in range(n):
            for j in range(i+1, n):
                S1 = data[keys[i]]
                S2 = data[keys[j]]
                result = coint(S1, S2, autolag=None)
```

```
            score = result[0]
            pvalue = result[1]
            score_matrix[i, j] = score
            pvalue_matrix[i, j] = pvalue
            if pvalue < significance:
                pairs.append((keys[i], keys[j], pvalue))
    return pairs

def KalmanFilterAverage(x):
  # Construct a Kalman filter
    kf = KalmanFilter(transition_matrices = [1],
    observation_matrices = [1],
    initial_state_mean = 0,
    initial_state_covariance = 1,
    observation_covariance=1,
    transition_covariance=.01)

  # Use the observed values of the price to get a rolling mean
    state_means, _ = kf.filter(x.values)
    state_means = pd.Series(state_means.flatten(), index=x.index)
    return state_means

# Kalman filter regression
def KalmanFilterRegression(x,y):
    delta = 1e-3
    trans_cov = delta / (1 - delta) * np.eye(2) # How much random walk wiggles
    obs_mat = np.expand_dims(np.vstack([[x], [np.ones(len(x))]]).T, axis=1)

    kf = KalmanFilter(n_dim_obs=1, n_dim_state=2, # y is 1-dimensional, (alpha, beta) is 2-dimensional
    initial_state_mean=[0,0],
    initial_state_covariance=np.ones((2, 2)),
    transition_matrices=np.eye(2),
    observation_matrices=obs_mat,
    observation_covariance=2,
    transition_covariance=trans_cov)

    # Use the observations y to get running estimates and errors for the state parameters
    state_means, state_covs = kf.filter(y.values)
    return state_means

def half_life(spread):
    spread_lag = spread.shift(1)
    spread_lag.iloc[0] = spread_lag.iloc[1]
    spread_ret = spread - spread_lag
    spread_ret.iloc[0] = spread_ret.iloc[1]
    spread_lag2 = sm.add_constant(spread_lag)
    model = sm.OLS(spread_ret,spread_lag2)
    res = model.fit()
    halflife = int(round(-np.log(2) / res.params[1],0))

    if halflife <= 0:
        halflife = 1
    return halflife
```

```
import numpy as np
import pandas as pd
import datetime as datetime
import statsmodels.formula.api as sm
from pandas.core import datetools
import statsmodels.tsa.stattools as ts
from pair import *

class PairsTrading(QCAlgorithm):

    def __init__(self):
        self.symbols = [
        'ASB', 'AF', 'BANC', 'BBVA', 'BBD', 'BCH', 'BLX', 'BSBR', 'BSAC', 'SAN',
        'CIB', 'BXS', 'BAC', 'BOH', 'BMO', 'NTB', 'BK', 'BNS', 'BKU', 'BCS', 'BBT',
        'BFR', 'CM', 'COF', 'C', 'CFG', 'CMA', 'CBU', 'CPF', 'BAP', 'CFR', 'CUBI',
        'DKT', 'DB', 'EVER', 'FNB', 'FBK', 'FCB', 'FBP', 'FCF', 'FHN', 'FBC', 'FSB',
        'GWB', 'AVAL', 'BSMX', 'SUPV', 'HDB', 'HTH', 'HSBC', 'IBN', 'ING', 'ITUB', 'JPM',
        'KB', 'KEY', 'LYG', 'MTB', 'BMA', 'MFCB', 'MSL', 'MTU', 'MFG', 'NBHC', 'OFG', 'PNC',
        'PVTD', 'PB', 'PFS', 'RF', 'RY', 'RBS', 'SHG', 'STT', 'STL', 'SCNB', 'SMFG', 'STI',
        'SNV', 'TCB', 'TD', 'USB', 'UBS', 'VLY', 'WFC', 'WAL', 'WBK', 'WF', 'YDKN', 'ZBK']
        self.data_resolution = 10
        self.num_bar = 6.5*60*60/(self.data_resolution)
        self.one_month = 6.5*20*60/(self.data_resolution)
        self.selected_num = 10
        self.pair_num = 120
        self.pair_threshod = 0.88
        self.BIC = -3.34
        self.count = 0
        self.pair_list = []
        self.selected_pair = []
        self.trading_pairs = []
        self.generate_count = 0
        self.open_size = 2.32
        self.close_size = 0.5
        self.stop_loss = 6
        self.data_count = 0

    def Initialize(self):
        self.SetStartDate(2016,1,1)
        self.SetEndDate(2018,5,1)
        self.SetCash(50000)

        for i in range(len(self.symbols)):
            equity = self.AddEquity(self.symbols[i],Resolution.Minute).Symbol
            self.symbols[i] = equity
            self.symbols[i].prices = []
            self.symbols[i].dates = []

    def generate_pairs(self):
        for i in range(len(self.symbols)):
            for j in range(i+1,len(self.symbols)):
                self.pair_list.append(pairs(self.symbols[i],self.symbols[j]))

        self.pair_list = [x for x in self.pair_list if x.cor > self.pair_threshod]

        self.pair_list.sort(key = lambda x: x.cor, reverse = True)

        if len(self.pair_list) > self.pair_num:
            self.pair_list = self.pair_list[:self.pair_num]


    def pair_clean(self,list):
        l = []
        l.append(list[0])
        for i in list:
            symbols = [x.a for x in l] + [x.b for x in l]
```

```
        if i.a not in symbols and i.b not in symbols:
            l.append(i)
        else:
            pass
    return l

def OnData(self,data):
    if not self.Securities[self.symbols[0]].Exchange.ExchangeOpen:
        return
    #data aggregation
    if self.data_count < self.data_resolution:
        self.data_count +=1
        return
    # refill the initial df
    if len(self.symbols[0].prices) < self.num_bar:
        for i in self.symbols:
            if data.ContainsKey(i) is True:
                i.prices.append(float(data[i].Close))
                i.dates.append(data[i].EndTime)
            else:
                self.Log('%s is missing'%str(i))
                self.symbols.remove(i)
        self.data_count = 0
        return
    # generate paris
    if self.count == 0 and len(self.symbols[0].prices) == self.num_bar:
        if self.generate_count == 0:
            for i in self.symbols:
                i.df = pd.DataFrame(i.prices, index = i.dates, columns = ['%s'%str(i)])

            self.generate_pairs()
            self.generate_count +=1


        self.Log('pair list length:'+str(len(self.pair_list)))
    # correlation selection
        for i in self.pair_list:
            i.cor_update()
    # updatet the dataframe and correlation selection
        if len(self.pair_list[0].a_price) != 0:
            for i in self.pair_list:
                i.df_update()
                i.cor_update()

        self.selected_pair = [x for x in self.pair_list if x.cor > 0.9]
    # cointegration selection
        for i in self.selected_pair:
            i.cointegration_test()

        self.selected_pair = [x for x in self.selected_pair if x.adf < self.BIC]
        self.selected_pair.sort(key = lambda x: x.adf)

        if len(self.selected_pair) == 0:
            self.Log('no selected pair')
            self.count += 1
            return

        self.selected_pair = self.pair_clean(self.selected_pair)
        for i in self.selected_pair:
            i.touch = 0
            self.Log(str(i.adf) + i.name)

        if len(self.selected_pair) > self.selected_num:
            self.selected_pair = self.selected_pair[:self.selected_num]
```

```
            self.count +=1
            self.data_count = 0
            return

        #update the pairs
        if self.count != 0 and self.count < self.one_month:

            num_select = len(self.selected_pair)

            for i in self.pair_list:
                if data.ContainsKey(i.a) is True and data.ContainsKey(i.b) is True:
                    i.price_record(data[i.a],data[i.b])
                else:
                    self.Log('%s has no data'%str(i.name))
                    self.pair_list.remove(i)

        ## selected pairs

            for i in self.selected_pair:
                i.last_error = i.error

            for i in self.trading_pairs:
                i.last_error = i.error

        ## enter
            for i in self.selected_pair:
                price_a = float(data[i.a].Close)
                price_b = float(data[i.b].Close)
                i.error = price_a - (i.model.params[0] + i.model.params[1]*price_b)
                if (self.Portfolio[i.a].Quantity == 0 and self.Portfolio[i.b].Quantity == 0) and i not in self.trading_pairs:
                    if i.touch == 0:
                        if i.error < i.mean_error - self.open_size*i.sd and i.last_error > i.mean_error - self.open_size*i.sd:
                            i.touch += -1
                        elif i.error > i.mean_error + self.open_size*i.sd and i.last_error < i.mean_error + self.open_size*i.sd:
                            i.touch += 1
                        else:
                            pass
                    elif i.touch == -1:
                        if i.error > i.mean_error - self.open_size*i.sd and i.last_error < i.mean_error - self.open_size*i.sd:
                            self.Log('long %s and short %s'%(str(i.a),str(i.b)))
                            i.record_model = i.model
                            i.record_mean_error = i.mean_error
                            i.record_sd = i.sd
                            self.trading_pairs.append(i)
                            self.SetHoldings(i.a, 5.0/(len(self.selected_pair)))
                            self.SetHoldings(i.b, -5.0/(len(self.selected_pair)))
                            i.touch = 0
                    elif i.touch == 1:
                        if i.error < i.mean_error + self.open_size*i.sd and i.last_error > i.mean_error + self.open_size*i.sd:
                            self.Log('long %s and short %s'%(str(i.b),str(i.a)))
                            i.record_model = i.model
                            i.record_mean_error = i.mean_error
                            i.record_sd = i.sd
                            self.trading_pairs.append(i)
                            self.SetHoldings(i.b, 5.0/(len(self.selected_pair)))
                            self.SetHoldings(i.a, -5.0/(len(self.selected_pair)))
                            i.touch = 0
                    else:
                        pass
                else:
                    pass

            # close
            for i in self.trading_pairs:
                if data.ContainsKey(i.a) and data.ContainsKey(i.b):
                    price_a = float(data[i.a].Close)
```

```
                price_b = float(data[i.b].Close)
                i.error = price_a - (i.record_model.params[0] + i.record_model.params[1]*price_b)
                if ((i.error < i.record_mean_error + self.close_size*i.record_sd and i.last_error >i.record_mean_error +
self.close_size*i.record_sd)
                    or (i.error > i.record_mean_error - self.close_size*i.record_sd and i.last_error <i.record_mean_error -
self.close_size*i.record_sd)):
                    self.Log('close %s'%str(i.name))
                    self.Liquidate(i.a)
                    self.Liquidate(i.b)
                    self.trading_pairs.remove(i)
                elif i.error < i.record_mean_error - self.stop_loss*i.record_sd or i.error > i.record_mean_error +
self.stop_loss*i.record_sd:
                    self.Log('close %s to stop loss'%str(i.name))
                    self.Liquidate(i.a)
                    self.Liquidate(i.b)
                    self.trading_pairs.remove(i)
                else:
                    pass




        self.count +=1
        self.data_count = 0
        return

    if self.count == self.one_month:
        self.count = 0
        self.data_count = 0
        return


import numpy as np
import pandas as pd
import datetime as datetime
import statsmodels.formula.api as sm
import statsmodels.tsa.stattools as ts

class pairs(object):

    def __init__(self,a,b):
        self.a = a
        self.b = b
        self.name = str(a) + ':' + str(b)
        self.df = pd.concat([a.df,b.df],axis = 1).dropna()
        self.num_bar = self.df.shape[0]
        self.cor = self.df.corr().ix[0][1]
        self.error = 0
        self.last_error = 0
        self.a_price = []
        self.a_date = []
        self.b_price = []
        self.b_date = []

    def cor_update(self):
        self.cor = self.df.corr().ix[0][1]

    def cointegration_test(self):
        self.model = sm.ols(formula = '%s ~ %s'%(str(self.a),str(self.b)), data = self.df).fit()
        self.adf = ts.adfuller(self.model.resid,autolag = 'BIC')[0]
        self.mean_error = np.mean(self.model.resid)
        self.sd = np.std(self.model.resid)

    def price_record(self,data_a,data_b):
        self.a_price.append(float(data_a.Close))
        self.a_date.append(data_a.EndTime)
```

```
        self.b_price.append(float(data_b.Close))
        self.b_date.append(data_b.EndTime)

    def df_update(self):
        new_df = pd.DataFrame({str(self.a):self.a_price,str(self.b):self.b_price},index = [self.a_date]).dropna()
        self.df = pd.concat([self.df,new_df])
        self.df = self.df.tail(self.num_bar)
        for i in [self.a_price,self.a_date,self.b_price,self.b_date]:
            i = []
```