Pairs Trading with Machine Learning on Distributed Python Platform

CAPSTONE

Submitted in Partial Fulfillment of

the Requirements for

the Degree of

Master of Science (Finance and Risk Engineering)

at the

NEW YORK UNIVERSITY
TANDON SCHOOL OF ENGINEERING

by

Yicheng Wang

May 2019

Acknowledgements

I would like to extend my sincere gratitude to Prof. Song Tang for his expert guidance and constructive feedback throughout the course of this project. I appreciate the patience with which he would tackle my doubts and the time he would spare from his busy schedule to meet during the course of his work-day to discuss. His inputs to this project are invaluable. I would also like to thank the Finance and Risk Engineering department at NYU Tandon School of Engineering to give me the opportunity to work on this advanced topic for my capstone project, during the course of which I learned a lot.

Yicheng Wang

May 2019

ABSTRACT

This capstone project implements a distributed Python platform that can be used to test quantitative models for trading financial instruments in a network setting under client/server infrastructure. Normally, we backtest locally using past historical data to check the performance of our trading strategies. The performance result, in this case, is usually an illusion of what the actual performance is in real-time trading. We also show in this paper this conclusion by showing that our quantitative trading model performs much worse in the simulated trading than that in backtesting environment. Therefore, we build this Python platform not only for implementing trading strategies and backtesting them historically but also for simulating trades similar to what is in real market, acting as another control before real-time trading.

Contents

Chapter 1

Introduction

This capstone project does the following several things coded in Python: it first sets market data retrieval using Unicorn data feed and parses market data in json format to store market data for backtesting in SQLite database; then implements trading logic; sets up Python Client/Server communication and multi-threading and implements real-time feed to simulate real trade; finally displays trading analysis and PnL on web dashboard. The program has to run in order as described. The program design is also shown in flow chart Figure 1.1.

The server is a multi-thread application which coordinates among all the client applications. Its main purposes are 1) messaging among all the participants, 2) maintain a market participant list, and the list of stocks traded by participants, and 3) generate a consolidated order book for all the participants. The client application, also a multi-threading application in network-oriented environment, will communicate with the server. Each application will implement required

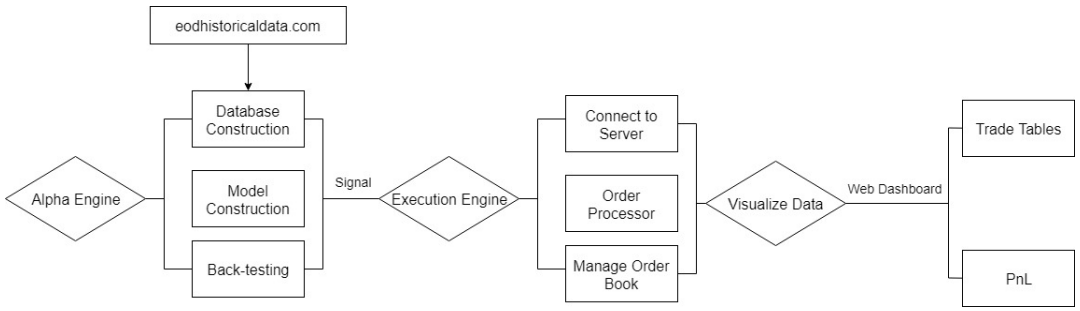

Figure 1.1: Program Design

messages in json format.

Moreover, 1) The sender and receiver threads will support TCP/IP protocol and through Internet sockets; 2) The sender and receiver threads must achieve data synchronization using event and queue; 3) the application will handle direct market data feeds in json format for historical and real-time data from eodhistorical-data.com; 4) the application will have an integrated database for data persistence. The database we use is sqlite3. We implement tables and SQL statements for our own model buildup, as well as back testing. 5) We will implement a trading model. 6) We manage our own order books and calculate P/L.

The trading model is a pairs trading model on stocks in SP500. We use machine learning techniques in selecting pairs. Specifically, we use PCA to reduce dimensions and then DBSCAN clustering to group stocks. We then identify pairs within clusters to implement dollar neutral Bollinger Band pairs trading strategy. Finally, we construct a portfolio with pairs equally weighted. We achieve a 2.5 Sharpe ratio backtested in 2018. However, we can potentially lose money when we trade in simulated market using client/server infrastructure we built.

We proceed as follows. In Chapter 2: Background, we provide some backgrounds in pairs trading model and machine learning techniques we used. Chapter 3: Data, we describe how we retrieve, parse and store data into database. In Chapter 4: Trading Model, we describe our trading logic, how we train, build and backtest our model. In Chapter 5: Client/Server, we provide details of client server infrastructure and how they communicate. In Chapter 6: Simulated Trade, we explain how we implement trading strategy to trade in client/server infrastructure. In Chapter 7: Performance Analysis, we show our performance of backtest and simulated trade results and how we move the visualization to web dashboard using flask. At last, we have Chapter 8: Conclusion and appendix where source codes will be provided.

Chapter 2

Background

2.1 Pairs Trading

Pairs trading is a market-neutral trading strategy that matches a long position with a short position in a pair of highly correlated instruments such as two stocks, exchange-traded funds (ETFs), currencies, commodities or options. Pairs traders wait for weakness in the correlation and then go long the under-performer while simultaneously short selling the over-performer, closing the positions as the relationship returns to statistical norms.[6]

2.1.1 Cointegration Test

In order to find pairs to trade in pairs trading, we look for cointegrated relationship in pairs. A pair is cointegrated if individual instruments are of same order and the linear combination of them is stationary. We test stationarity of the linear combination of the pair, say y_t. Stationarity is referred as weak stationarity in this case that a series is stationary if the mean and autocovariance are independent of time and the variance is finite for all times.[13] We simply regress y_t on its lagged values y_{t-1} and find out whether the coefficient ϕ is 1 or not. Consider autoregressive process of order 1, AR(1), as shown:[11]

$$y_t = \phi y_{t-1} + \epsilon_t \tag{2.1}$$

where ϵ_t is white noise error term with mean zero and constant variance. Then,

$$\Delta y_t = \delta y_{t-1} + \epsilon_t \tag{2.2}$$

where Δ is first difference and $\delta = \phi - 1$. If $\delta = 0$, then $\Delta y_t = \epsilon_t$, it means y_t is random walk and non-stationary. Otherwise, it is a stationary process. This is called Dickey-Fuller Test.

Moreoever, we have Augmented Dickey-Fuller (ADF) Test, which allows for higher-order autoregressive processes, as shown:[11]

$$\Delta y_t = \alpha + \beta t + \gamma y_{t-1} + \sum_{j=1}^{p} \delta_j \Delta y_{t-j} + \epsilon_t \tag{2.3}$$

For this model, we test for the null hypothesis as $H_0 : \gamma = 0$ against alternative hypothesis as $H_1 : \gamma < 0$. We reject the null if the test statistic is smaller than critical value of a significance level.

In this project, we use ADF test to check cointegration of two stock price series. We construct linear combination of two stock price series as:

$$y_t = S_t^1 - aS_t^2 \tag{2.4}$$

where S_t^1 and S_t^2 are price series of stock 1 and stock 2.

2.1.2 Bollinger Band Strategy

Bollinger band strategy is typically used in pairs trading to capture profit between upper and lower bands. Upper and lower band are constructed 1-2 standard deviations from the moving average of the series y_t.[5] There are also many ways of entry and exit, long and short. In our trading model, we long when y_t crosses down the lower band, short when y_t crosses up the upper band, where the profit is captured between buy and sell points, as shown in Figure 2.1.

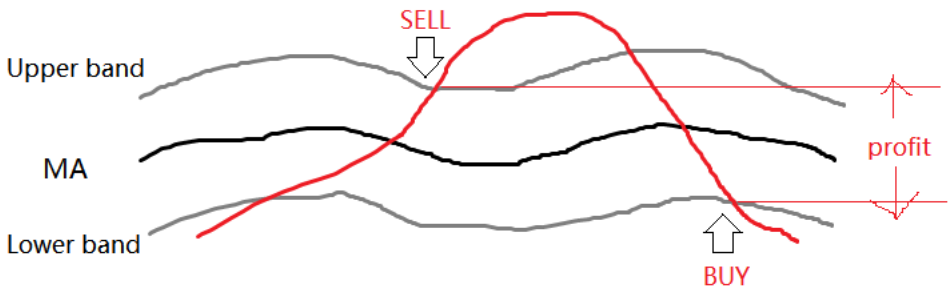

Figure 2.1: Bollinger Band

2.2 Machine Learning

2.2.1 Principle Component Analysis

Principle Component Analysis (PCA) is used for reducing the number of variables comprising a dataset while retaining the variability in the data, or identifying hidden patterns in the data, and classifying them according to how much of the information, stored in the data, they account for.[1] We use PCA for our stock market data for both purposes.

We take stock price panel data as score matrix A, and coefficient vector l that generates linear combination on A which yields principle components Y of smaller dimension than A. Principle components Y are independent and each coefficient vector l_i is required to maximize the variance of its corresponding principle component Y_i, as shown:[1]

$$\max Var(Y_i) = l_i^T C l_i \tag{2.5}$$

$$s.t.$$

$$||l_i^T|| = 1, \forall i$$

$$l_i^T l_j = 0, \forall j \neq i$$

which has Lagrangian form:

$$L(l_i, \lambda_i, \delta) = l_i^T C l_i - \lambda_i (l_i^T l_i - 1) - \delta (l_i^T l_i) \tag{2.6}$$

Maximizing L by taking partial derivatives to 0, we obtain $Cl_i = \lambda_i l_i$, where C is covariance matrix of A and λ_i, l_i are corresponding eigenvalues and eigenvectors.

Therefore, principle components are constructed from eigenvectors of score matrix with score matrix itself: $Y_i = l_i^T A$.

2.2.2 DBSCAN Clustering

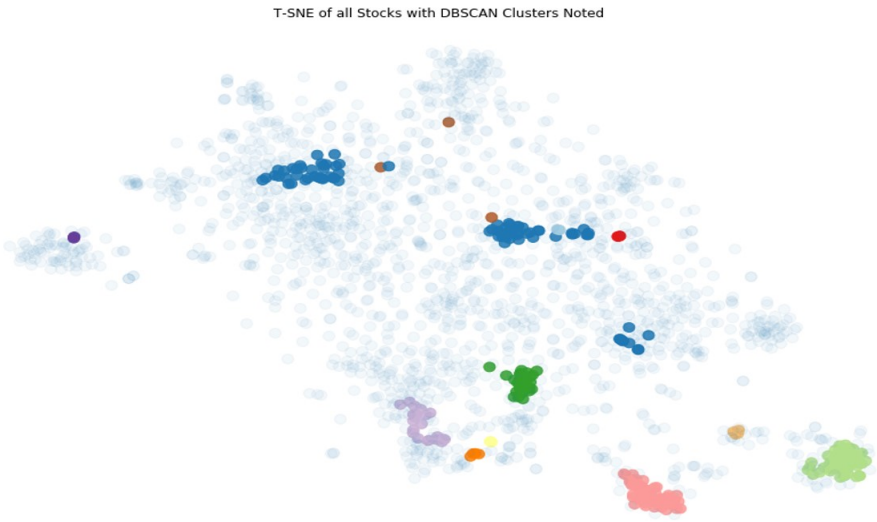

Figure 2.2: DBSCAN Clustering

Density-based spatial clustering of applications with noise (DBSCAN) is a density-based clustering non-parametric algorithm: given a set of points in some space, it groups together points that are closely packed together (points with many nearby neighbors).[12] Unlike normal clustering method that group all points, DBSCAN does not group outliers that lie alone in low-density regions. Another characteristic of DBSCAN is that it is likely to produce different clusters at each run. An example is shown in Figure 2.1.

DBSCAN has two important hyperparameters. Minimum sample size is minimum number of points in a group. Distance parameter is largest distance between points in a group. We pick $\epsilon = 1.8, minsamples = 3$ to get a reasonable number of resulting clusters, which is about 8 clusters.

Chapter 3

Data

3.1 Data Information

3.1.1 Data Tables

1. "TickerName": stock market data from eodhistoricaldata and are in json format, shown in figure 3.2 includes symbol, date, open, high, low, close, adjusted close, volume from 2014-01-01 to recent, where only adjusted close price is used in our trading model.

2. "sp500": SP500 constituent data from pkgstore.datahub.io in json format, shown in figure 3.1, which includes name, sector and symbol.

3. "stockpairs": stock pairs constituent data from training model, which includes ticker1 and ticker2 in pairs, score that represent the strength of cointegration, profit and loss for the pair in backtesting.

4. "pairprices": stock pairs price data with adjusted close price of each pair from stock market data, residuals from pair prices regression, and bollinger band constructed from residuals.

5. "trades": trades data that record pair, date, price, quantity and PnL of trade status everyday.

	name	sector	symbol
	Filter	Filter	Filter
1	Agilent Techn...	Health Care	A
2	American Airli...	Industrials	AAL
3	Advance Auto...	Consumer Dis...	AAP
4	Apple Inc.	Information T...	AAPL
5	AbbVie Inc.	Health Care	ABBV

Figure 3.1: Table: SP500 Constituents

	symbol	date	open	high	low	close	adjusted_close	volume
	Filter	Filter	Filter	Filter	Filter	Filter	Filter	Filter
1	A	2014-01-02	57.1	57.1	56.15	56.21	38.2757	1916160
2	A	2014-01-03	56.39	57.345	56.26	56.92	38.7592	1866651
3	A	2014-01-06	57.4	57.7	56.56	56.64	38.5686	1777472
4	A	2014-01-07	56.95	57.63	56.93	57.45	39.1201	1461048
5	A	2014-01-08	57.33	58.54	57.17	58.39	39.7602	2659468

Figure 3.2: Table: Stock A

3.1.2 Data Retrieval

We first put SP500 constituent data into database, then retrieve data from each ticker from SP500 and also SP500 index price.

Function get_daily_data() takes in ticker name, start date, end date, data url, api key to get market data for one stock starting from start date to end date. Python packpage urlopen is used to open url and data is parsed from json format to dictionary then to pandas dataframe in Function download_stock_data(). Then, the dataframe is stored in SQLite in this function. These functions are shown in Figure 3.3 and 3.4.

```
def get_daily_data(symbol='', start=data_start_date, end=data_end_date, requestType=requestURL,
                   apiKey=myEodKey, completeURL=None):
    if not completeURL:
        symbolURL = str(symbol) + '?'
        startURL = "from=" + str(start)
        endURL = "to=" + str(end)
        apiKeyURL = "api_token=" + myEodKey
        completeURL = requestURL + symbolURL + startURL + '&' + endURL + '&' + apiKeyURL + '&period=d&fmt=json'

    # if cannot open url
    try:
        with urllib.request.urlopen(completeURL) as req:
            data = json.load(req)
            return data
    except:
        pass
```

Figure 3.3: Function: get_daily_data()

```
' populate stock data for each stock '
def download_stock_data(ticker, metadata, engine, table_name):
    column_names = ['symbol','date','open','high','low','close','adjusted_close','volume']
    price_list = []
    clear_a_table(table_name, metadata, engine)

    if 'GSPC' not in ticker:
        symbol_full = str(ticker) + ".US"
        stock = get_daily_data(symbol=symbol_full)
    else:
        stock = get_daily_data(symbol=ticker)

    if stock:
        for stock_data in stock:
            price_list.append([str(ticker), stock_data['date'], stock_data['open'], stock_data['high'],
                               stock_data['low'], stock_data['close'], stock_data['adjusted_close'],
                               stock_data['volume']])

    stocks = pd.DataFrame(price_list, columns=column_names)
    stocks.to_sql(table_name, con=engine, if_exists='replace', index=False, chunksize=5)
```

Figure 3.4: Function: download_stock_data()

3.2 Database Design

Our database design comprises of 504 tables with two entity relationship, as shown in Figure 3.1. First, each of 500 ticker database with ”symbol” and ”date” as primary key and ”symbol” as foreign key maps to ”symbol” as primary key from sp500 constituents database. Second, table ”pairprices” and table ”trades” both have ”symbol1”, ”symbol2” that reference to ”ticker1”, ”ticker2” in table stockpairs.

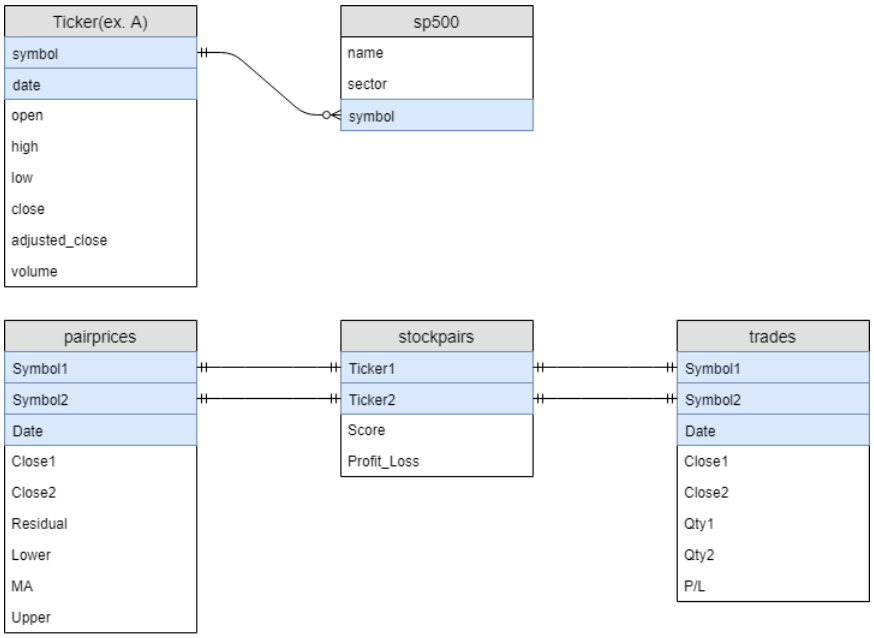

Figure 3.5: Database Design

Chapter 4

Trading Model

Our trading model is pairs trading model. Pairs are constructed from cointegrated stocks in SP500 stocks in function training_model(). Then, each pairs is to implement bollinger band strategy in function building_model(). Later, our trading model will enter function backtesting() with trading signals and corresponding PnL.

4.1 Training

4.1.1 Parameters

- training start date: 2014-01-01, training end date: 2018-01-01
- capital: 1,000,000 for each pair
- cointegration: significance = 0.05
- Bollinger Band: standard deviation = 2, moving average period = 10
- PCA: principle components = 50, epsilon = 1.8, minimum sample = 3

4.1.2 Steps

1. 500 stocks price return data of 4 years are reduced to 50 principle components using PCA (background in Section 2.2.1). The resulting dataframe is then in 50 * 500 dimension.

```
'identify cointegrated pairs from clusters'
def Cointegration(cluster, significance, start_day, end_day):
    pair_coin = []
    p_value = []
    adf = []
    n = cluster.shape[0]
    keys = cluster.keys()
    for i in range(n):
        for j in range(i+1,n):
            asset_1 = Price.loc[start_day:end_day, keys[i]]
            asset_2 = Price.loc[start_day:end_day, keys[j]]
            results = sm.OLS(asset_1, asset_2)
            results = results.fit()
            predict = results.predict(asset_2)
            error = asset_1 - predict
            ADFtest = ts.adfuller(error)
            if ADFtest[1] < significance:
                pair_coin.append([keys[i], keys[j]])  # pair names
                p_value.append(ADFtest[1])  # p value, smaller the better
                adf.append(ADFtest[0])  # adf test stats, larger the better
    return p_value, pair_coin, adf
```

Figure 4.1: Function: Cointegration()

2. We apply DBSCAN Clustering (background in 2.2.2) to group stocks according to 50 principle components. It will result in 6 to 10 groups with minimum of 3 stocks per group.

3. Pairs are selected within each group. In each group, every two stocks go through cointegration test (background in 2.1.1). As shown in Figure 4.1, only pairs that pass the test most significantly (smallest p-value, or largest t-statistics) in each group are selected. Information about pairs are stored in table "stockpairs". As shown in Figure 4.2, all pairs have large score (t-statistics).

4. We then regress one stock price series to another stock price series for each pair using training data. Residual is then constructed in this way as in Equation (2.4). Later, we build bollinger band using residuals (background in Section 2.1.2). Pairs prices, residuals and bollinger bands data are stored in table "pairprices".

4.2 Backtesting

We backtest from 2018-01-01 to 2019-01-01 for one year. A class is constructed for each stock pair shown in Figure 4.3, where trade is created with stock pair data of each day and is updated if there is a trading signal. Each pair is assigned

	Ticker1	Ticker2	Score
	Filter	Filter	Filter
1	AEP	ES	3.5184
2	AFL	AON	2.8846
3	HBAN	NTRS	3.7143
4	CL	KO	3.1849

Figure 4.2: Table: Stockpairs

with equal capital and is market neutral. Profit and loss is also calculated daily according current position and price. Backtesting result is stored in table "trades".

```
class StockPair:

    def __init__(self, symbol1, symbol2, start_date, end_date):
        self.ticker1 = symbol1
        self.ticker2 = symbol2
        self.start_date = start_date
        self.end_date = end_date
        self.trades = {}
        self.total_profit_loss = 0.0

    def __str__(self):
        return str(self.__class__) + ": " + str(self.__dict__) + "\n"

    def __repr__(self):
        return str(self.__class__) + ": " + str(self.__dict__) + "\n"

    def createTrade(self, date, close1, close2, res, lower, upper, qty1 = 0, qty2 = 0, profit_loss = 0.0):
        self.trades[date] = np.array([close1, close2, res, lower, upper, qty1, qty2, profit_loss])

    def updateTrades(self):  # dollar neutral, available dollar for buy/sell for each pair
        trades_matrix = np.array(list(self.trades.values()))

        for index in range(1, trades_matrix.shape[0]):
            # RES SELL SIGNAL: buy asset 1, sell asset 2
            if (trades_matrix[index-1, 2] < trades_matrix[index-1, 4] and
                trades_matrix[index, 2] > trades_matrix[index, 4]):
                trades_matrix[index, 5] = int(capital / trades_matrix[index, 0])
                trades_matrix[index, 6] = int(-capital / trades_matrix[index, 1])
            # RES BUY SIGNAL: sell asset 1, buy asset 2
            elif (trades_matrix[index-1, 2] > trades_matrix[index-1, 3] and
                trades_matrix[index, 2] < trades_matrix[index, 3]):
                trades_matrix[index, 5] = int(-capital / trades_matrix[index, 0])
                trades_matrix[index, 6] = int(capital / trades_matrix[index, 1])
            # no act
            else:
                trades_matrix[index, 5] = trades_matrix[index-1, 5]
                trades_matrix[index, 6] = trades_matrix[index-1, 6]

            'update profit and loss'
            trades_matrix[index, 7] = trades_matrix[index, 5] * (trades_matrix[index, 0] - trades_matrix[index-1, 0])
                                    + trades_matrix[index, 6] * (trades_matrix[index, 1] - trades_matrix[index-1, 1])
            trades_matrix[index, 7] = round(trades_matrix[index, 7], 2)
            self.total_profit_loss += trades_matrix[index, 7]

        for key, index in zip(self.trades.keys(), range(0, trades_matrix.shape[0])):
            self.trades[key] = trades_matrix[index]

        return pd.DataFrame(trades_matrix[:, range(5, trades_matrix.shape[1])], columns=['Qty1', 'Qty2', 'P/L'])
```

Figure 4.3: Class: StockPair

Chapter 5

Client Server

The sender and receiver threads will support TCP/IP protocol and through Internet sockets. Server will wait for connection from client, will receive and process messages from client, and will process the request and send back response if client ask for. Socket diagram is shown in Figure 5.1.

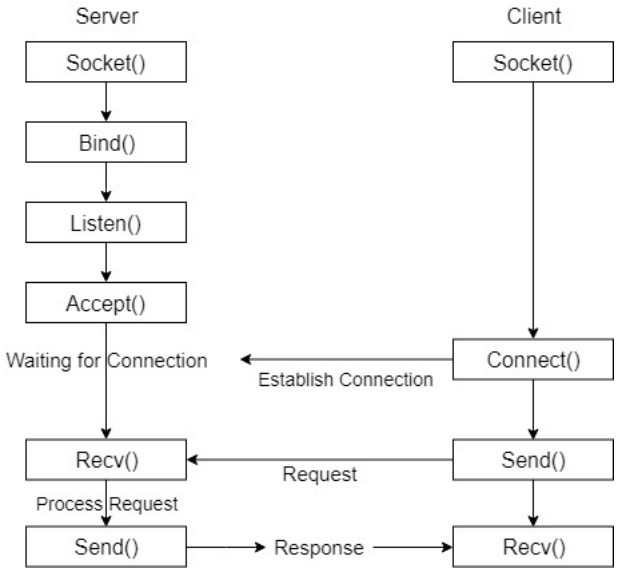

Figure 5.1: Socket Diagram

The sender and receiver threads achieve data synchronization using event and queue. When the client sends a request message to the server, server returns a message queue while the client is waiting for it. Whenever this event is set, data

in queue will be available to the client. This synchronization process is shown in Figure 5.2.

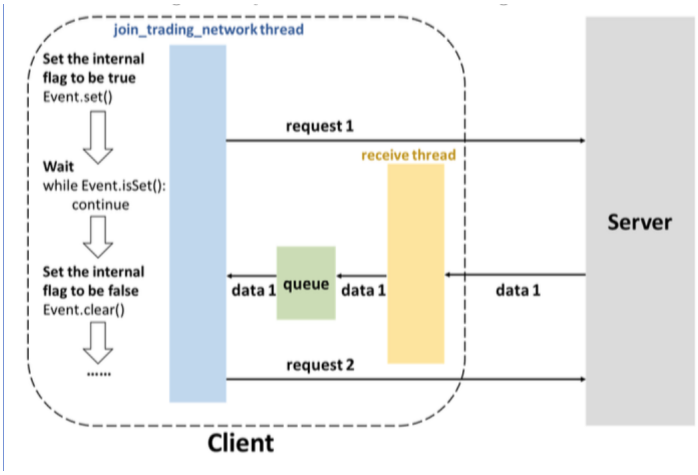

Figure 5.2: Synchronization Process Diagram

5.1 Order Book

Server will provide consolidated books for 30 trading days:

1. Simulated from market data starting from 1/2/2019.
2. Order book is consist of: order index, symbol, side (buy or sell), price, quantity, and order status, shown in Figure 5.3.
3. Each simulated trading date has one new book for every 30 seconds from daily historical data, with buy orders and sell orders simulated from the high and low price from the day, with price step of 0.05 and daily volume randomly distributed cross all price points.
4. Each simulated trading day lasts 30 seconds, following by 5 seconds of pending closing phase and 5 seconds of market closed phase before market reopen. There are 5 phases of market:
   (a) Not Open, start of simulated trading
   (b) Pending Open,

(c) Open, 30s

(d) Pending Close, 5s

(e) Market Close, 5s

5. The book supports partial fill, based on comparison of order quantity and available quantity on the book. The order status could be New, Filled or Partial Filled.

6. If the market status is pending close, trade price will be worse by 0.01.

7. All orders placed are limit orders.

	OrderIndex	Symbol	Side	Price	Qty	Status
1	1	AEP	Buy	72.30	160104	New
2	2	AEP	Sell	73.66	54961	New
3	3	AEP	Buy	72.35	28675	New
4	4	AEP	Sell	73.61	195949	New
5	5	AEP	Buy	72.40	50182	New
6	6	AEP	Sell	73.56	167273	New
7	7	AEP	Buy	72.45	229404	New
8	8	AEP	Sell	73.51	164884	New
9	9	AEP	Buy	72.50	145767	New
10	10	AEP	Sell	73.46	181611	New

Figure 5.3: Order Book

Chapter 6

Simulated Trading

After training, building, and backtesting the trading model, we will start simulated trading with a "real-time" data feed for 30 days, starting from 01/02/2019. Every 40 seconds, there is a market data feed of another day read from "eodhistoricaldata.com".

During simulated trading, first, it will logon to that includes a list of stocks from our trading model. Then it will loop to get market status until market open. It will send orders to server only during market open and pending closing. It will get order book information given a list of stocks to access the best available prices. Function get_orders() tries to get the trading signal using latest data and trained model, then orders are placed and traded if there is any signal on that day. After placing orders, a new day will start every 40 seconds. Then we loop to get market status and repeat this process again.

Finally, when 30 days passed, we quit client and server connection and start to do trading analysis, which will be published to web dashboard. We design an algorithm to detect the ending of 30-day trading period: we will quit the connection whenever the market status remain in "Close" for more than 150 seconds.

The flowchart of trading under network setting is shown in Figure 6.1.

Figure 6.1: Trading Under Network Setting Flowchart

Chapter 7

Performance Analysis

7.1 PnL

According to Figure 6.1, after both backtesting and simulated trading, we have trading analyses that will be displayed on the web dashboard. Figure 7.2 shows the backtesting result. We can see that backtested from 2018 to 2019, we have a profit of over 1 million, which is about 25% profit, starting with 1 million capital for each pair and 4 millions in total. All 4 pairs trades are profitable and the hit ratio is 100%. The maximum drawdown is about only 3%. The resulting Sharpe ratio is 2.5.

Cummulative PnL plot is also shown in Figure 6.2. Performance of our pairs trading model is much better than SP500 performance. The backtesting result shows that our trading model is very profitable, which is a contrast to our simulated trading result in Figure 6.4. There are several reasons that result in huge losses. First and the most important, our pairs trading model fails since pairs have different relationships during the first month of 2019 than the training data. SP500 index movement in 2018 is quite mean-reverting, where its constituents might also have stable relationships in this year. However, SP500 index at the start of 2019 is more momentum, where its constituents are likely to result in different relationships than in 2018. Second, we have flaws in our design of order book and order matching algorithm. Third, our trading model has not closed many positions in one month. Further discussions of these issues and potential improvements are in conclusion.

Trading Analysis

Profit	Total_Trades	Profit_Trades	Loss_Trades	Maximum_Drawdown	Sharpe_ratio
US$1062590.67	4	4	0	-117954.64	2.5

Figure 7.1: Backtesting Statistics

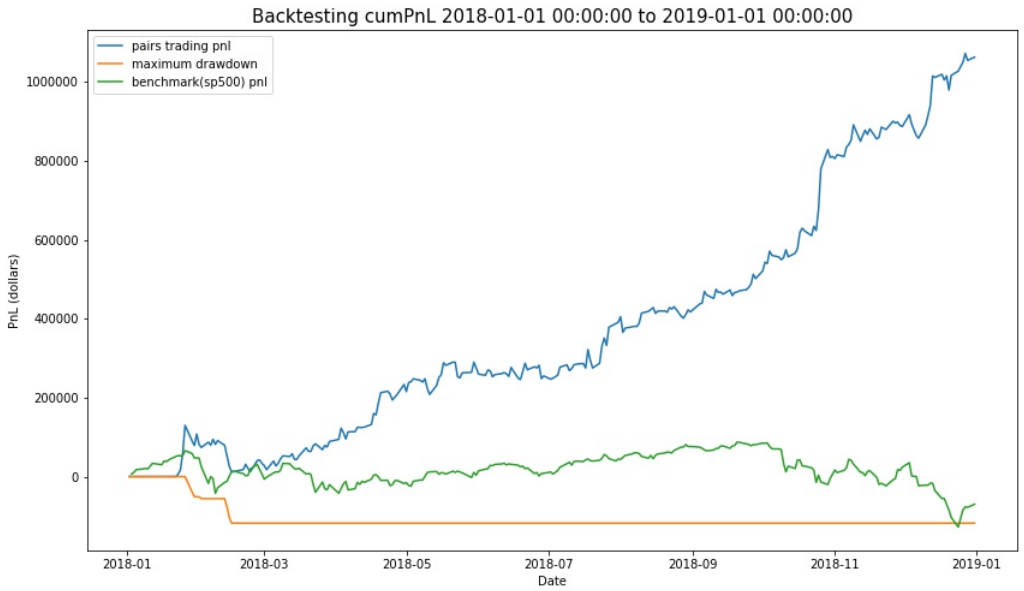

Figure 7.2: Backtesting PnL

7.2 Web Dashboard

Our web dashboard is built through flask. There are several modules on the web dashboard:

- Home Page: displays our pairs trading strategy logic, shown in Figure 7.4;
- Stock Pairs: displays Table stockspairs;
- Building Model: displays Table pairprices;
- Back Testing: displays Table trades;
- Trading Analysis: displays performance analysis table and plotted PnL for backtesting result;

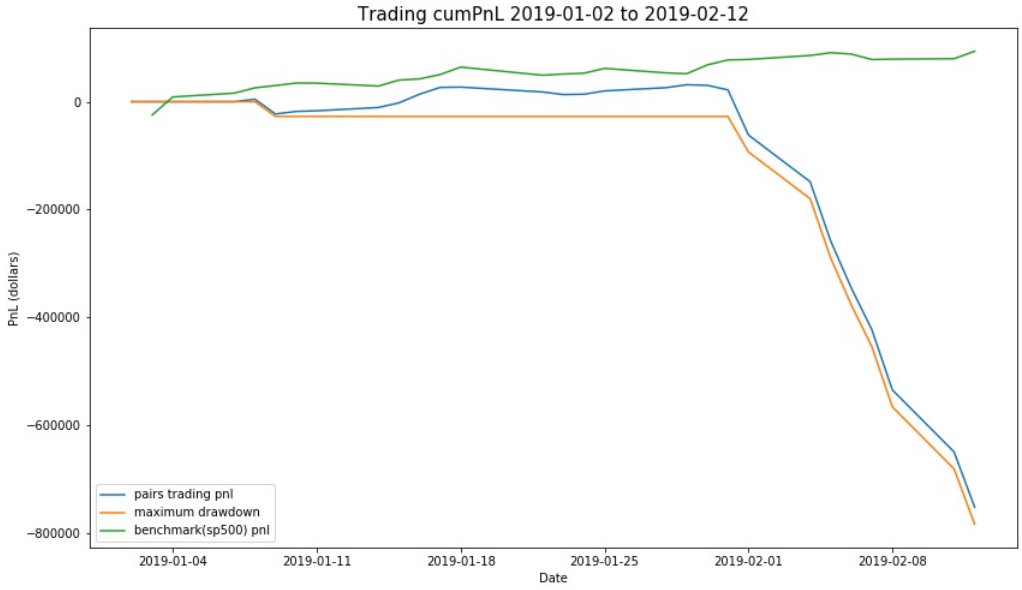

Figure 7.3: Simulated Trading PnL

- Real Trading: displays performance analysis table and plotted PnL for simulated tradingg result;

Whenever a module button is clicked on the page, the program will run on the server/client side and the display will show when the process is finished. Each module has to run in order. For "Real Trading" module, it will wait for over 10 minutes until the performance result shows on the page.

Flask will run on the main thread with the client side. We have another thread called client thread that will do simulated trading part with client/server communication. To do that, we need a global variable "bClientThreadStarted" to keep track if client thread start. It is set to false initially, and to true whenever simulated trading starts. The main thread will wait for client thread finishes to continue.

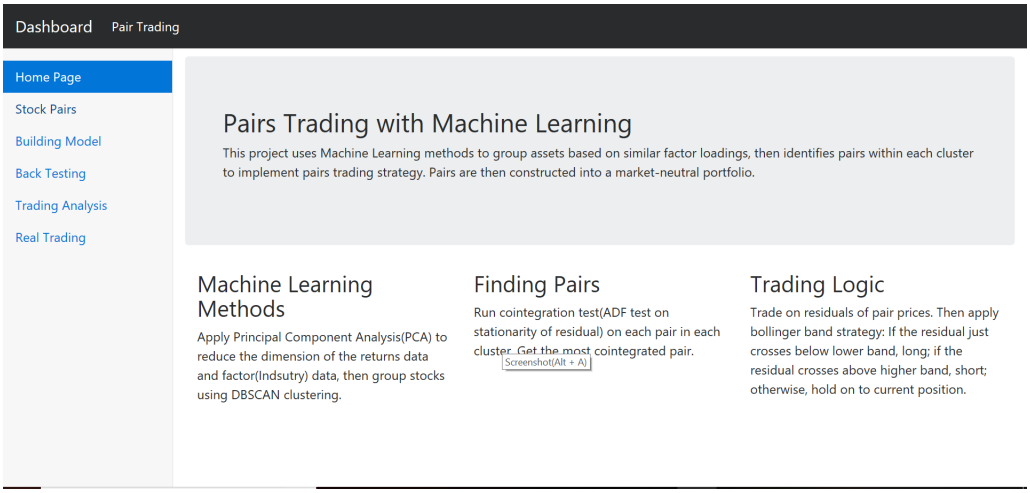

Figure 7.4: Dashboard Home Page

Chapter 8

Conclusions and Future Work

This capstone project builds a Python platform that can be used to test quantitative trading models in a network setting under client/server infrastructure with performance analysis displayed on web dashboard. From our backtesting and simulated trading results, we see that the performance of simulated trading is much worse than that of backtesting, although backtesting shows that our trading model has a 2.5 Sharpe ratio and 25% annual return. It concludes that we should be careful in real-time trading even with a profitable model shown in backtesting and we always want to do paper trading before real trading.

This python platform is a good tool for testing trading models, alerting for potential losses. However, there are limitations. Our assumptions of order book are simple and our simulated trading is not equivalent to paper trading. Paper trading is to trade based on real-time order books with no real money evolved. Our simulated trading has order book that is simulated from historical data and is fixed for each day. Our order book is too simple as compared to a real order book and is lack of real market dynamics. We would need tick level data to actually simulate a real market, for our first improvement that can be done to this project. Furthermore, we can improve our order matching algorithm and allow for market orders. We always place order at the best bid and offer of one order book of that day and it is always filled at this price. While in the real market, the limit order is fulfilled at our price and better, possibly split into different trades. It also has market order that takes many prices to get as much quantity filled as possible. For our pairs trading model, market orders are more reasonable because we are doing

dollar neutral strategy and our purpose is to get all quantity of orders filled.

There are also ways to improve our trading model. First, we have many hyperparameters in our trading model, including training and testing time periods, significance level in adjusted dickey fuller test, number of standard deviations and moving average period in bollinger band construction, number of principle components and epsilon in PCA. These hyperparameters can be optimized through cross validation. Second, we can do kalman filter instead of one simple ordinary least square. Third, we should set up stop loss level to prevent losses as in our simulated trading. Fourth, we should also include transaction costs in our model.

Appendix A

Appendix: Code

A.1 Platform Client

```
# -*- coding: utf-8 -*
#!/usr/bin/env python3

import socket
from socket import AF_INET, SOCK_STREAM
import threading
import queue

import json
import sys
import urllib.request
import pandas as pd
import matplotlib.pyplot as plt
import datetime as dt
import talib
import numpy as np
import time

from sklearn.cluster import DBSCAN
from sklearn import preprocessing
from sklearn.decomposition import PCA
import statsmodels.api as sm
import statsmodels.tsa.stattools as ts

from sqlalchemy import Column, ForeignKey, Integer, Float, String
from sqlalchemy import create_engine
from sqlalchemy import MetaData
from sqlalchemy import Table
from sqlalchemy import inspect
from sqlalchemy import and_


from flask import Flask, render_template
app = Flask(__name__, template_folder='templates')



clientID = "yicheng"


' download data '
```

```
data_start_date = dt.datetime(2014,1,1)  # hours:minute:seconds
data_end_date = dt.date.today()  # only dates
requestURL = "https://eodhistoricaldata.com/api/eod/"
myEodKey = "5ba84ea974ab42.45160048"
requestSP500 = "https://pkgstore.datahub.io/core/s-and-p-500-companies/constituents_json/data/64dd3e9582b936b0352fdd826ecd3c95/constituents_json.json"

' trading '
engine = create_engine('sqlite:///pairs_trading.db')
engine.execute("PRAGMA foreign_keys = ON")
metadata = MetaData()
metadata.reflect(bind=engine)  # bind to Engine, load all tables

' Parameters '
training_start_date = dt.datetime(2014,1,1)
training_end_date = dt.datetime(2018,1,1)
backtesting_start_date = dt.datetime(2018,1,1)
backtesting_end_date = dt.datetime(2019,1,1)
capital = 1000000.
significance = 0.05
k = 2
mvt = 10
# PCA
N_PRIN_COMPONENTS = 50
epsilon = 1.8



def get_daily_data(symbol='', start=data_start_date, end=data_end_date, requestType=requestURL,
                   apiKey=myEodKey, completeURL=None):
    if not completeURL:
        symbolURL = str(symbol) + '?'
        startURL = "from=" + str(start)
        endURL = "to=" + str(end)
        apiKeyURL = "api_token=" + myEodKey
        completeURL = requestURL + symbolURL + startURL + '&' + endURL + '&' + apiKeyURL + '&period=d&fmt=json'

    # if cannot open url
    try:
        with urllib.request.urlopen(completeURL) as req:
            data = json.load(req)
            return data
    except:
        pass


' populate stock data for each stock '
def download_stock_data(ticker, metadata, engine, table_name):
    column_names = ['symbol','date','open','high','low','close','adjusted_close','volume']
    price_list = []
    clear_a_table(table_name, metadata, engine)

    if 'GSPC' not in ticker:
        symbol_full = str(ticker) + ".US"
        stock = get_daily_data(symbol=symbol_full)
    else:
        stock = get_daily_data(symbol=ticker)

    if stock:
        for stock_data in stock:
            price_list.append([str(ticker), stock_data['date'], stock_data['open'], stock_data['high'],
                               stock_data['low'], stock_data['close'], stock_data['adjusted_close'],
                               stock_data['volume']])

    stocks = pd.DataFrame(price_list, columns=column_names)
```

```
106     stocks.to_sql(table_name, con=engine, if_exists='replace', index=False, chunksize=5)
107
108
109 def execute_sql_statement(sql_st, engine):
110     result = engine.execute(sql_st)
111     result_df = pd.DataFrame(result.fetchall())
112     result_df.columns = result.keys()
113     return result_df
114
115
116 ''' create table '''
117 def create_sp500_info_table(name, metadata, engine, null=False):
118     table = Table(name, metadata,
119                   Column('name', String(50), nullable=null),
120                   Column('sector', String(50), nullable=null),
121                   Column('symbol', String(50), primary_key=True, nullable=null),
122                   extend_existing = True)  # constructor
123     table.create(engine, checkfirst=True)
124
125 def create_price_table(name, metadata, engine, null=True):
126     if name != 'GSPC.INDX':
127         foreign_key = 'sp500.symbol'
128         table = Table(name, metadata,
129                       Column('symbol', String(50), ForeignKey(foreign_key),
130                              primary_key=True, nullable=null),
131                       Column('date', String(50), primary_key=True, nullable=null),
132                       Column('open', Float, nullable=null),
133                       Column('high', Float, nullable=null),
134                       Column('low', Float, nullable=null),
135                       Column('close', Float, nullable=null),
136                       Column('adjusted_close', Float, nullable=null),
137                       Column('volume', Integer, nullable=null),
138                       extend_existing = True)
139     else:
140         table = Table(name, metadata,
141             Column('symbol', String(50), primary_key=True, nullable=null),
142             Column('date', String(50), primary_key=True, nullable=null),
143             Column('open', Float, nullable=null),
144             Column('high', Float, nullable=null),
145             Column('low', Float, nullable=null),
146             Column('close', Float, nullable=null),
147             Column('adjusted_close', Float, nullable=null),
148             Column('volume', Integer, nullable=null),
149             extend_existing = True)
150     table.create(engine, checkfirst=True)
151
152 def create_stockpairs_table(table_name, metadata, engine):
153     table = Table(table_name, metadata,
154                   Column('Ticker1', String(50), primary_key=True, nullable=False),
155                   Column('Ticker2', String(50), primary_key=True, nullable=False),
156                   Column('Score', Float, nullable=False),
157                   Column('Profit_Loss', Float, nullable=False),
158                   extend_existing=True)
159     table.create(engine, checkfirst=True)
160
161 def create_pairprices_table(table_name, metadata, engine, null=True):
162     table = Table(table_name, metadata,
163                   Column('Symbol1', String(50), ForeignKey('stockpairs.Ticker1'), primary_key=
      True, nullable=null),
164                   Column('Symbol2', String(50), ForeignKey('stockpairs.Ticker2'), primary_key=
      True, nullable=null),
165                   Column('Date', String(50), primary_key=True, nullable=null),
166                   Column('Close1', Float, nullable=null),
167                   Column('Close2', Float, nullable=null),
168                   Column('Residual', Float, nullable=null),
169                   Column('Lower', Float, nullable=null),
170                   Column('MA', Float, nullable=null),
171                   Column('Upper', Float, nullable=null),
```

```
                    extend_existing=True)
    table.create(engine, checkfirst=True)

def create_trades_table(table_name, metadata, engine, null=False):
    table = Table(table_name, metadata,
                  Column('Symbol1', String(50), ForeignKey('stockpairs.Ticker1'), primary_key=
    True, nullable=null),
                  Column('Symbol2', String(50), ForeignKey('stockpairs.Ticker2'), primary_key=
    True, nullable=null),
                  Column('Date', String(50), primary_key=True, nullable=null),
                  Column('Close1', Float, nullable=null),
                  Column('Close2', Float, nullable=null),
                  Column('Qty1', Float, nullable=null),
                  Column('Qty2', Float, nullable=null),
                  Column('P/L', Float, nullable=null),
                  extend_existing=True)
    table.create(engine, checkfirst=True)

def clear_a_table(table_name, metadata, engine):
    conn = engine.connect()
    table = metadata.tables[table_name]
    delete_st = table.delete()
    conn.execute(delete_st)


def download_market_data(metadata, engine, sp500_info_df):
    print(" >>>>>>>>>>>>>>>>>>>>>>>>>>>><<<<<<<<<<<<<<<<<<<<<<<<<<<<")
    print("Downloading data ...")

    ' put sp500 constituent data into databases '
    create_sp500_info_table('sp500', metadata, engine)
    clear_a_table('sp500', metadata, engine)  # clear table before insert
    sp500_info_df.to_sql('sp500', con=engine, if_exists='append', index=False,
                         chunksize=5)

    ' get data for each ticker from sp500 '
    for symbol in sp500_info_df.Symbol:
        create_price_table(symbol, metadata, engine)
        download_stock_data(symbol, metadata, engine, symbol)

    ' SP500 index price '
    create_price_table('GSPC.INDX', metadata, engine)
    download_stock_data('GSPC.INDX', metadata, engine, 'GSPC.INDX')

    print("Finished downloading.")


def training_data(metadata, engine, significance, sp500_info_df,
                  training_start_date, training_end_date):
    print(" >>>>>>>>>>>>>>>>>>>>>>>>>>>><<<<<<<<<<<<<<<<<<<<<<<<<<<<")
    print("Training data ...")
    print("Start date:", training_start_date, ", End date:", training_end_date)

    ' get training set '
    Price = pd.DataFrame()

    for symbol in sp500_info_df.Symbol:
        select_st = "SELECT date, adjusted_close From " + "\"" + symbol + "\"" +  \
        " WHERE date >= " + "\"" + str(training_start_date) + "\"" +     \
        " AND date <= " + "\"" + str(training_end_date) + "\"" + ";"
        try:
            result_df = execute_sql_statement(select_st, engine)
            result_df.set_index('date', inplace=True)  # date as index
            result_df.columns = [symbol]  # name is column
            Price = pd.concat([Price, result_df], axis=1, sort=True)
        except:
            pass

```

```
    ' PCA: reduce dimension '
    Price.sort_index(inplace=True)
    Price.fillna(method='ffill', inplace=True)
    Price = Price.loc[:,(Price>0).all(0)]  # every price > 0

    Price_ret = Price.pct_change()
    Price_ret = Price_ret.replace([np.inf, -np.inf], np.nan)
    Price_ret.dropna(axis=0, how='all', inplace=True) # drop first row (NA)
    Price_ret.dropna(axis=1, how='any', inplace=True)

    pca = PCA(n_components=N_PRIN_COMPONENTS)
    pca.fit(Price_ret)
    X = pd.DataFrame(pca.components_.T, index=Price_ret.columns)
    sp500_info_df.set_index('Symbol', inplace=True)
    X = pd.concat([X, sp500_info_df.Sector.T], axis=1, sort=True)
    X = pd.get_dummies(X)

    ' DBSCAN: identify clusters from stocks that are closest '
    X.dropna(axis=0, how='any', inplace=True)
    X_arr = preprocessing.StandardScaler().fit_transform(X)
    clf = DBSCAN(eps=epsilon, min_samples=3)

    # labels is label values from -1 to x
    # -1 represents noisy samples that are not in clusters
    clf.fit(X_arr)
    clustered = clf.labels_
    # all stock with its cluster label (including -1)
    clustered_series = pd.Series(index=X.index, data=clustered.flatten())
    # clustered stock with its cluster label
    clustered_series = clustered_series[clustered_series != -1]

    poss_cluster = clustered_series.value_counts().sort_index()
    print(poss_cluster)

    'identify cointegrated pairs from clusters'
    def Cointegration(cluster, significance, start_day, end_day):
        pair_coin = []
        p_value = []
        adf = []
        n = cluster.shape[0]
        keys = cluster.keys()
        for i in range(n):
            for j in range(i+1,n):
                asset_1 = Price.loc[start_day:end_day, keys[i]]
                asset_2 = Price.loc[start_day:end_day, keys[j]]
                results = sm.OLS(asset_1, asset_2)
                results = results.fit()
                predict = results.predict(asset_2)
                error = asset_1 - predict
                ADFtest = ts.adfuller(error)
                if ADFtest[1] < significance:
                    pair_coin.append([keys[i], keys[j]])  # pair names
                    p_value.append(ADFtest[1])  # p value, smaller the better
                    adf.append(ADFtest[0])  # adf test stats, larger the better
        return p_value, pair_coin, adf

    "Pair selection method"
    "select a pair with lowest p-value from each cluster"
    def PairSelection(clustered_series, significance,
                      start_day=str(training_start_date), end_day=str(training_end_date)):
        Opt_pairs = []   # to get best pair in cluster i
        tstats = []

        for i in range(len(poss_cluster)):
            cluster = clustered_series[clustered_series == i]
            result = Cointegration(cluster, significance, start_day, end_day)
            if len(result[0]) > 0:
                if np.min(result[0]) < significance:
```

```
                    index = np.where(result[0] == np.min(result[0]))[0][0]
                    Opt_pairs.append([result[1][index][0], result[1][index][1]])
                    tstats.append(round(result[2][index], 4))

        return Opt_pairs, tstats

    stock_pairs, tstats = PairSelection(clustered_series, significance)
    # put into sql table
    create_stockpairs_table('stockpairs', metadata, engine)
    clear_a_table('stockpairs', metadata, engine)
    stock_pairs = pd.DataFrame(stock_pairs, columns=['Ticker1', 'Ticker2'])
    stock_pairs["Score"] = -1 * np.array(tstats)
    stock_pairs["Profit_Loss"] = 0.0
    stock_pairs.to_sql('stockpairs', con=engine, if_exists='append', index=False, chunksize=5)

    print(stock_pairs[["Ticker1", "Ticker2"]])
    print("Finished training.")
    return stock_pairs


def building_model(metadata, engine, k, mvt,
                   backtesting_start_date, backtesting_end_date):
    global ols_results
    '''
    get pair prices, moving averages, bollinger bands
    k: number of std
    mvt: moving average period
    '''
    print(" >>>>>>>>>>>>>>>>>>>>>>><<<<<<<<<<<<<<<<<<<<<<<<<<")
    print("Building Model ...")
    print("Parameters: k =", k, ", moving average =", mvt)

    select_st = "SELECT Ticker1, Ticker2 from stockpairs;"
    stock_pairs = execute_sql_statement(select_st, engine)

    create_pairprices_table('pairprices', metadata, engine, mvt)
    clear_a_table('pairprices', metadata, engine)

    for pair in stock_pairs.values:
        select_st = "SELECT stockpairs.Ticker1 as Symbol1, stockpairs.Ticker2 as Symbol2, \
                     " + pair[0] + ".date as Date, " + pair[0] + ".Adjusted_close as Close1, \
                     " + pair[1] + ".Adjusted_close as Close2 \
                     From " + pair[0] + ", " + pair[1] + ", stockpairs \
                     Where (((stockpairs.Ticker1 = " + pair[0] + ".symbol) and \
                     (stockpairs.Ticker2 = " + pair[1] + ".symbol)) and \
                    (" + pair[0] + ".date = " + pair[1] + ".date)) \
                    and " + pair[0] + ".date >= " + "\"" + str(training_start_date) + "\"" + \
                    " AND " + pair[0] + ".date <= " + "\"" + str(training_end_date) + "\"  \
                    ORDER BY Symbol1, Symbol2;"

        result_df = execute_sql_statement(select_st, engine)

        select_st = "SELECT stockpairs.Ticker1 as Symbol1, stockpairs.Ticker2 as Symbol2, \
                     " + pair[0] + ".date as Date, " + pair[0] + ".Adjusted_close as Close1, \
                     " + pair[1] + ".Adjusted_close as Close2 \
                     FROM " + pair[0] + ", " + pair[1] + ", stockpairs \
                     WHERE (((stockpairs.Ticker1 = " + pair[0] + ".symbol) and \
                     (stockpairs.Ticker2 = " + pair[1] + ".symbol)) and \
                     (" + pair[0] + ".date = " + pair[1] + ".date)) \
                     and " + pair[0] + ".date >= " + "\"" + str(backtesting_start_date) + "\"" + \
                     " AND " + pair[0] + ".date <= " + "\"" + str(backtesting_end_date) + "\" \
                     ORDER BY Symbol1, Symbol2;"
        result_df2 = execute_sql_statement(select_st, engine)

        # get bollinger band
```

```
371         results = sm.OLS(result_df.Close1, sm.add_constant(result_df.Close2)).fit()
372         predict = results.params[0] + results.params[1] * result_df2.Close2
373         ols_results[pair[0]] = results
374         error = np.subtract(result_df2.Close1, predict)
375         upperband, middleband, lowerband = talib.BBANDS(error, timeperiod=mvt,
376                                                          nbdevup=k, nbdevdn=k, matype=0)
377         result_df2[['Residual', 'Lower', 'MA', 'Upper']] = pd.DataFrame([error, lowerband,
    middleband, upperband]).T.round(4)
378         result_df2.to_sql('pairprices', con=engine, if_exists='append', index=False, chunksize
    =5)
379
380     print("Finished building model.")
381
382
383 class StockPair:
384
385     def __init__(self, symbol1, symbol2, start_date, end_date):
386         self.ticker1 = symbol1
387         self.ticker2 = symbol2
388         self.start_date = start_date
389         self.end_date = end_date
390         self.trades = {}
391         self.total_profit_loss = 0.0
392
393     def __str__(self):
394         return str(self.__class__) + ": " + str(self.__dict__) + "\n"
395
396     def __repr__(self):
397         return str(self.__class__) + ": " + str(self.__dict__) + "\n"
398
399     def createTrade(self, date, close1, close2, res, lower, upper, qty1 = 0, qty2 = 0,
    profit_loss = 0.0):
400         self.trades[date] = np.array([close1, close2, res, lower, upper, qty1, qty2,
    profit_loss])
401
402     def updateTrades(self):  # dollar neutral, available dollar for buy/sell for each pair
403         trades_matrix = np.array(list(self.trades.values()))
404
405         for index in range(1, trades_matrix.shape[0]):
406             # RES SELL SIGNAL: buy asset 1, sell asset 2
407             if (trades_matrix[index-1, 2] < trades_matrix[index-1, 4] and
408                 trades_matrix[index, 2] > trades_matrix[index, 4]):
409                 trades_matrix[index, 5] = int(capital / trades_matrix[index, 0])
410                 trades_matrix[index, 6] = int(-capital / trades_matrix[index, 1])
411             # RES BUY SIGNAL: sell asset 1, buy asset 2
412             elif (trades_matrix[index-1, 2] > trades_matrix[index-1, 3] and
413                 trades_matrix[index, 2] < trades_matrix[index, 3]):
414                 trades_matrix[index, 5] = int(-capital / trades_matrix[index, 0])
415                 trades_matrix[index, 6] = int(capital / trades_matrix[index, 1])
416             # no act
417             else:
418                 trades_matrix[index, 5] = trades_matrix[index-1, 5]
419                 trades_matrix[index, 6] = trades_matrix[index-1, 6]
420
421             'update profit and loss'
422             trades_matrix[index, 7] = trades_matrix[index, 5] * (trades_matrix[index, 0] -
    trades_matrix[index-1, 0])      \
423                                       + trades_matrix[index, 6] * (trades_matrix[index, 1] -
    trades_matrix[index-1, 1])
424             trades_matrix[index, 7] = round(trades_matrix[index, 7], 2)
425             self.total_profit_loss += trades_matrix[index, 7]
426
427         for key, index in zip(self.trades.keys(), range(0, trades_matrix.shape[0])):
428             self.trades[key] = trades_matrix[index]
429
430         return pd.DataFrame(trades_matrix[:, range(5, trades_matrix.shape[1])], columns=['Qty1'
    , 'Qty2', 'P/L'])
431
```

```

def back_testing(metadata, engine, backtesting_start_date, backtesting_end_date):
    print(" >>>>>>>>>>>>>>>>>>>>>>>><<<<<<<<<<<<<<<<<<<<<<<<<<")
    print("Backtesting ...")
    print("Start date:", backtesting_start_date, ", End date:", backtesting_end_date)

    print('create StockPair')
    stock_pair_map = dict()

    select_st = 'SELECT Ticker1, Ticker2 FROM stockpairs;'
    stock_pairs = execute_sql_statement(select_st, engine)

    for index, row in stock_pairs.iterrows():
        aKey = (row['Ticker1'], row['Ticker2'])
        stock_pair_map[aKey] = StockPair(row['Ticker1'], row['Ticker2'],
                                         backtesting_start_date, backtesting_end_date)

    print('create Trades')
    select_st = 'SELECT * FROM pairprices;'
    result_df = execute_sql_statement(select_st, engine)

    for index in range(result_df.shape[0]):
        aKey = (result_df.at[index, 'Symbol1'], result_df.at[index, 'Symbol2'])
        stock_pair_map[aKey].createTrade(result_df.at[index, 'Date'],
                     result_df.at[index, 'Close1'], result_df.at[index, 'Close2'],
                     result_df.at[index, 'Residual'], result_df.at[index, 'Lower'],
                     result_df.at[index, 'Upper'])

    print('update Trades')
    trades_df = pd.DataFrame(columns=['Qty1', 'Qty2', 'P/L'])
    for key, value in stock_pair_map.items():
        trades_df = trades_df.append(value.updateTrades(), ignore_index=True)

        table = metadata.tables['stockpairs']
        update_st = table.update().values(Profit_Loss=value.total_profit_loss).where( \
                and_(table.c.Ticker1==value.ticker1, table.c.Ticker2==value.ticker2))
        engine.execute(update_st)

    result_df = result_df[['Symbol1', 'Symbol2', 'Date', 'Close1', 'Close2']].join(trades_df)

    create_trades_table('trades', metadata, engine)
    clear_a_table('trades', metadata, engine)
    result_df.to_sql('trades', con=engine, if_exists='append', index=False, chunksize=5)

    print("Finished backtesting.")


'real time data according to market date'
def feed_realtime_data(ticker, start, end):
    global price_data
    column_names = ['symbol','date','adjusted_close']

    stock = get_daily_data(symbol=ticker, start=start, end=end)
    if stock:
        for stock_data in stock:
            price_data.append([str(ticker), stock_data['date'],
                               stock_data['adjusted_close']])
    stocks = pd.DataFrame(price_data, columns=column_names)
    stocks.adjusted_close = stocks.adjusted_close.astype(float)
    return stocks


def get_orders(market_date=None):
    orders_list = []

    select_st = 'SELECT Ticker1, Ticker2 FROM stockpairs;'
    pairs = execute_sql_statement(select_st, engine)

```

```
500     for index, row in pairs.iterrows():
501         # previous data for ols fit
502         select_st = "SELECT symbol, date, adjusted_close FROM "+str(row[0])+    \
503                     " WHERE date >= " + "\"" + str(backtesting_start_date) + "\"" +     \
504                     " AND date <= " + "\"" + str(backtesting_end_date) + "\"" + ";"
505         result1 = execute_sql_statement(select_st, engine)
506         select_st = "SELECT symbol, date, adjusted_close FROM "+str(row[1])+    \
507                     " WHERE date >= " + "\"" + str(backtesting_start_date) + "\"" +     \
508                     " AND date <= " + "\"" + str(backtesting_end_date) + "\"" + ";"
509         result2 = execute_sql_statement(select_st, engine)
510 
511         if market_date:
512             # append latest real data to previous data
513             stock1 = feed_realtime_data(row[0], market_date, market_date)
514             stock1 = stock1[stock1.symbol == row[0]]
515             result1 = pd.concat([result1, stock1], ignore_index=True)
516             stock2 = feed_realtime_data(row[1], market_date, market_date)
517             stock2 = stock2[stock2.symbol == row[1]]
518             result2 = pd.concat([result2, stock2], ignore_index=True)
519 
520         try:
521             results = ols_results[row[0]]
522             predict = results.params[0] + results.params[1] * result2.adjusted_close
523             error = np.subtract(result1.adjusted_close, predict)
524             upperband, middleband, lowerband = talib.BBANDS(error, timeperiod=mvt,
525                                                              nbdevup=k, nbdevdn=k, matype=0)
526             price1 = round(result1.adjusted_close.values[-1], 2)
527             price2 = round(result2.adjusted_close.values[-1], 2)
528 
529             if (error.values[-2] < upperband.values[-2] and error.values[-1] > upperband.values
    [-1]):
530                 amt1 = int(capital / price1)
531                 amt2 = int(capital / price2)
532                 order1 = 'Order New '+row[0]+' Buy '+str(price1)+' '+str(amt1)
533                 order2 = 'Order New '+row[1]+' Sell '+str(price2)+' '+str(amt2)
534                 orders_list.append(order1)
535                 orders_list.append(order2)
536                 print(order1, ',', order2)
537 
538             elif error.values[-2] > lowerband.values[-2] and error.values[-2] < lowerband.
    values[-1]:
539                 amt1 = int(capital / price1)
540                 amt2 = int(capital / price2)
541                 order1 = 'Order New '+row[0]+' Sell '+str(price1)+' '+str(amt1)
542                 order2 = 'Order New '+row[1]+' Buy '+str(price2)+' '+str(amt2)
543                 orders_list.append(order1)
544                 orders_list.append(order2)
545                 print(order1, ',', order2)
546 
547             else:
548                 print(row[0], row[1], 'No order signal.')
549 
550         except:
551             print('No order signal.')
552 
553     return orders_list
554 
555 
556 def receive(e, q):
557     """Handles receiving of messages."""
558     total_server_response = []
559     msg_end_tag = ".$$$$"
560 
561     while True:
562         try:
563             recv_end = False
564             # everytime only load certain size
565             server_response = client_socket.recv(BUFSIZ).decode("utf8")
```

```
566
567             if server_response:
568                 if msg_end_tag in server_response:  # if reaching end of message
569                     server_response = server_response.replace(msg_end_tag, '')
570                     recv_end = True
571
572                 # append every response
573                 total_server_response.append(server_response)
574
575                 # if reaching the end, put it into queue
576                 if recv_end == True:
577                     server_response_message = ''.join(total_server_response)
578                     data = json.loads(server_response_message)
579                     #print(data)
580                     q.put(data)
581                     total_server_response = []
582
583                     if e.isSet():
584                         e.clear()
585
586         except OSError:  # Possibly client has left the chat.
587             break
588
589
590 ' The logon message includes the list of stocks from client '
591 def get_stock_list_from_database():
592     select_st = 'SELECT Ticker1, Ticker2 FROM stockpairs;'
593     pairs = execute_sql_statement(select_st, engine)
594     tickers = pd.concat([pairs["Ticker1"], pairs["Ticker2"]], ignore_index=True)
595     tickers.drop_duplicates(keep='first', inplace=True)
596     tickers.sort_values(axis=0, ascending=True, inplace=True, kind='quicksort')
597     print(tickers)
598     return tickers
599
600 def logon():
601     tickers = get_stock_list_from_database();
602     client_msg = json.dumps({'Client':clientID, 'Status':'Logon', 'Stocks':tickers.str.cat(sep=
        ',')})
603     return client_msg
604
605 def get_user_list():
606     client_msg = "{\"Client\":\"" + clientID + "\", \"Status\":\"User List\"}"
607     return client_msg
608
609 def get_stock_list():
610     client_msg = "{\"Client\":\"" + clientID + "\", \"Status\":\"Stock List\"}"
611     return client_msg
612
613 def get_market_status():
614     client_msg = json.dumps({'Client':clientID, 'Status':'Market Status'})
615     return client_msg
616
617 def get_order_table(stock_list):
618     client_msg = json.dumps({'Client':clientID, 'Status':'Order Inquiry', 'Symbol':stock_list})
619     return client_msg
620
621 def enter_a_new_order(symbol, side, price, qty):
622     client_msg = json.dumps({'Client':clientID, 'Status':'New Order', 'Symbol':symbol, 'Side':
        side, 'Price':price, 'Qty':qty})
623     return client_msg
624
625 def quit_connection():
626     client_msg = "{\"Client\":\"" + clientID + "\", \"Status\":\"Quit\"}"
627     return client_msg
628
629 def send_msg(client_msg):
630     client_socket.send(bytes(client_msg, "utf8"))
631     data = json.loads(client_msg)
```

```
632     return data
633
634 def set_event(e):
635     e.set();
636
637 def wait_for_an_event(e):
638     while e.isSet():
639         continue
640
641 def get_data(q):
642     data = q.get()
643     q.task_done()
644 #     print(dt.datetime.now(), data)
645     return data
646
647
648 # command in queue
649 def join_trading_network(e, q):
650     global market_period_list, record_order_df
651     last_close_time = time.time()
652
653     threading.Thread(target=receive, args=(e,q)).start()
654
655     set_event(e)
656     send_msg(logon())  # automatic logon
657     wait_for_an_event(e)
658     get_data(q)
659
660     set_event(e)
661     send_msg(get_user_list())  # automatic print out user list
662     wait_for_an_event(e)
663     get_data(q)
664
665     set_event(e)
666     send_msg(get_stock_list())  # automatically print out stock list
667     wait_for_an_event(e)
668     get_data(q)
669
670     while True:
671         set_event(e)
672         client_msg = get_market_status()  # automatically print market status
673         send_msg(client_msg)
674         wait_for_an_event(e)
675         data = get_data(q)
676         market_status = data["Market Status"]
677
678         'The client will loop until market open'
679         if (market_status == "Market Closed" or
680             market_status == "Pending Open" or
681             market_status == "Not Open"):
682             # if market closed too long, stop trading
683             if time.time() - last_close_time > 150:
684                 print('>>>> Stop trading after ', time.time() - last_close_time, 'seconds')
685                 break;
686             time.sleep(1)
687             continue
688
689         last_close_time = time.time()
690
691         ' place order every 40s (1day) '
692         print('==========================================================')
693         market_period = data["Market Period"]
694         market_period_list.append(market_period)  # store past dates
695         print("Current market status is:", market_status)
696         print("Market period is:", market_period_list)
697
698         ' pLace order according to strategy using previous close price'
699         if len(market_period_list) > 1:
```

```
            prev_date = market_period_list[-2]
            orders_list = get_orders(prev_date)  # up to previous day close price
        else:
            orders_list = get_orders()

        'The client will send orders to server only during market open and pending closing'
        if orders_list:

            for order in orders_list:
                order_list = order.split(" ")
                mySymbol = order_list[2]
                mySide = order_list[3]
                myPrice = float(order_list[4])
                myQuantity = int(order_list[5])

                set_event(e)
                send_msg(get_order_table([mySymbol]))  # pass in list
                wait_for_an_event(e)
                data = get_data(q)
                order_data = json.loads(data)
                order_table = pd.DataFrame(order_data["data"])
                if order_table.empty:
                    print('Empty table')
                    continue

                if mySide == 'Buy':
                    order_table = order_table[order_table["Side"] == 'Sell']
                    order_table.sort_values('Price', ascending=True, inplace=True)
                    order_table.reset_index(drop=True, inplace=True)
                    best_price = order_table.loc[0, 'Price']
                    order_index = order_table.loc[0, 'OrderIndex']
                else:
                    order_table = order_table[order_table["Side"] == 'Buy']
                    order_table.sort_values('Price', ascending=False, inplace=True)
                    order_table.reset_index(drop=True, inplace=True)
                    best_price = order_table.loc[0, 'Price']
                    order_index = order_table.loc[0, 'OrderIndex']
                print(order_table.iloc[0, :])
                print('today best price', best_price, ', previous day close price', myPrice, ',
     order index', order_index)

                set_event(e)
                client_msg = enter_a_new_order(symbol=mySymbol, side=mySide, price=best_price,
    qty=myQuantity)
                send_msg(client_msg)
                wait_for_an_event(e)
                data = get_data(q)

                'record orders'
                record_order = pd.Series([market_period, mySymbol, mySide, best_price,
    myQuantity])
                record_order_df = pd.concat([record_order_df, record_order], axis=1)

        time.sleep(30)  # skip to next day

    record_order_df = record_order_df.T
    try:
        record_order_df.columns = ['Date', 'Symbol', 'Side', 'Price', 'Quantity']
        record_order_df.loc[record_order_df['Side']=='Sell', 'Quantity'] = -1.*record_order_df.
    loc[record_order_df['Side']=='Sell', 'Quantity']
        record_order_df.set_index(['Symbol', 'Date'], inplace=True)
        print(record_order_df)
    except:
        print('No Orders!!!!')


    set_event(e)
    send_msg(quit_connection())  # automatically quit
```

```
764     wait_for_an_event(e)
765
766
767
768 'define function to calculate maximum drawdown'
769 def MaxDrawdown(Ret_Cum):
770     # ret_cum also can be portfolio position series
771     ContVal = np.zeros(np.size(Ret_Cum))
772     MaxDD = np.zeros(np.size(Ret_Cum))
773     for i in range(np.size(Ret_Cum)):
774         if i == 0:
775             if Ret_Cum[i] < 0:
776                 ContVal[i] = Ret_Cum[i]
777             else:
778                 ContVal[i] = 0
779         else:
780             ContVal[i] = Ret_Cum[i] - np.nanmax(Ret_Cum[0:(i+1)])
781         MaxDD[i] = np.nanmin(ContVal[0:(i+1)])
782     return MaxDD
783
784
785 @app.route('/')
786 def index():
787     return render_template("index.html")
788
789
790 @app.route('/data_prep')
791 def data_prep():
792     inspector = inspect(engine)
793
794     sp500_info = get_daily_data(completeURL=requestSP500)
795     sp500_info_df = pd.DataFrame(sp500_info)
796     if len(inspector.get_table_names()) == 0:  # if no market data, download market data
797         download_market_data(metadata, engine, sp500_info_df)
798     else:
799         print(" >>>>>>>>>>>>>>>>>>>>>>>><<<<<<<<<<<<<<<<<<<<<<<<")
800         print("Data already downloaded ...")
801
802     stock_pairs = training_data(metadata, engine, significance, sp500_info_df,
803                                 training_start_date, training_end_date)
804     pairs = stock_pairs.transpose()
805     list_of_pairs = [pairs[i] for i in pairs]
806     return render_template("data_prep.html", pair_list=list_of_pairs)
807
808
809 @app.route('/build_model')
810 def build_model():
811     building_model(metadata, engine, k, mvt,
812                    backtesting_start_date, backtesting_end_date)
813
814     select_st = "SELECT * from pairprices;"
815     result_df = execute_sql_statement(select_st, engine)
816     result_df = result_df.transpose()
817     list_of_pairs = [result_df[i] for i in result_df]
818     return render_template("build_model.html", pair_list=list_of_pairs)
819
820
821 @app.route('/back_test')
822 def model_back_testing():
823     back_testing(metadata, engine, backtesting_start_date, backtesting_end_date)
824
825     select_st = "SELECT * from stockpairs;"
826     result_df = execute_sql_statement(select_st, engine)
827     result_df['Score'] = result_df['Score'].map('{:.4f}'.format)
828     result_df['Profit_Loss'] = result_df['Profit_Loss'].map('${:,.2f}'.format)
829     result_df = result_df.transpose()
830     list_of_pairs = [result_df[i] for i in result_df]
831     return render_template("back_testing.html", pair_list=list_of_pairs)
```

```python


@app.route('/trade_analysis')
def trade_analysis():
    print(" >>>>>>>>>>>>>>>>>>>>>>>>><<<<<<<<<<<<<<<<<<<<<<<<<<<<")
    print("Generating trading analysis ...")

    select_st = "SELECT printf(\"US$%.2f\", sum(Profit_Loss)) AS Profit, count(Profit_Loss) AS
     Total_Trades, \
                 sum(CASE WHEN Profit_Loss > 0 THEN 1 ELSE 0 END) AS Profit_Trades, \
                 sum(CASE WHEN Profit_Loss < 0 THEN 1 ELSE 0 END) AS Loss_Trades FROM StockPairs
     ;"
    result_df = execute_sql_statement(select_st, engine)

    'sp500 pnl'
    select_st = "SELECT symbol, date, adjusted_close FROM [GSPC.INDX]"+    \
                 " WHERE date >= " + "\"" + str(backtesting_start_date) + "\"" +     \
                 " AND date <= " + "\"" + str(backtesting_end_date) + "\"" + ";"
    sp_df = execute_sql_statement(select_st, engine)
    sp_df['ret'] = sp_df['adjusted_close'].pct_change()
    sp_df['cumpnl'] = capital * (1 + sp_df['ret']).cumprod() - capital
    sp_df.index = pd.to_datetime(sp_df.date)

    'Get pnl'
    select_st = 'SELECT Ticker1, Ticker2 FROM stockpairs;'
    pair_df = execute_sql_statement(select_st, engine)
    select_st = 'SELECT * FROM trades;'
    pnl_df = execute_sql_statement(select_st, engine)
    total_pnl = pd.DataFrame(0, columns=["P/L"], index=pnl_df.Date.unique())

    for value in pair_df.values:
        pnl = pnl_df.loc[pnl_df.Symbol1==value[0], ["Date","P/L"]]
        pnl.set_index("Date", inplace=True)
        total_pnl = total_pnl.add(pnl)  # adding two dataframe

    cumpnl = total_pnl.cumsum()
    maxdraw = MaxDrawdown(cumpnl['P/L'].values)
    result_df["Max_Drawdown"] = maxdraw[-1]
    cumret = cumpnl.pct_change()
    cumret = cumret.replace(np.inf, np.nan)
    cumret = cumret.replace(-np.inf, np.nan)
    result_df["Sharpe"] = np.sqrt(252) * np.nanmean(cumret) / np.nanstd(cumret)
    result_df = result_df.round(2)

    print(result_df.to_string(index=False))
    result_df = result_df.transpose()
    trade_results = [result_df[i] for i in result_df]

    'plot'
    cumpnl.index = pd.to_datetime(cumpnl.index)
    maxdraw = pd.DataFrame(maxdraw, index=cumpnl.index)
    fig = plt.figure(figsize=(12,7))
    plt.title('Backtesting cumPnL '+str(backtesting_start_date)+' to '+str(backtesting_end_date
     ),
              fontsize=15)
    plt.xlabel('Date')
    plt.ylabel('PnL (dollars)')
    plt.plot(cumpnl, label='pairs trading pnl')
    plt.plot(maxdraw, label='maximum drawdown')
    plt.plot(sp_df['cumpnl'], label='benchmark(sp500) pnl')
    plt.legend()
    plt.tight_layout()
    fig.savefig('static/plots/backtest_pnl.jpg')
    plt.show()
    return render_template("trade_analysis.html", trade_list=trade_results)


@app.route('/real_trade')
```

```python
def real_trade():
    global bClientThreadStarted, client_thread

    print(" >>>>>>>>>>>>>>>>>>>>>>><<<<<<<<<<<<<<<<<<<<<<<<<<")
    print("Real trading ...", bClientThreadStarted)

    if bClientThreadStarted == False:
        client_thread.start()
        bClientThreadStarted = True
        print("Client thread starts ...", bClientThreadStarted)
        client_thread.join()  # wait until this thread finishes, then continue main thread

    'real trade analysis'
    print(" >>>>>>>>>>>>>>>>>>>>>>><<<<<<<<<<<<<<<<<<<<<<<<<<")
    print("Trading analysis ...")
    get_orders(market_period_list[-1])
    stocks_df = pd.DataFrame(price_data, columns=['symbol','date','adjusted_close'])
    stocks_df.adjusted_close = stocks_df.adjusted_close.astype(float)
    total_pnl = pd.Series(0, index=stocks_df.date.unique())

    try:
        for stock in record_order_df.index.levels[0]:
            order_df = record_order_df.loc[stock,:]
            stock_df = stocks_df[stocks_df['symbol']==stock]
            stock_df.set_index('date', inplace=True)
            join_df = stock_df.join(order_df)
            join_df.fillna(method='ffill', inplace=True)
            join_df['pnl'] = (join_df['adjusted_close'] - join_df['Price']) * join_df['Quantity']
            total_pnl = total_pnl.add(join_df.pnl, fill_value=0)  # series + series
    except:
        pass  # if no orders

    result_df = pd.DataFrame()
    result_df.loc[0,'Profits'] = sum(total_pnl)
    result_df.loc[0,'Total_Trades'] = len(record_order_df) / 2

    cumpnl = total_pnl.cumsum()
    maxdraw = MaxDrawdown(cumpnl.values)
    result_df.loc[0,"Max_Drawdown"] = maxdraw[-1]
    cumret = cumpnl.pct_change()
    cumret = cumret.replace(np.inf, np.nan)
    cumret = cumret.replace(-np.inf, np.nan)
    result_df.loc[0,"Sharpe"] = np.sqrt(252) * np.nanmean(cumret) / np.nanstd(cumret)
    result_df = result_df.round(2)

    print(result_df)
    result_df = result_df.transpose()
    trade_results = [result_df[i] for i in result_df]

    'sp500 pnl'
    select_st = "SELECT symbol, date, adjusted_close FROM [GSPC.INDX]"+    \
                " WHERE date >= " + "\"" + str(market_period_list[0]) + "\"" +      \
                " AND date <= " + "\"" + str(market_period_list[-1]) + "\"" + ";"
    sp_df = execute_sql_statement(select_st, engine)
    sp_df['ret'] = sp_df['adjusted_close'].pct_change()
    sp_df['cumpnl'] = capital * (1 + sp_df['ret']).cumprod() - capital
    sp_df.index = pd.to_datetime(sp_df.date)

    'plot'
    cumpnl.index = pd.to_datetime(cumpnl.index)
    maxdraw = pd.DataFrame(maxdraw, index=cumpnl.index)
    fig = plt.figure(figsize=(12,7))
    plt.title('Trading cumPnL '+str(market_period_list[0])+' to '+str(market_period_list[-1]),
              fontsize=15)
    plt.xlabel('Date')
    plt.ylabel('PnL (dollars)')
    plt.plot(cumpnl, label='pairs trading pnl')
```

```
964     plt.plot(maxdraw, label='maximum drawdown')
965     plt.plot(sp_df['cumpnl'], label='benchmark(sp500) pnl')
966     plt.legend()
967     plt.tight_layout()
968     fig.savefig('static/plots/trade_pnl.jpg')
969     plt.show()
970 
971     return render_template("real_trade.html", trade_list=trade_results)
972 
973 
974 
975 if(len(sys.argv) > 1) :
976     clientID = sys.argv[1]
977 else:
978     clientID = "Yicheng"
979 
980 HOST = socket.gethostbyname(socket.gethostname())
981 PORT = 6500
982 BUFSIZ = 1024
983 ADDR = (HOST, PORT)
984 
985 client_socket = socket.socket(AF_INET, SOCK_STREAM)
986 client_socket.connect(ADDR)
987 
988 
989 
990 if __name__ == "__main__":
991     market_period_list = []
992     price_data = []
993     record_order_df = pd.DataFrame()
994     ols_results = {}
995 
996     'real trade'
997     e = threading.Event()
998     q = queue.Queue()
999     client_thread = threading.Thread(target=join_trading_network, args=(e,q))
1000 
1001    'dashboard'
1002    bClientThreadStarted = False
1003    app.run()
```

Listing A.1: Platform Client

A.2 Platform Server

```
1  # -*- coding: utf-8 -*-
2  #!/usr/bin/env python3
3  
4  
5  import socket
6  from threading import Thread
7  import json
8  import urllib.request
9  import sys
10 import pandas as pd
11 import random
12 import sched, time
13 import datetime as dt
14 
15 from sqlalchemy import create_engine
16 from sqlalchemy import MetaData
17 
18 
19 serverID = "Server1"
20 
```

```
startDate = dt.datetime(2019,1,1)  # hours:minute:seconds
endDate = dt.date.today()  # only dates
requestURL = "https://eodhistoricaldata.com/api/eod/"
myEodKey = "5ba84ea974ab42.45160048"

' trading '
engine = create_engine('sqlite:///pairs_trading.db')
engine.execute("PRAGMA foreign_keys = ON")
metadata = MetaData()
metadata.reflect(bind=engine)  # bind to Engine, load all tables


def get_daily_data(symbol='', start=startDate, end=endDate, requestType=requestURL,
                   apiKey=myEodKey, completeURL=None):
    if not completeURL:
        symbolURL = str(symbol) + '?'
        startURL = "from=" + str(start)
        endURL = "to=" + str(end)
        apiKeyURL = "api_token=" + myEodKey
        completeURL = requestURL + symbolURL + startURL + '&' + endURL + '&' + apiKeyURL + '&
    period=d&fmt=json'
    print(completeURL)

    # if cannot open url
    try:
        with urllib.request.urlopen(completeURL) as req:
            data = json.load(req)
            return data
    except:
        pass


def accept_incoming_connections():
    while True:
        client, client_address = platform_server.accept()
        print("%s:%s has connected." % client_address)
        client_thread = Thread(target=handle_client, args=(client,))
        client_thread.setDaemon(True)
        client_thread.start()


def handle_client(client):
    """Handles a single client connection."""
    global symbols
    price_unit = 0.001
    client_msg = client.recv(buf_size).decode("utf8")
    data = json.loads(client_msg)
    print(data)
    clientID = data["Client"]
    status = data["Status"]
    msg_end_tag = ".$$$$"

    if status == "Logon":

        if (clientID in clients.values()):
            text = "%s duplicated connection request!" % clientID
            server_msg = "{\"Server\":\"" + serverID + "\", \"Response\":\"" + text + "\", \"
    Status\":\"Rejected\"}"
            server_msg = "".join((server_msg, msg_end_tag))
            client.send(bytes(server_msg, "utf8"))
            print(text)
            client.close()
            return

        else:
            text = "Welcome %s!" % clientID
            server_msg = "{\"Server\":\"" + serverID + "\", \"Response\":\"" + text + "\", \"
    Status\":\"Ack\"}"
```

```
            server_msg = "".join((server_msg, msg_end_tag))
            client.send(bytes(server_msg, "utf8"))
            clients[client] = clientID
            print (clients[client])
            client_symbols = list(data["Stocks"].split(','))
            symbols.extend(client_symbols)
            symbols = sorted(set(symbols))

    try:
        while True:
            msg = client.recv(buf_size).decode("utf8")
            data = json.loads(msg)
            print(data)

            if data["Status"] == "Quit":
                text = "%s left!" % clientID
                server_msg = "{\"Server\":\"" + serverID + "\", \"Response\":\"" + text + "\", \"Status\":\"Done\"}"
                print(server_msg)

            elif data["Status"] == "Order Inquiry":
                    if "Symbol" in data and data["Symbol"] != "":
                        server_msg = json.dumps(order_table.loc[order_table['Symbol'].isin(data["Symbol"])].to_json(orient='table'))

            elif data["Status"] == "New Order":
                if market_status == "Market Closed":
                    data["Status"] = "Order Reject"

                if  ((order_table["Symbol"] == data["Symbol"]) &
                    (order_table["Side"] != data["Side"]) &
                    (abs(order_table["Price"] - float(data["Price"])) < price_unit) &
                    (order_table["Status"] != 'Filled')).any():

                    mask = (order_table["Symbol"] == data["Symbol"]) & \
                            (order_table["Side"] != data["Side"]) & \
                            (abs(order_table["Price"] - float(data["Price"])) < price_unit) & \
                            (order_table["Status"] != 'Filled')
                    order_qty = order_table.loc[(mask.values), 'Qty']

                    if (order_qty.item() == data['Qty']):
                        order_table.loc[(mask.values), 'Qty'] = 0
                        order_table.loc[(mask.values), 'Status'] = 'Filled'
                        data["Status"] = "Fill"
                    elif (order_qty.item() < data['Qty']):
                        data['Qty'] = order_qty.items()  # return your quantity
                        order_table.loc[(mask.values), 'Qty'] = 0
                        order_table.loc[(mask.values), 'Status'] = 'Filled'
                        data["Status"] = "Order Partial Fill"
                    else:
                        order_table.loc[(mask.values), 'Qty'] -= data['Qty']
                        order_table.loc[(mask.values), 'Status'] = 'Partial Filled'
                        data["Status"] = "Order Fill"

                else:
                    if market_status == "Pending Closing":
                        order_table_for_pending_closing = order_table[(order_table["Symbol"] == data["Symbol"]) &
                                                                      (order_table["Side"] != data["Side"])].iloc[[0,-1]]
                        prices = order_table_for_pending_closing["Price"].values

                        if data["Side"] == "Buy":
                            price = float(prices[0])
                            price += 0.01
                        else:
                            price = float(prices[-1])
                            price -= 0.01
```

```
150                         data["Price"] = str(round(price,2))
151                         data["Status"] = "Order Fill"
152                     else:
153                         data["Status"] = "Order Reject"
154                 server_msg = json.dumps(data)
155 
156             elif data["Status"] == "User List":
157                 user_list = str('')
158                 for clientKey in clients:
159                     user_list += clients[clientKey] + str(',')
160                 server_msg = json.dumps({'User List':user_list})
161 
162             elif data["Status"] == "Stock List":
163                 #stock_list = symbols.str.cat(sep=',')
164                 stock_list = ','.join(symbols)
165                 server_msg = json.dumps({"Stock List":stock_list})
166 
167             elif data["Status"] == "Market Status":
168                 server_msg = json.dumps({"Server":serverID, "Market Status":market_status, "
    Market Period":market_period})
169 
170             else:
171                 text = "Unknown Message from Client"
172                 server_msg = "{\"Server\":\"" + serverID + "\", \"Response\":\"" + text + "\",
    \"Status\":\"Unknown Message\"}"
173                 print(server_msg)
174 
175             server_msg = "".join((server_msg, msg_end_tag))
176             client.send(bytes(server_msg, "utf8"))
177 
178             if data["Status"] == "Quit":
179                 client.close()
180                 del clients[client]
181                 users = ''
182                 for clientKey in clients:
183                     users += clients[clientKey] + ','
184                     print(users)
185                 return
186 
187     except KeyboardInterrupt:
188         sys.exit(0)
189 
190     except json.decoder.JSONDecodeError:
191         del clients[client]
192         sys.exit(0)
193 
194 clients = {}
195 
196 
197 def generate_qty(number_of_qty):
198     total_qty = 0
199     list_of_qty = []
200     for index in range(number_of_qty):
201         qty = random.randint(1,101)
202         list_of_qty.append(qty)
203         total_qty += qty
204     return (total_qty, list_of_qty)
205 
206 
207 def populate_order_table(symbols, start, end):
208     price_scale = 0.05
209     global order_index, order_table
210     order_table.drop(order_table.index, inplace=True)
211 
212     for symbol in symbols:
213         stock = get_daily_data(symbol, start, end)
214 
215         for stock_data in stock:
```

```
216             (total_qty, list_of_qty) = generate_qty(int((float(stock_data['high'])-float(
    stock_data['low']))/price_scale))
217             buy_price = float(stock_data['low']);
218             sell_price = float(stock_data['high'])
219             daily_volume = float(stock_data['volume'])
220 
221             for index in range(0, len(list_of_qty)-1, 2):
222                 order_index += 1
223                 order_table.loc[order_index] = [order_index, symbol, 'Buy', buy_price, int((
    list_of_qty[index]/total_qty)*daily_volume), 'New']
224                 buy_price += 0.05
225                 order_index += 1
226                 order_table.loc[order_index] = [order_index, symbol, 'Sell', sell_price, int((
    list_of_qty[index+1]/total_qty)*daily_volume), 'New']
227                 sell_price -= 0.05
228 
229     print(order_table)
230     print(market_status, market_period)
231 
232 
233 '''
234 (1) Server will provide consolidated books for 30 trading days,
235     (a) simulated from market data starting from 1/2/2019.
236     (b) Each simulated trading date has one book, with buy orders and sell orders
237         simulated from the high and low price from the day, with daily volume randomly
238         distributed cross all price points.
239     (c) Each simulated trading date starts with a new book simulated from corresponding
240         daily historical data
241 '''
242 def create_market_interest(index):
243     global market_period, symbols
244 
245     market_periods = pd.bdate_range('2019-01-02', '2019-04-01').strftime("%Y-%m-%d").tolist()
246 
247     # in order
248     startDate = market_periods[index]
249     endDate = market_periods[index]
250 
251     if len(order_table) == 0 or (market_status != "Market Closed" and market_status != "Pending
      Closing"):
252         market_period = startDate
253         populate_order_table(symbols, startDate, endDate)
254         print(market_status, "Creating market interest")
255     else:
256         print(market_status, "No new market interest")
257 
258 '''
259 (2) Each simulated trading day lasts 30 seconds,
260     following by 5 seconds of pending closing phase
261     and 5 seconds of market closed phase before market reopen
262 '''
263 def update_market_status(status, day):
264     global market_status
265     global order_index
266     global order_table
267 
268     market_status = status
269     create_market_interest(day)
270 
271     market_status = 'Open'
272     print(market_status)
273     time.sleep(30)
274 
275     market_status = 'Pending Closing'
276     print(market_status)
277     time.sleep(5)
278 
279     market_status = 'Market Closed'
```

```
280     print(market_status)
281
282     order_table.fillna(0)
283     order_index = 0
284     time.sleep(5)
285
286 '''
287 (3) There are 5 phases of market:
288     (a) Not Open, start
289     (b) Pending Open,
290     (c) Open,  30
291     (d) Pending Close, 5
292     (e) Market Closed 5
293 '''
294 def set_market_status(scheduler, time_in_seconds):
295     value = dt.datetime.fromtimestamp(time_in_seconds)
296     print(value.strftime('%Y-%m-%d %H:%M:%S'))
297
298     # 40s for one day
299     for day in range(total_market_days):
300         scheduler.enter(40*day+1,1, update_market_status, argument=('Pending Open', day))
301     scheduler.run()
302
303
304 port = 6500
305 buf_size = 1024
306 platform_server = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
307 print(socket.gethostname())
308 platform_server.bind((socket.gethostname(), port))
309
310 if __name__ == "__main__":
311
312     market_status = "Not Open"
313     market_period = "2019-01-01"
314     order_index = 0
315     total_market_days = 30
316
317     symbols = []
318     order_table_columns = ['OrderIndex', 'Symbol', 'Side', 'Price', 'Qty', 'Status']
319     order_table = pd.DataFrame(columns=order_table_columns)
320     order_table = order_table.fillna(0)
321
322     platform_server.listen(1)
323     print("Waiting for client requests")
324     time.sleep(80)  # wait for backtesting to finish
325
326     try:
327         scheduler = sched.scheduler(time.time, time.sleep)
328         current_time_in_seconds = time.time()
329         scheduler_thread = Thread(target=set_market_status, args=(scheduler,
    current_time_in_seconds))
330         scheduler_thread.setDaemon(True)
331
332         server_thread = Thread(target=accept_incoming_connections)
333         server_thread.setDaemon(True)
334
335         server_thread.start()
336         scheduler_thread.start()
337
338         scheduler_thread.join()  # wait until scheduler finished
339         server_thread.join()  # server finish after scheduler finished
340
341     except (KeyboardInterrupt, SystemExit):
342         platform_server.close()
343         sys.exit(0)
```

Listing A.2: Platform Client

Bibliography

[1] Principle component analysis.

[2] V. Bos and S. Mauw. *A LaTeX macro package for Message Sequence Charts—Maintenance document—Describing version* , June 2002. Included in MSC macro package distribution.

[3] V. Bos and S. Mauw. *A LaTeX macro package for Message Sequence Charts—Reference Manual—Describing version* , June 2002. Included in MSC macro package distribution.

[4] V. Bos and S. Mauw. *A LaTeX macro package for Message Sequence Charts—User Manual—Describing version* , June 2002. Included in MSC macro package distribution.

[5] C. A. Daniel Herlemont. Pairs trading, convergence trading, cointegration. 2003.

[6] J. Folger. Guide to pairs trading, 2018. [Online; accessed 08-May-2019].

[7] M. Goossens, S. Rahtz, and F. Mittelbach. *The LaTeX Graphics Companion*. Addison-Wesley, 1997.

[8] ITU-TS. ITU-TS Recommendation Z.120: Message Sequence Chart (MSC). Geneva, 1997.

[9] L. Lamport. *LaTeX—A Document Preparation System—User's Guide and Reference Manual*. Adsison-Wesley, 2nd edition, 1994. Updated for LaTeX 2_ε.

[10] E. Rudolph, P.Graubmann, and J. Grabowski. Tutorial on message sequence charts (MSC'96). In *FORTE*, 1996.

[11] R. SSEKUMA. A study of cointegration models with applications, 2011.

[12] Wikipedia contributors. Dbscan — Wikipedia, the free encyclopedia, 2019. [Online; accessed 08-May-2019].

[13] Wikipedia contributors. Stationarity process — Wikipedia, the free encyclopedia, 2019. [Online; accessed 08-May-2019].