Cryptocurrency Statistical Arbitrage Based on Cointegration and Preference Relations

Yingyao Liu, Feixue Ouyang, Xiaoling Huang, and Shijie Shao
School of Data Science
The Chinese University of Hong Kong, Shenzhen
{120090628, 121090427, 121090204, 120090491}@link.cuhk.edu.cn

Abstract

This paper presents an analysis of constructing cointegrated portfolios of cryptocurrencies, utilizing the Johansen cointegration test and Engle-Granger two-step approach. Our study focuses on the development and evaluation of two distinct cryptocurrency arbitrage strategies based on the identified cointegration relationships. We employ statistical tests to confirm the presence of cointegration among selected cryptocurrencies, namely Bitcoin (BTC), Ethereum (ETH), Bitcoin Cash (BCH), and Litecoin (LTC). Then, by trading the spread and assigning weights based on the cointegration relationship, we devised two trading strategies to capture arbitrage opportunities when the conintegration relationship is temporarily violated. The findings suggest the viability of cointegration-based arbitrage strategies in cryptocurrency markets, providing insights for investors and traders seeking to capitalize on price inefficiencies.

1 Introduction

Cryptocurrency markets have emerged as a dynamic and rapidly evolving domain, characterized by their volatility and potential for significant gains. Within this landscape, arbitrage—the practice of exploiting price differentials for profit—has garnered considerable attention from investors, traders, and researchers alike. [1] The allure of arbitrage lies in its promise of generating returns by capitalizing on price inefficiencies across various cryptocurrency exchanges. [2]

The emergence of Bitcoin and numerous alternative cryptocurrencies has opened up new avenues for investors and traders. Recent research conducted by Chuen et al. (2017) discovered minimal correlations between the Cryptocurrency Index (CRIX), a diversified portfolio of cryptocurrencies, and traditional assets. [3]This means that investing in the cryptocurrency market can, to a certain extent, mitigate the risks associated with traditional markets, making the portfolio risk more diversified.

In contrast to traditional equity markets, cryptocurrencies often exhibit strong correlations. For instance, the daily returns correlations among the major digital currencies—BTC, ETH, LTC, and BCH—exceeded 75 percent, as depicted in Figure 1. Due to the high volatility of cryptocurrencies, investments in cryptocurrencies typically adopt neutral trading strategies. The strong correlations among cryptocurrencies motivate us to explore statistical arbitrage strategies based on cointegration or mean reversion. [4]Currently, due to the absence of fundamental financial information for cryptocurrencies, most research on cryptocurrency trading is based on technical analysis or arbitrage exploiting price differentials across various exchanges. [5]

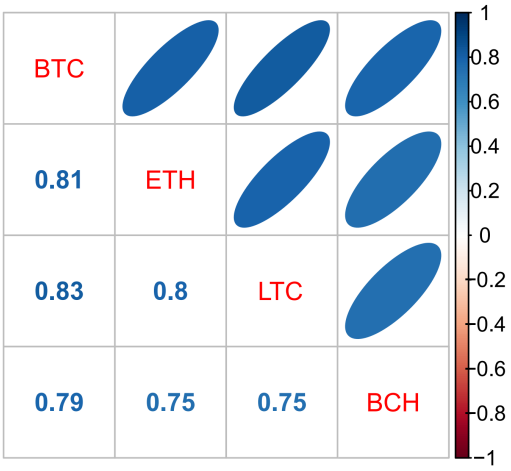

Figure 1: Correlation in daily returns of BTC, ETH, LTC, and BCH [4]

In this study, we obtained data from the Binance platform and processed it accordingly.We investigate the process of forming a cointegrated group of cryptocurrencies. This process entails conducting various statistical tests, including the Johansen cointegration test (Johansen, 1988), and employing the classical 2-step approach developed by Engle and Granger (1987). After establishing the cointegration relationships, we developed two distinct statistical arbitrage strategies based on the findings of our research. Our strategies enrich the application of statistical arbitrage in cryptocurrency investment.

2 Data Preparation

2.1 Data Collection

Binance is one of the leading cryptocurrency exchange platforms globally, founded in 2017. With millions of users worldwide, Binance offers a wide range of cryptocurrency trading services, including spot trading, futures trading, margin trading, among others, along with a variety of cryptocurrency financial derivatives and tools.

The data for our analysis is sourced from the Binance platform, covering both spot and futures trading. We obtained the data from the following two URLs:

- Spot: `https://data.binance.vision/data/spot/monthly/klines`
- Futures: `https://data.binance.vision/data/futures/um/monthly/klines`

The downloaded data primarily includes trading data for the following four major cryptocurrency assets paired against the US Dollar (USDT):

- Bitcoin (BTCUSDT)
- Ethereum (ETHUSDT)
- Litecoin (LTCUSDT)
- Bitcoin Cash (BCHUSDT)

These assets are among the most representative and liquid assets in the cryptocurrency market and are actively traded on the Binance platform.

Table 5 offers a preview of the downloaded data.

Table 1: Preview of BTCUSDT Spot Trading Data

	open_time	open	high	low	close	volume
0	2022-01-01 00:00:00	46216.93	46271.08	46208.37	46250.00	40.57574
1	2022-01-01 00:01:00	46250.00	46344.23	46234.39	46312.76	42.38106
2	2022-01-01 00:02:00	46312.76	46381.69	46292.75	46368.73	51.29955
...	...	...	...	...	...	...
1051117	2023-12-31 23:57:00	42240.92	42276.65	42240.92	42276.65	9.58764
1051118	2023-12-31 23:58:00	42276.64	42281.10	42276.64	42281.10	8.38641
1051119	2023-12-31 23:59:00	42281.10	42283.59	42258.94	42283.58	38.92904

2.2 Data Preprocess

In this section, we perform preprocessing steps on the collected trading data. The main data preprocessing tasks include calculating the differences between the opening and closing prices (`open_diff` and `close_diff`), as well as computing the returns.

- Opening and Closing Price Differences (open_diff, close_diff): We calculate the differences between the opening and closing prices for each observation in the dataset. The opening difference (open_diff_t) and closing difference (close_diff_t) for a given time period t are computed as follows:

$$\text{open\_diff}_t = \text{open}_t - \text{open}_{t-1}$$
$$\text{close\_diff}_t = \text{close}_t - \text{close}_{t-1}$$

  where open_t and close_t represent the opening and closing prices of the current time period t, respectively, and close_{t-1} represents the closing price of the previous time period.

- Returns: Additionally, we compute the returns for each observation. The return (Return_t) for a given time period t is calculated as the percentage change in price from the opening to the closing price, using the formula:

$$\text{return}_t = \frac{\text{close\_diff}_t}{\text{close}_{t-1}}$$

  where close_diff_t represents the closing difference for the current time period, and close_{t-1} represents the closing price of the previous time period.

These preprocessing steps are crucial for preparing the data for further analysis and modeling. Once completed, the preprocessed dataset will be ready for exploratory data analysis and model development.

3 Cointegration Test and Analysis

3.1 Engle-Granger Two-Step Method

Before we commence this section, it's important to note that the EG statistical tests herein utilize minutelevel data from January to February 2021. Our objective is to validate the cointegration relationships among cryptocurrencies for spread construction. This section is solely dedicated to discussing EG statistical tests and methodologies for construction, and any observed spread relationships do not necessarily imply strategies discussed later on.

The classical approach to cointegration testing, developed by Engle and Granger (1987), is known for its ability to capture long-term co-movements. [6] Initially, we conduct a linear regression analysis on some given $I(1)$ time series data, then we should test the stationarity of the residuals. We must ensure that the residuals from the OLS model are stationary; otherwise, we may encounter issues with spurious regression.In order to ensure stationarity, a series of unit-root tests are conducted on the residuals of the Ordinary Least Squares (OLS) model, encompassing the Augmented Dickey Fuller (ADF) test (Dickey and Fuller, 1979) [7], the Phillips-Peron (PP) test (Phillips and Perron, 1988), and the Kwiatkowski-Phillips-Schmidt-Shin (KPSS) test (Kwiatkowski et al., 1992) [8].

Given the objective of examining stationarity across price series, first-order price differences, and residuals derived from the OLS model, the testing hypotheses are formulated as follows:

- Augmented Dickey Fuller test and Phillips-Peron test:
  - H_0: Presence of unit root in observable price series
  - H_a: Unit root does not exist in observable price series
- Kwiatkowski-Phillips-Schmidt-Shin test:
  - H_0: Unit root does not exist in observable price series
  - H_a: Presence of unit root in observable price series

The Engle-Granger method involves a linear regression on given time series data. In this study, we use BTC time series data. In fact, the choice of which cryptocurrency to use as the dependent variable is inconsequential. We propose the OLS model as follow:

$$BTC_t = c + \beta_1 ETH_t + \beta_2 LTC_t + \beta_3 BCH_t + \epsilon_t$$

Once a cointegrating relationship has been established, a battery of three stationarity tests is conducted on the residuals. The prices of these cryptocurrency assets are considered cointegrated only if they all exhibit $I(1)$ processes and the residuals ϵ_t from the above Ordinary Least Squares (OLS) model are stationary.

After establishing the cointegrating relationship, we perform 3 stationarity tests on the residuals. We can see the results in Table 2. According to the result, we accept the null hypothesis (there exits unit-root in time series) from both ADF and PP tests. It shows that the daily opening prices of all 4 cryptocurrencies are non-stationary. From the KPSS test, we should reject the null hypothesis which again indicates that daily opening prices of all 4 cryptocurrencies are non-stationary. However, after the first differencing, the opening prices of all 4 cryptocurrencies become stationary. Therefore, we can conclude that the opening prices of BTC, ETH, LTC, and BCH are $I(1)$ processes.

Table 2: Summary of p-values from stationarity tests on cryptocurrencies' prices and their first order differences

	BTC_t	ETH_t	LTC_t	BCH_t	ΔBTC_t	ΔETH_t	ΔLTC_t	ΔBCH_t
ADF-test	0.50	0.35	0.18	0.32	0.0	0.0	0.0	0.0
PP-test	0.48	0.32	0.18	0.30	0.0	0.0	0.0	0.0
KPSS-test	0.01	0.01	0.01	0.1	0.1	0.1	0.1	0.1

We establish linear relationships between BTC with ETH, LTC and BCH,see Figure 2. Notably, all the regressing coefficients appear to be statistically significant with p-values less than 1%. The adjusted R-squared is 70.9%, which is relatively high.

	Dependent variable:
	BTC
ETH	13.40^{***}
	(0.075)
LTC	46.02^{***}
	(0.588)
BCH	6.55^{***}
	(0.172)
Constant	0.052
	(0.134)
Observations	84880
R^2	0.709
Adjusted R^2	0.709
F Statistic	68840^{***} (df = 3; 84876)
Note:	*p<0.1; **p<0.05; ***p<0.01

Figure 2: Ordinary least square regression model for BTC

From Figure above, we obtain the following linear model:

$$BTC_t = 0.052 + 13.40ETH_t + 46.02LTC_t + 6.55BCH_t + \epsilon_t$$

Next we conduct stationarity tests on the residuals. The result is showed as Figure 4. The results from all three stationarity tests affirm the stability of residuals derived from the linear regression model, indicating their stationary nature over time. Consequently, our approach suggests that the prices of BTC, ETH, LTC, and BCH are inherently cointegrated. With this validation, one can proceed to construct the spread utilizing the proposed linear coefficients:

$$SPREAD_t = BTC_t - 13.40ETH_t - 46.02LTC_t - 6.55BCH_t$$

Test	Value
ADF Statistic	-44.34199063655972
p-value	0.0
PP Statistic	-290.46719023608136
p-value	0.0
KPSS Statistic	0.17213444373332498
p-value	0.1

Figure 3: Stationarity tests on residuals

3.2 Johansen Method

A general vector autoregressive model is similar to the AR(p) model except that each quantity is vector valued and matrices are used as the coefficients. The general form of the VAR(p) model, without drift, is given by:

$$x_t = \mu + A_1 x_{t-1} + A_2 x_{t-2} + \ldots + A_p x_{t-p} + w_t$$

, where μ is the vector-valued mean of the series, A_i are the coefficient matrices for each lag and w_t is a multivariate Gaussian noise term with mean zero.

At this stage we can form a Vector Error Correction Model (VECM) by differencing the series:

$$\Delta x_t = \mu + \Pi x_{t-1} + \Gamma_1 \Delta x_{t-1} + \ldots + \Gamma_{p-1} \Delta x_{t-p+1} + w_t$$

where $\Delta x_t = x_t - x_{t-1}$ is the differencing operator, A is the coefficient matrix for the first lag and Γ_i are the matrices for each differenced lag.

The Johansen test checks for the situation of no cointegration, which occurs when the matrix $A = 0$. The Johansen test is more flexible than the CADF procedure outlined in the previous article and can check for multiple linear combinations of time series for forming stationary portfolios.

To achieve this an eigenvalue decomposition of A is carried out. The rank of the matrix A is given by r and the Johansen test sequentially tests whether this rank r is equal to zero, equal to one, through to $r = n - 1$, where n is the number of time series under test.

The null hypothesis of $r = 0$ means that there is no cointegration at all. A rank $r > 0$ implies a cointegrating relationship between two or possibly more time series.

The eigenvalue decomposition results in a set of eigenvectors. The components of the largest eigenvector admits the important property of forming the coefficients of a linear combination of time series to produce a stationary portfolio.

And another approach of Johansen test is using the trace statistics of matrix A, which can achieve similar results as the eigenvalue approach.But according to Hjalmarsson, Erik and Osterholm, Par [9], Overall, the performance of the trace test appears worse than that of the maximum eigenvalue test. Both tests, however, have large enough deviations from the nominal size that practitioners should be aware of the problems associated with Johansen's procedures under these circumstances.

The proposed sequence of additional tests helps alleviate some of the sensitivity of the Johansen procedures to deviations from the strict unit-root assumption. They do not, however, eliminate the problem.

As an example, we can do the Johansen test with respect to the futures data with trace statistics:

- Trace statistics:

$$\begin{bmatrix} 61.8577 \\ 18.8316 \\ 8.5903 \\ 2.1639 \end{bmatrix}$$

- Trace critical values:

$$\begin{bmatrix} 44.4929 & 47.8545 & 54.6815 \\ 27.0669 & 29.7961 & 35.4628 \\ 13.4294 & 15.4943 & 19.9349 \\ 2.7055 & 3.8415 & 6.6349 \end{bmatrix}$$

- Hypothesis test using trace statistics:
  - Hypothesis 0: $r = 0$ (No cointegrating relationship)
    trace statistic: 61.8577 > critical value: 54.6815
    $\Rightarrow$ Reject hypothesis 0 at 99%
  - Hypothesis 1: $r <= 1$ (At most 1 cointegrating relationship)
    trace statistic: 18.8316 < critical value: 27.0669
    $\Rightarrow$ Hypothesis 1 can't be rejected
  - We claim that there exists one integrating relationship

We can also do the Johansen test with respect to the futures data with eigenvalue statistics:

- Eigenvalue statistics:

$$\begin{bmatrix} 43.0260 \\ 10.2413 \\ 6.4263 \\ 2.1639 \end{bmatrix}$$

- Eigenvalue critical values:

$$\begin{bmatrix} 25.1236 & 27.5858 & 32.7172 \\ 18.8928 & 21.1314 & 25.865 \\ 12.2971 & 14.2639 & 18.52 \\ 2.7055 & 3.8415 & 6.6349 \end{bmatrix}$$

- Hypothesis test using trace statistics:
  - Hypothesis 0: $r = 0$ (No cointegrating relationship)
    trace statistic: 43.0260 > critical value: 32.7172
    $\Rightarrow$ Reject hypothesis 0 at 99%
  - Hypothesis 1: $r <= 1$ (At most 1 cointegrating relationship)
    trace statistic: 10.2413 < critical value: 18.8928
    $\Rightarrow$ Hypothesis 1 can't be rejected
  - We claim that there exists one cointegrating relationship

And after making sure the existence of cointegrating relationship, we can construct the spread.

- Normalized eigenvector matrix:

$$\begin{bmatrix} 1.0000e+00 & 1.0000e+00 & 1.0000e+00 & 1.0000e+00 \\ 3.0900e+00 & -2.0389e+01 & -5.1609e+01 & -3.9532e+00 \\ 1.9015e+02 & 4.1910e+01 & 1.3235e+03 & -2.2238e+02 \\ -1.7479e+02 & 2.9793e+01 & 3.1638e+01 & 1.9089e+01 \end{bmatrix}$$

- Construct spread using the 1st column of normalized eigenvector matrix:

$$\text{spread} = \text{BTC} + 3.09 \times \text{ETH} + 190.16 \times \text{LTC} + 174.80 \times \text{BCH}$$

- Augmented Dickey–Fuller test of constructed spread
  - ADF statistics: -6.27
  - ADF p-value: 0.00
  - ADF lags: 1
  - ADF number of observations: 525598
  - ADF critical values: {1%: -3.4303, 5%: -2.8615, 10%: -2.5667}
  - Stationary: True

Therefore, the Johansen test provides us a proof that the ointegration relationship exists, and the subsequent ADF test convinces us that the residual/spread is stationary, which can be used for further trading.

3.3 Portfolio Time series

The main purpose of our cointegration testing procedure is to construct a tradable mean reverting portfolio. From a trading perspective, periodic movements or fluctuations that lead to frequent crossing of the equilibrium levels from both directions are desired, especially given that entry and exit rules are based on deviations from the mean price. For this reason, figure suggest that model from Engle-Granger Method is practical to trade.

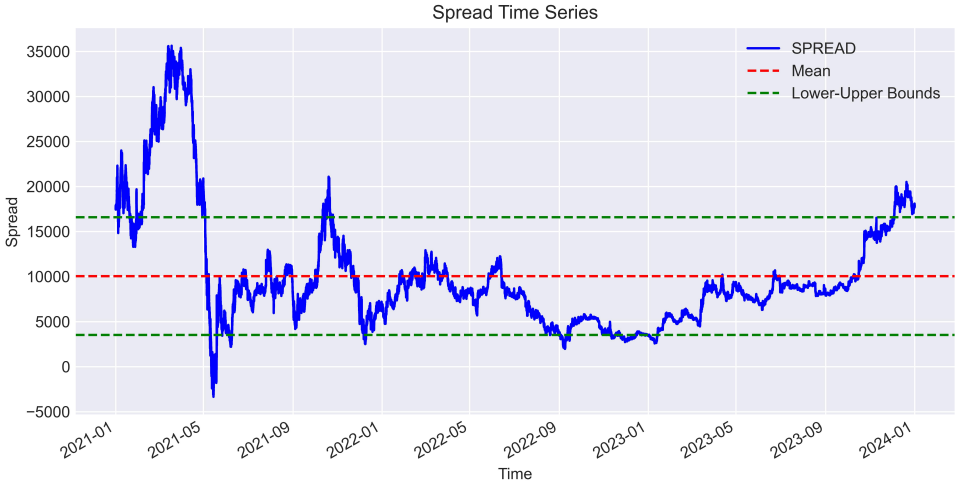

Figure 4: Spread Time series

Next, we construct a time series model to capture dynamic of the spread in order to design the trading rules. Generally, we expect the spread to be highly mean-reverting since that is the main criterion we used to choose the 4 crypto-assets in the first place. In discrete time, the spread is often assumed to be stationary Autoregressive Moving Average (ARIMA) processes since they are also mean-reverting by design. This is the basic requirement we set forth to designing our trading signals.To estimate the orders and parameters of an ARMA process, we rely on sample autocorrelation function (ACF) and partial autocorrelation function (PACF) plots.As suggested by ACF and PACF plots respectively in Figure 5, the residuals can be modeled by ARIMA(1,0,1). We summarize the test in Figure 6.

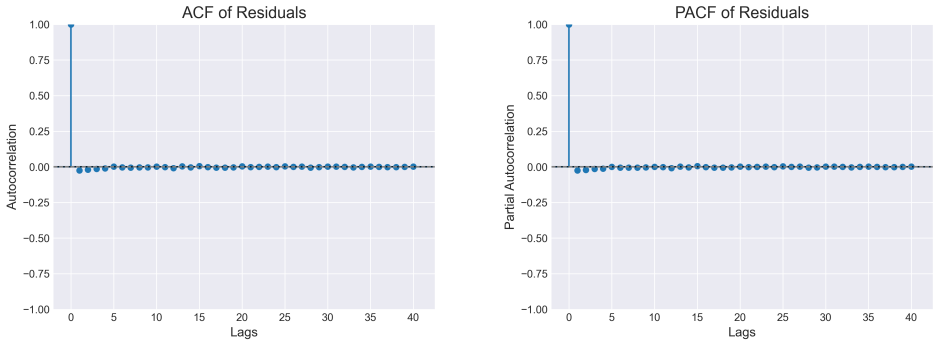

Figure 5: ACF and PACF graph of residual

	Dependent variable:
	residuals
AR(1) Coeff.	0.7008^{***}
	(0.003)
MA(1) Coeff.	-0.7278^{***}
	(0.003)
Intercept	$-4.24e{-6}$
	(0.010)
Observation	1,051,119
Log Likelihood	$-4,052,718.86$
σ^2	130.7534
Akaike Inf. Crit.	8,105,445.720
Note:	*p<0.1; **p<0.05; ***p<0.01

Figure 6: Fitting the spread to the ARIMA(1, 0, 1) model

To ensure that ARMA(1,0,1) is an appropriate model, Ljung-Box test is performed on residuals from the ARIMA model to check for the overall randomness and no autocorrelation in the residual time series. Ljung-Box test hypothesis: Ho: Price series are independent, or no serial correlations Ha: Price series are not independent, or they exhibit serial correlations The p-value of 0.44 for the Ljung-Box test on residuals implies the failure to reject the null hypothesis that there is no autocorrelation in the residuals. We conclude that prices of the spread are ARIMA(1,0,1) processes.

4 Statistical Arbitrage Strategies

In this section, we explored 2 trading strategies to capture arbitrage opportunities. In Section 4.1, we directly trade the spread constructed by linear regression when its value significantly deviates from the mean value. In Section 4.2, we make use of the spreads between pairs to construct preference relationships between assets, and assigned weights accordingly to each asset.

In order to make this backtest closer to real market trading, we set the transaction cost to be 0.01%.

4.1 Trading the spread

4.1.1 Strategy validation with full-sample

In trading practices, finding the right times for entry and exit is key to a profitable trading system. As previously mentioned, the default entry and exit thresholds in back-testing strategy are typically set around 1 deviation, denoted by σ, above and below the mean spread level, assuming that traders can go both long and short on the spread. It is crucial to assure that these entry and exit levels are set

to maximize profits in terms of the trading costs and transaction frequency. Thus, by back-testing a trading system with multiple entry/exit levels, we can get a sense of how profitable the system is by looking at different performance criteria such as profit and loss, Sharpe ratio, and more. A mean reversion strategy for trading the spread is described by the following linear combination:

$$SPREAD_t = -\beta_1 ETH_t - \beta_2 LTC_t - \beta_3 BCH_t + BTC_t$$

Different from traditional equity market, traders can purchase fractional shares up to 8 decimal places in the cryptocurrency market. Thus we do not have a requirement to purchase integer shares of cryptocurrency. Upon receiving a signal to open a position, we will either go long or short on the maximum number of spreads based on the cash we hold. If there is insufficient cash available, we will allow the cash balance to go negative, indicating borrowed funds, and simultaneously go long on one unit of the spread. For short positions, the borrowed portfolio value will not exceed the current cash balance. Similarly, if cash is too scarce to short one unit of the spread without exceeding leverage limits, we will still choose to short one unit. This approach allows us to maximize the use of funds when capital is abundant and to execute trades when capital is scarce. In this case, we are only concerned with whether the trading strategy is theoretically feasible, so we use the full sample to compute the mean and standard deviation, and do not take into account extraterritorial costs such as transaction costs. Let us now state our trading rule:
(i) Long the spread/Exit short position when

$$SPREAD_t < \mu + c\sigma$$

(ii) Short the spread/Exit long position when

$$SPREAD_t > \mu - c\sigma$$

where c is a multiple we choose to set our entry/exit thresholds.

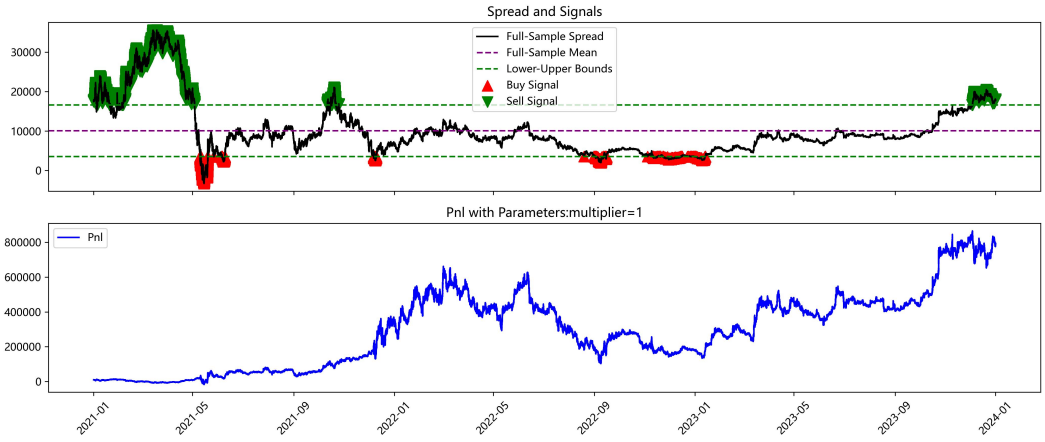

Figure 7: Trading strategy with full sample

From the figure, we observe that able to make profits during the mean-reverting process of spread. Then we summarize some trade statistics to evaluate the performance of our strategy:

Table 3: Statistics for trading strategies

Total Trades	Average Win	Max Equity	Min Equity	Win Rate	Drawdown
4	210109	865371	-16933	790266	558709

This strategy is very robust, with extremely high win rates, very low drawdown, and basically steady profits every time you open a position.However, to apply this trading strategy to real trading, we need to consider a few more points. First, we need information from past samples to estimate the long-term mean and standard deviation. Second, the cointegration relation may be adjusted as the market changes. And, the optimal trading threshold c needs to be further determined. Finally, we need to consider transaction costs.

4.1.2 Dynamic estimation of cointegration, mean and standard deviation

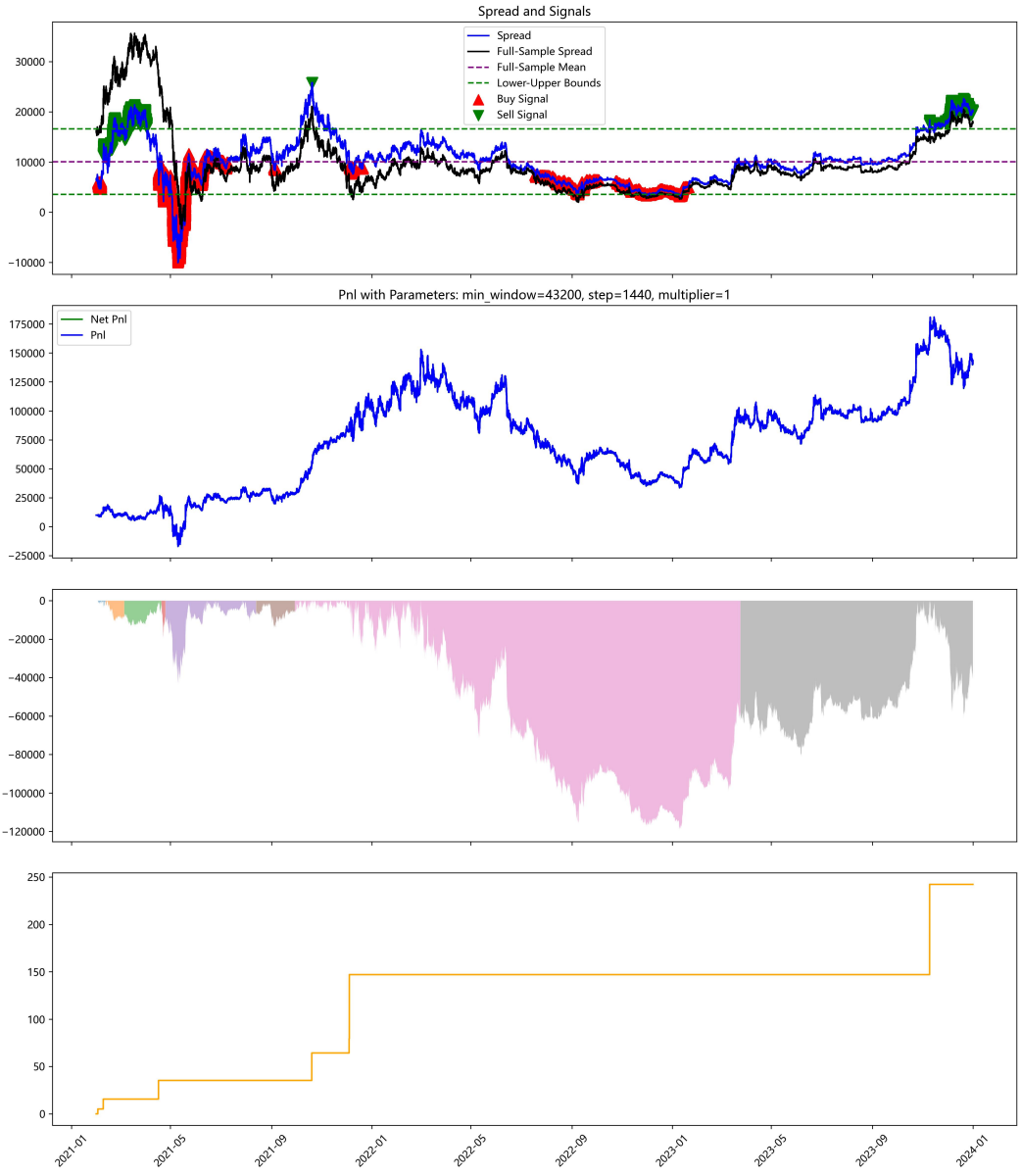

Figure 8: Trading strategy

To take slippage cost into consideration, we use VWAP(Volume-Weighted Average Price) to represent real transaction price. The formula of VWAP is given:

$$\text{VWAP} = \frac{\sum(\text{Price}_t \times \text{Volume}_t)}{\sum \text{Volume}_t}$$

To ensure robust estimation of cointegration and associated statistics such as mean and standard deviation, we implement an expanding window methodology. Initially, we define a minimum window length, which is critical for accurate parameter estimation. Within this window, we estimate the cointegration relationship among the cryptocurrencies and compute the spread. From this, we derive the mean and standard deviation of the spread, which are essential for signal generation.

Using the cointegration coefficients obtained from the analysis, we project future spreads. To enhance the computational efficiency of our model, we introduce a parameter 'step', which determines the

prediction interval for the spread. After each prediction, we expand the window to include the data from this interval before commencing the next cycle. This approach not only adapts to market changes by incorporating new data continuously but also ensures that each estimation leverages the maximum available data for accuracy.

To find the best threshold c for trading, we conduct a simple grid search to find best parameter. The summary statistics is given:

Table 4: Statistics for trading strategies with different entry and exit thresholds

Threshold level	0.5σ	0.7σ	σ	1.3σ	1.5σ
Num.Trades	5	5	5	3	3
Win Rate	100	100	100	100	100
Average Win	5117	8730	30839	17324	46271
Max Equity	39501	58190	180754	80506	182924
Min Equity	-6828	-4180	-16992	-27208	-68409
Final Equity	24306	43078	142486	60716	154661
Drawdown	26533	39800	119402	68124	158956

4.2 Construct portfolio from preference relations

4.2.1 Model Setup

In order to further optimize the arbitrage portfolio, we managed to estimate the utility of each asset from their preference relationships, and hence decide their weights accordingly. [10] The detailed setting is as follows:

We define the preference function $\rho^* : S \times S \to \mathbb{R}$ that describes the preference relationship between a pair. For instance, $\rho^*(s_i, s_j) > 0$ implies that s_i is preferred to s_j, and hence we tend to long s_i and short s_j. Based on ρ^*, we can derive a latent utility function u^* such that the following condition is satisfied:

$$\rho^*(s_i, s_j) = u^*(s_i) - u^*(s_j).$$

Additionally, more assumptions on ρ^* and u^* can be derived:

$$\rho^*(s_i, s_i) = u^*(s_i) - u^*(s_i) = 0, \tag{1}$$

$$\rho^*(s_i, s_j) = -\rho^*(s_j, s_i), \tag{2}$$

$$(\rho^*(s_i, s_j) > 0) \wedge (\rho^*(s_j, s_k) > 0) \Rightarrow (\rho^*(s_i, s_k) > 0). \tag{3}$$

This implies that the ranking among assets are transitive.

Considering all asset pairs, we can express it in a matrix form:

$$\mathbf{B}\mathbf{u}^* = \rho^*$$

$$\mathbf{B} = \begin{bmatrix} 1 & -1 & 0 & 0 \\ 1 & 0 & -1 & 0 \\ 1 & 0 & 0 & -1 \\ 0 & 1 & -1 & 0 \\ 0 & 1 & 0 & -1 \\ 0 & 0 & 1 & -1 \end{bmatrix}$$

We can easily estimate ρ^* from the deviation of pair spreads, but this does not guarantee the transitivity of the preference relationship described in (1)(2)(3). Hence, we define $\rho : S \times S \to \mathbb{R}$ as the empirically estimated preference function, i.e., the price spread. By solving the following optimization problem, we can obtain the preference function ρ^* and utility function u^*:

$$\rho^* = \arg \min_{\rho' \in P^*} d(\rho', \rho)$$

$$\text{s.t.} \quad Bu^* = \rho^*$$

$$\sum_{i=0}^{N} u_i^* = 0$$

The authors of the original paper derived the analytical solutions as follows, and we have displayed the detailed derivation in Appendix A:

$$\mathbf{u}^* = \frac{1}{N}\mathbf{B}^T\rho$$

According to the sign of the utility function u^*, we can decide the trading direction of each asset, i.e., long the positive and short the negative. Moreover, we can further assign the weights for each asset as follows:

$$\begin{cases} s_i \in T_L & \text{if } u^* > 0, \\ s_i \in T_S & \text{if } u^* < 0, \end{cases}$$

$$w(s_i) = \begin{cases} \frac{|u^*(s_i)|}{\sum_{s_j \in T_L} |u^*(s_j)|} & \text{if } s_i \in T_L, \\ \frac{|u^*(s_i)|}{\sum_{s_j \in T_S} |u^*(s_j)|} & \text{if } s_i \in T_S, \\ 0 & \text{otherwise}, \end{cases}$$

4.2.2 Experiment 1

The steps to realize this trading strategy is as follows:

1. Calculate the preference functions $\rho(si, sj)^(t)$ for all pairs (si, sj). Here, we take the negative standardized spread between pairs as the preference function, i.e.:

$$\rho(s_i, s_j)^{(t)} = -\frac{c_{i,j}^{(t)} - \mu_{i,j}^{(t)}}{\sigma_{i,j}^{(t)}}$$

where $c_{i,j}^{(t)}$ is the price spread between securities s_i and s_j at timestamp t, and the parameters $\mu_{i,j}^{(t)}$ and $\sigma_{i,j}^{(t)}$ are the mean and the standard deviation of the price spread $c_{i,j}^{(t)}$, respectively. The price spread between securities s_i and s_j at particular timestamp t is defined as:

$$c_{i,j}^{(t)} = \log\left(\frac{p_i^{(t)}}{p_j^{(t)}}\right)$$

where $p_i^{(t)}$ and $p_j^{(t)}$ denote historical prices of securities s_i and s_j at the time t respectively. This corresponds to the commonly used log-return between these two prices – due to the logarithm the spreads are centered around zero and symmetric. In [**?**], the price spread is calculated as the log ratio of normalized price series. However, we normalize the deviation from the expected price spread, not the prices. This modification ensures that the preference functions of different security pairs are comparable, alleviating the problem of choosing the thresholding parameter κ.

The parameters $\mu_{i,j}^{(t)}$ and $\sigma_{i,j}^{(t)}$ are estimated on a lookback period $[t-T, t]$ using an unbiased estimator:

$$\mu_{i,j}^{(t)} = \frac{1}{T-1}\sum_{\tau=t-T}^{t-1} c_{i,j}^{(\tau)}$$

$$\sigma_{i,j}^{(t)} = \sqrt{\frac{1}{T-2}\sum_{\tau=t-T}^{t-1}(c_{i,j}^{(\tau)} - \mu_{i,j}^{(t)})^2}$$

2. Calculate the utility function of each asset from the previous optimization program.

3. Construct the portfolios using the weight allocation scheme.

We calculate the optimized weight every 1 minute using data from the previous 60 minutes, and the backtest performance is as follows:

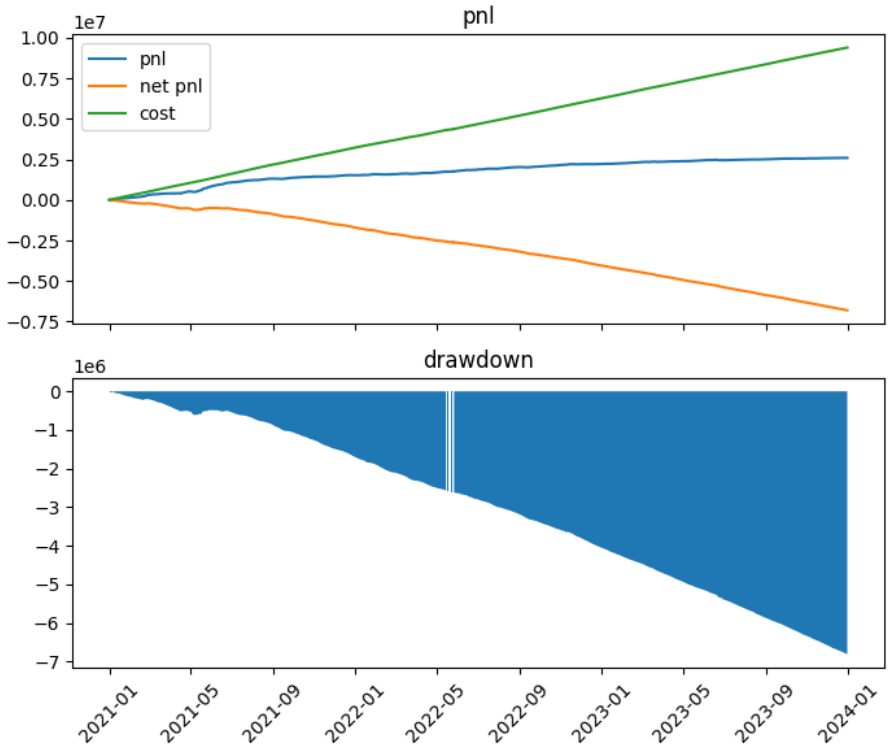

Figure 9: Backtest Result of Experiment 1

Although the pnl of this strategy is upward sloping, the transaction cost of this strategy is much higher than the pnl, resulting in a deeply negative net pnl. This implies that although this strategy is able to caputure the correct arbitrage direction, its profit capability is not able to cover the high transaction costs of this high frequency trading. This suggests that we should further optimize the trading signal and lower the trading frequency to reduce transaction costs.

4.2.3 Experiment 2: Lower Trading Frequency

In order to lower trading frequency, we set the portfolio rebalancing period to a longer time frame, and set the new target position as the mean of the minute-level target position over the longer time frame. Moreover, we tested the effects of different estimation window for the preference function as well. After multiple experiments, we discovered that 3 hours is the best rebalancing period, and the optimal look-back window is 360 minutes. The backtest result of the optimized portfolio is as follows:

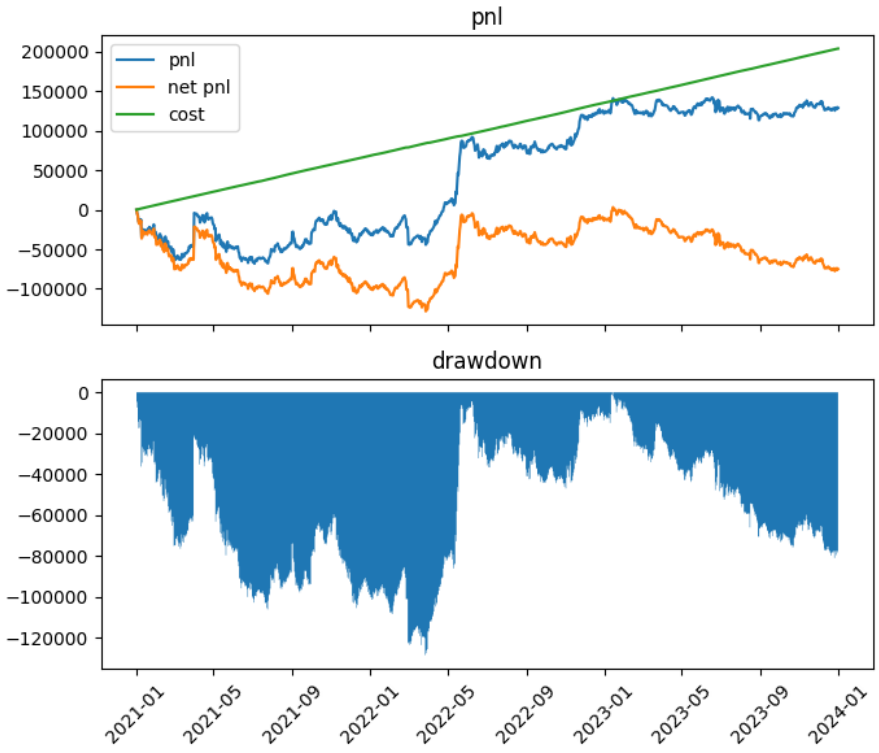

Figure 10: Backtest Result of Experiment 2

In this case, although the transaction cost has been sharply decreased, it is still higher than the pnl, yielding a negative pnl. Therefore, it is critical to further optimize the profitability of this strategy.

4.2.4 Experiment 3: Discard Insignificant Preference Relationship

In order to optimize trading signal, we reflect on the preference relationship. Recall that we regard the negative standardized spread between pairs as the preference function. For instance, when the standardized spread between s_i and s_j is 1.69, their preference function value is -1.69. This implies that the spread between s_i and s_j is abnormally high, hence we should short s_i and long s_j. However, when looking back at the data, we discovered that the preference function value is very small. This suggests that the spread between s_i and s_j is falling at normal intervals, hence no significant preference relationship exists between s_i and s_j. Hence, we decided to recalculate the preference function rho as follows:

$$\rho'(s_i, s_j)^{(t)} \begin{cases} \rho(s_i, s_j)^{(t)} & \text{if } |\rho(s_i, s_j)^{(t)}| > 1 \\ 0 & \text{if } |\rho(s_i, s_j)^{(t)}| <= 1, \end{cases}$$

After this adjustment, we backtested this strategy on different sets of parameters. We discovered that 2 hours is the best rebalancing period, and the optimal look-back window is 120 minutes. The backtest result is as follows:

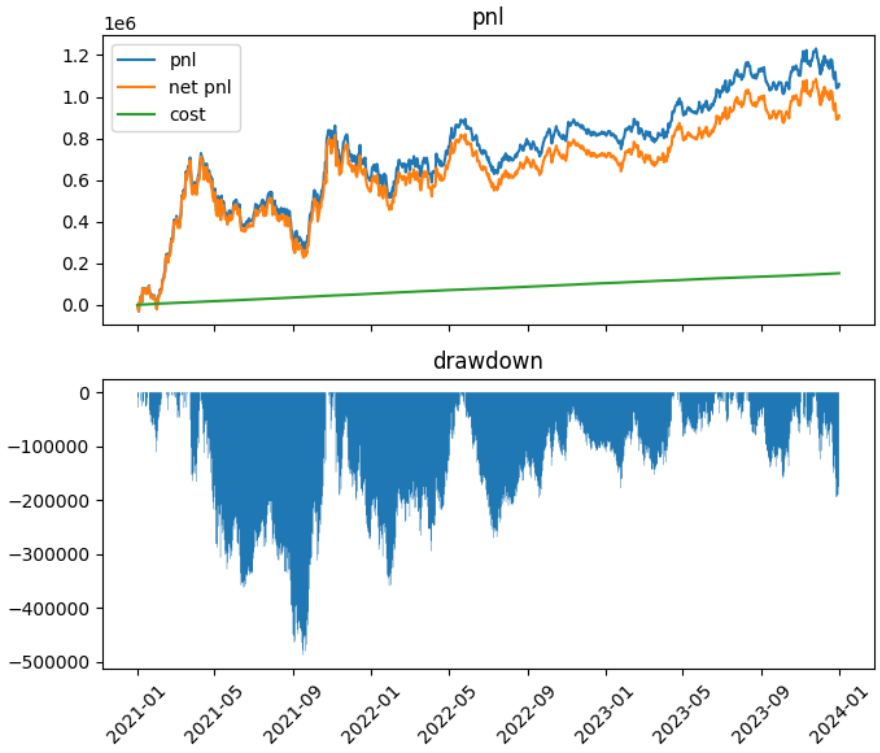

Figure 11: Backtest Result of Experiment 3

In this case, the transaction cost is controlled to an acceptable range, and we finally yield a positive net pnl. This implies that this regularization on the preference function significantly improves the efficiency of detecting arbitrage opportunities.

In summary, the statistics of the three above mentioned strategies are as follows:

Table 5: Statistics for trading strategies

Category	Total Trades	Annualized Return	Sharpe Ratio	max drawdown
Experiment 1	1563414	-2267.62%	-	7054.75%
Experiment 2	8647	-24.97%	-	128.45%
Experiment 3	5676	303.13%	1.29	59.87%

5 Conclusion

In this project, we successfully confirmed the existence of cointegration relationship among the selected cryptocurrencies by Engle-Granger test and Johansen test. Based on this discovery, we designed two trading strategies to capture arbitrage opportunities. For the first strategy, we construct mean-reverting portfolio based on cointegration relation, and design trading signal by estimating the mean and standard deviation of the spread between multiple cryptocurrency assets. By applying grid search, we optimized a threshold to build a portfolio that grew 14 times in three years. For the second strategy, we estimated the ranked utility of each asset based on spreads between pairs, and assigned weights accordingly. Although this strategy initially suffers from high transaction cost, we overcame this challenge by lowering trading frequency and regularizing the preference function between assets, achieving a sharpe ratio of 1.29.

6 Group member contributions

- Yingyao Liu: Implemented Strategy 2 (Section 4.2)

- Feixue Ouyang: Implemented Strategy 1 (Section 4.1), Engle-Granger Test(Section 3.1)
- Xiaoling Huang: Implemented Engle-Granger Test, Johansen Test(Section 3), Introduction & abstract
- Shijie Shao: Implemented Data preparation(Section 2), Johansen Test(Section 3.2), Strategy 1 (Section 4.1)

References

[1] Giancarlo Giudici, Alistair Milne, and Dmitri Vinogradov. Cryptocurrencies: market analysis and perspectives. *Journal of Industrial and Business Economics*, 47:1–18, 2020.

[2] Stephen Chan, Jeffrey Chu, Saralees Nadarajah, and Joerg Osterrieder. A statistical analysis of cryptocurrencies. *Journal of Risk and Financial Management*, 10(2):12, 2017.

[3] Jamie Kang and Tim Leung. Asynchronous adrs: overnight vs intraday returns and trading strategies. *Studies in Economics and Finance*, 34(4):580–596, 2017.

[4] Tim Leung and Hung Nguyen. Constructing cointegrated cryptocurrency portfolios for statistical arbitrage. *Studies in Economics and Finance*, 36(3):581–599, 2019.

[5] Igor Makarov and Antoinette Schoar. Trading and arbitrage in cryptocurrency markets. *Journal of Financial Economics*, 135(2):293–319, 2020.

[6] Robert F Engle and Clive WJ Granger. Co-integration and error correction: representation, estimation, and testing. *Econometrica: journal of the Econometric Society*, pages 251–276, 1987.

[7] David A Dickey and Wayne A Fuller. Distribution of the estimators for autoregressive time series with a unit root. *Journal of the American statistical association*, 74(366a):427–431, 1979.

[8] Denis Kwiatkowski, Peter CB Phillips, Peter Schmidt, and Yongcheol Shin. Testing the null hypothesis of stationarity against the alternative of a unit root: How sure are we that economic time series have a unit root? *Journal of econometrics*, 54(1-3):159–178, 1992.

[9] Erik Hjalmarsson and Pär Österholm. Testing for cointegration using the johansen methodology when variables are near-integrated. 2007.

[10] Stjepan Beguv, Andro Mer'cep, Zvonko Kostanjv, et al. Statistical arbitrage portfolio construction based on preference relations. Technical report, 2023.

A Appendix: Potential Method – Derivation and Implementation Details

Let $\{s_1, \ldots, s_N\}$ be an indexed set of N securities. The potential method defines a preference function $\rho^* : S \times S \to \mathbb{R}$ as the one for which there exists a utility u^* which satisfies the condition:

$$\rho^*(s_i, s_j) = u^*(s_i) - u^*(s_j) \quad \forall s_i, s_j \in S,\, i < j. \tag{A.1}$$

It is clear that the preference function ρ^* needs to be asymmetric, i.e., $\rho^*(s_i, s_j) = -\rho^*(s_j, s_i)$. This implies that the preference function needs to be evaluated for $\binom{N}{2}$ pairs (elements of $S \times S$) rather than N^2 pairs. The Equation (A.1) can be written in matrix form as:

$$\rho^* = Bu^* \tag{A.2}$$

where u^* is the $N \times 1$ vector of security utilities, ρ^* is the $\binom{N}{2} \times 1$ vector of pairwise preferences, and B is the $\binom{N}{2} \times N$ graph incidence matrix.

If there is no utility vector u^* for which the constraint (A.2) is satisfied given ρ, then the function ρ is not a preference function and an approximation of ρ needs to be found for which (A.2) will hold for some u^*. This task can be expressed as an optimization problem:

$$\rho^* = \arg\min_{\rho' \in P} d(\rho', \rho) \tag{A.4}$$

$$\text{s.t. } Bu^* = \rho^* \tag{A.5}$$

$$\sum_{i=0}^{N} u_i^* = 0 \tag{A.6}$$

where P is the set of all preference vectors ρ^* and d is a distance function. The constraint $\sum_{i=0}^{N} u_i^* = 0$ is needed in order to obtain a unique solution, because a preference relation obtained from a utility function is invariant to scaling, i.e., $\forall s_i, s_j \in S, \forall \alpha \in \mathbb{R}^+,\, u^*(s_i) \approx \alpha u^*(s_i)$.

$$B^T B u^* = B^T \rho \tag{A.7}$$

$$\sum_{i=0}^{N} u_i^* = 0. \tag{A.8}$$

Equations (A.7) and (A.8) can be added together to get:

$$(B^T B + J^T) u^* = B^T \rho, \tag{A.9}$$

where J is a matrix of ones with the same dimension as $B^T B$. Finally solving (A.9) for u^* results in:

$$u^* = (B^T B + J)^{-1} B^T \rho, \tag{A.10}$$

which can be further simplified by employing the fact that $B^T B$ is the Laplacian of a complete graph which implies that term $(B^T B + J)^{-1} = \frac{1}{N} I$. Thus the final solution to the Equation (A.9) is:

$$u^* = \frac{1}{N} B^T \rho. \tag{A.11}$$

The preference vector ρ^* can be calculated by simply plugging u^* back into Equation (A.2):

$$\rho^* = Bu^*. \tag{A.12}$$

A Appendix: codes for strategy 1

```
import statsmodels.api as sm
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import os
import glob
from tqdm import tqdm
from sklearn.linear_model import LinearRegression


column_names = ["Open time", "Open", "High", "Low", "Close", "Volume",
                "Close time", "Quote asset volume", "Number of trades",
                "Taker buy base asset volume", "Taker buy quote asset volume", "Ignore"]

def aggregate_file(name, cutoff_date=None):
    directory_path = os.path.join(os.getcwd(), 'data', name)
    file_paths = glob.glob(os.path.join(directory_path, '*.csv'))

    #                column_names
    dataframes = [pd.read_csv(file_path, header=None, names=column_names) for file_path in

    aggregate = pd.concat(dataframes, ignore_index=True)
    aggregate['Open time'] = pd.to_datetime(aggregate['Open time'], unit='ms')

    #                     cutoff_date
    if cutoff_date is not None:
        cutoff_datetime = pd.to_datetime(cutoff_date)
        aggregate = aggregate[aggregate['Open time'] < cutoff_datetime]

    aggregate = aggregate.sort_values(by='Open time').reset_index(drop=True)
    return aggregate

def calculate_vwap(data):
    # Calculate the typical price for each row
    data['Typical_Price'] = (data['High'] + data['Low'] + data['Close']) / 3

    # Calculate the product of the typical price and the volume
    data['TP_Volume'] = data['Typical_Price'] * data['Volume']

    # Calculate the cumulative TP_Volume and cumulative Volume
    data['Cum_TP_Volume'] = data['TP_Volume'].cumsum()
    data['Cum_Volume'] = data['Volume'].cumsum()

    # Calculate the VWAP
    data['VWAP'] = data['Cum_TP_Volume'] / data['Cum_Volume']

    # Drop the intermediate columns used for calculation
    data.drop(['Typical_Price', 'TP_Volume', 'Cum_TP_Volume', 'Cum_Volume'], axis=1, inpla

    return data

# cut_off = '2021-06-01'
#
# BTC = aggregate_file('BTC',cut_off)
# LTC = aggregate_file('LTC',cut_off)
# BCH = aggregate_file('BCH',cut_off)
# ETH = aggregate_file('ETH',cut_off)

BTC = aggregate_file('BTC')
LTC = aggregate_file('LTC')
BCH = aggregate_file('BCH')
ETH = aggregate_file('ETH')
```

```
BTC = calculate_vwap(BTC)
ETH = calculate_vwap(ETH)
LTC = calculate_vwap(LTC)
BCH = calculate_vwap(BCH)

BTC['Open_diff_1'] = BTC['Open'].diff()
LTC['Open_diff_1'] = LTC['Open'].diff()
ETH['Open_diff_1'] = ETH['Open'].diff()
BCH['Open_diff_1'] = BCH['Open'].diff()

X = pd.concat([LTC['Open_diff_1'],ETH['Open_diff_1'],BCH['Open_diff_1']],axis=1).dropna()
X.columns = ['LTC', 'ETH', 'BCH']
Y = BTC['Open_diff_1'].dropna()
X = sm.add_constant(X)

model = sm.OLS(Y, X)
results = model.fit()
LTC_coef = results.params['LTC']
ETH_coef = results.params['ETH']
BCH_coef = results.params['BCH']

spread = pd.DataFrame()
spread['Spread'] = - ETH_coef*ETH['Open'] - BCH_coef*BCH['Open'] - LTC_coef*LTC['Open'] +

spread = pd.concat([BTC['Open time'],spread],axis=1).dropna()
spread.columns = ['Open time','Spread']
spread = spread.set_index('Open time')

mean_value = spread['Spread'].mean()
std_dev = spread['Spread'].std()


def calculate_cointegration(Y, X):
    #                       NaN
    if np.any(np.isnan(Y)) or np.any(np.isnan(X)):
        print("Warning: NaN values found in the data.")
        return False, {}
    if np.any(np.isinf(Y)) or np.any(np.isinf(X)):
        print("Warning: Inf values found in the data.")
        return False, {}

    #
    try:
        reg = LinearRegression().fit(X, Y)
        params = {'const': reg.intercept_, 'coeffs': reg.coef_}
        return True, params
    except Exception as e:
        print(f"Error during regression: {e}")
        return False, {}


#
def generate_trade_signals(crypto_dfs, min_window=300, step=60, multiply=1.0):
    all_data = pd.concat(
        [crypto_dfs['BTC']['Open'], crypto_dfs['ETH']['Open'], crypto_dfs['LTC']['Open'],
        axis=1)
    all_data.columns = ['BTC', 'ETH', 'LTC', 'BCH']
    all_data_diff = all_data.diff().dropna()  #

    #                   DataFrame
    signals_df = pd.DataFrame(index=all_data.index[min_window:])
    signals_df['signal'] = 0
    signals_df['Spread'] = np.nan
    signals_df['ETH_coef'] = np.nan
    signals_df['BCH_coef'] = np.nan
```

```
    signals_df['LTC_coef'] = np.nan

    for end in tqdm(range(min_window, len(all_data) + 1, step)):
        window_data = all_data_diff.iloc[:end]
        Y = window_data['BTC']
        X = window_data[['LTC', 'ETH', 'BCH']]

        has_coint, params = calculate_cointegration(Y, X)
        if has_coint:
            coeffs = np.array([params['coeffs'][i] for i in range(3)])
            # Calculate spreads using original data
            original_window_data = all_data.iloc[:end]
#
            spreads = original_window_data[['LTC', 'ETH', 'BCH']].dot(-coeffs) + original
            mean_spread = spreads.mean()
            std_spread = spreads.std()

            # Apply the coefficients to the next step observations using original data
            next_index_end = min(end + step, len(all_data))
#
            next_data = all_data[['LTC', 'ETH', 'BCH']].iloc[end:next_index_end].dot(-coe
            signals_df.loc[next_data.index, 'Spread'] = next_data
            signals_df.loc[next_data.index, ['ETH_coef', 'BCH_coef', 'LTC_coef']] = coeffs

            # Generate trading signals
            signals_df.loc[next_data.index, 'signal'] = (
                    (next_data < mean_spread - multiply * std_spread).astype(int) -
                    (next_data > mean_spread + multiply * std_spread).astype(int)
            )
    return signals_df.dropna()



def plot_backtest_results(trades_df, signals_df,min_window,step,multiplier):
    fig, (ax1, ax2, ax3, ax4) = plt.subplots(4, 1, sharex=True, figsize=(14, 16))
    #             Spread
    spread_filter = spread.loc[spread.index.intersection(signals_df.index)]
    ax1.plot(signals_df.index, signals_df['Spread'], label='Spread',color='blue')
    ax1.plot(spread_filter.index, spread_filter['Spread'], label='Full-Sample␣Spread', col
    ax1.axhline(y=mean_value, color='purple', linestyle='--', label='Full-Sample␣Mean')
    ax1.axhline(y=mean_value - std_dev * multiplier, color='g', linestyle='--', label='Low
    ax1.axhline(y=mean_value + std_dev * multiplier, color='g', linestyle='--')
    #             signal             == 1
    buy_signals = signals_df[signals_df['signal'] == 1]
    ax1.scatter(buy_signals.index, buy_signals['Spread'], color='red', label='Buy␣Signal'

    #             signal             == -1
    sell_signals = signals_df[signals_df['signal'] == -1]
    ax1.scatter(sell_signals.index, sell_signals['Spread'], color='green', label='Sell␣Sig
    ax1.set_title('Spread␣and␣Signals')
    ax1.legend()

    ax2.plot(trades_df.index, trades_df['value'], label='Net␣Pnl', color='green')
    ax2.plot(trades_df.index, trades_df['net_value'], label='Pnl', color='blue')
    ax2.set_title(f'Pnl␣with␣Parameters:␣min_window={min_window},␣step={step},␣multiplier=
    ax2.legend()

    trades_df['running_max'] = np.maximum.accumulate(trades_df['value'])
    trades_df['drawdown'] = trades_df['value'] - trades_df['running_max']
    diff = trades_df.index.to_series().diff()
    breaks = diff > pd.Timedelta('1min')
    segments = breaks.cumsum()

    #
    for segment in segments.unique():
```

```
        segment_df = trades_df[segments == segment]
        ax3.fill_between(segment_df.index, 0, segment_df['drawdown'], step='pre', alpha=0

    ax4.plot(trades_df.index, trades_df['transaction_cost'], label='transaction_cost', col
    #           X
    plt.xticks(rotation=45)
    plt.tight_layout()
    name = 'window method with multiplier' + str(multiplier) + 'step' + str(step) + '.jpeg
    plt.savefig(name, dpi=300)
    plt.show()

    return trades_df['drawdown'].min()



def backtest(crypto_dfs, spread_signals):
    spread_signals = spread_signals.dropna()
    crypto_dfs = {key: df.loc[df.index.intersection(spread_signals.index)] for key, df in

    transaction_costs = (abs(ETH_coef * crypto_dfs['ETH']['VWAP']) +
                         abs(BCH_coef * crypto_dfs['BCH']['VWAP']) +
                         abs(LTC_coef * crypto_dfs['LTC']['VWAP']) +
                         abs(crypto_dfs['BTC']['VWAP'])) * 0.0001

    trades_df = pd.DataFrame(index=spread_signals.index)
    trades_df['position'] = 0.0
    trades_df['cash'] = 10000.0
    trades_df['transaction_cost'] = 0.0  #
    trades_df['value'] = 10000.0
    trade_results = []
    current_position = 0
    cumulative_cost = 0  #

    for i in tqdm(range(1, len(spread_signals))):
        signal = spread_signals['signal'].iloc[i]
        market_value = spread_signals['Spread'].iloc[i]
        transaction_cost = transaction_costs.iloc[i]
        # units = 1
        if current_position == 0 and signal != 0:
            #
            units = max(int(trades_df['cash'].iloc[i - 1] / (abs(market_value + transactio
            current_position = signal
            position_value = market_value * units * signal
            trades_df.at[spread_signals.index[i], 'position'] = position_value
            open_position_value = position_value
            open_transaction_cost = transaction_cost * units
            cumulative_cost += open_transaction_cost
            trades_df.at[spread_signals.index[i], 'cash'] = trades_df.at[spread_signals.in
        elif current_position * signal == -1:
            #
            end_transaction_cost = transaction_cost * units
            cumulative_cost += end_transaction_cost
            trades_df.at[spread_signals.index[i], 'cash'] = trades_df.at[spread_signals.in
            end_position_value = market_value * current_position * units
            trade_result = end_position_value - open_position_value - open_transaction_cos
            trade_results.append(trade_result)
            trades_df.at[spread_signals.index[i], 'position'] = 0
            current_position = 0
        else:
            #
            if current_position != 0:
                trades_df.at[spread_signals.index[i], 'position'] = market_value * current
            trades_df.at[spread_signals.index[i], 'cash'] = trades_df.at[spread_signals.in
        #
        trades_df.at[spread_signals.index[i], 'transaction_cost'] = cumulative_cost
```

```
            trades_df.at[spread_signals.index[i], 'value'] = trades_df.at[spread_signals.index
                                                             trades_df.at[spread_signals.index
        trades_df['net_value'] = trades_df['value'] + trades_df['transaction_cost']
        daily_values = trades_df['value'].resample('D').last()
        daily_returns = daily_values.pct_change().dropna()

        #
        mean_daily_returns = daily_returns.mean()
        std_dev_daily_returns = daily_returns.std()

        #                2               %
        annual_risk_free_rate = 0.02
        daily_risk_free_rate = (1 + annual_risk_free_rate) ** (1 / 365) - 1

        #       Sharpe
        sharpe_ratio = (mean_daily_returns - daily_risk_free_rate) / std_dev_daily_returns * n

        #
        profits = [result for result in trade_results if result > 0]
        losses = [result for result in trade_results if result < 0]
        highest_value = max(trades_df['value'])
        lowest_value = min(trades_df['value'])
        final_net_value = trades_df['value'].iloc[-1]
        max_profit = max(profits) if profits else 0
        max_loss = min(losses) if losses else 0

        metrics = [
            len(trade_results),
            len(profits) / len(trade_results) * 100 if trade_results else 0,
            len(losses) / len(trade_results) * 100 if trade_results else 0,
            sum(profits) / len(profits) if profits else 0,
            sum(losses) / len(losses) if losses else 0,
            highest_value,
            lowest_value,
            final_net_value,
            max_profit,
            max_loss,
            sharpe_ratio
        ]

        return trades_df, metrics

    crypto_dfs = {
        'BTC': BTC.set_index('Open time'),
        'ETH': ETH.set_index('Open time'),
        'LTC': LTC.set_index('Open time'),
        'BCH': BCH.set_index('Open time')
    }


    #    window43200    ,step 1440
    with open('trades information test.txt', 'w') as f:
        header = "{:<12} {:<13} {:<12} {:<12} {:<12} {:<15} {:<15} {:<16} {:<11} {:<10
            "Multiplier", "Total Trades", "Win Rate (%)", "Loss Rate (%)", "Average Win", "Ave
            "Highest Value", "Lowest Value", "Final Net Value", "Max Profit", "Max Loss", "Sha
        )
        f.write(header + '\n')

        #                    c
        c = [0.5,0.7,1,1.3,1.5]
        for multiplier in c:
            #
            trade_signals = generate_trade_signals(crypto_dfs, min_window=43200, step=1440, mu

            #
```

```
backtest_results, metrics = backtest(crypto_dfs, trade_signals)

#
drawdown = plot_backtest_results(backtest_results, trade_signals, 43200, 1440, mul

#
data_string = "{:<12} {:<15} {:<15f} {:<15f} {:<15f} {:<15f} {:<16f} {:<16f} {:<17
    multiplier, metrics[0], metrics[1], metrics[2], metrics[3], metrics[4], metric
    metrics[7], metrics[8], metrics[9], metrics[10], -drawdown
)

#
f.write(data_string + '\n')
```

A Appendix: codes for strategy 2

```
# %%
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
from datetime import datetime
from itertools import product

import statsmodels.api as sm
import os
import glob
from tqdm import tqdm

# %%
column_names = ["Open time", "Open", "High", "Low", "Close", "Volume",
                "Close time", "Quote asset volume", "Number of trades",
                "Taker buy base asset volume", "Taker buy quote asset volume", "Ignore"]

def calculate_vwap(data):
    # Calculate the typical price for each row
    data['Typical_Price'] = (data['High'] + data['Low'] + data['Close']) / 3

    # Calculate the product of the typical price and the volume
    data['TP_Volume'] = data['Typical_Price'] * data['Volume']

    # Calculate the cumulative TP_Volume and cumulative Volume
    data['Cum_TP_Volume'] = data['TP_Volume'].cumsum()
    data['Cum_Volume'] = data['Volume'].cumsum()

    # Calculate the VWAP
    data['vwap'] = data['Cum_TP_Volume'] / data['Cum_Volume']

    # Drop the intermediate columns used for calculation
    data.drop(['Typical_Price', 'TP_Volume', 'Cum_TP_Volume', 'Cum_Volume'], axis=1, inpla

    return data

def data_preparation(name):
    file_paths = glob.glob(f'../data/futures/{name}USDT/1m/*.csv')
    dataframes = [pd.read_csv(file_path, header=None, names=column_names) for file_path in
    aggregate = pd.concat(dataframes)
    aggregate.drop(aggregate.index[0],inplace=True)
    aggregate[['Open time', 'Close time']] = aggregate[['Open time', 'Close time']].astype
    aggregate['Open time'] = aggregate['Open time'].apply(lambda x: datetime.utcfromtimest
    aggregate['Close time'] = aggregate['Close time'].apply(lambda x: datetime.utcfromtime
    aggregate.set_index('Open time',inplace=True)
    aggregate[["Open", "High", "Low", "Close", "Volume",
                "Quote asset volume", "Number of trades",
                "Taker buy base asset volume", "Taker buy quote asset volume", "Ignore"]]
                "Quote asset volume", "Number of trades",
                "Taker buy base asset volume", "Taker buy quote asset volume", "Ignore"]]

    df = calculate_vwap(aggregate)
    df.sort_index(inplace=True)

    return df

BTC = data_preparation('BTC')
LTC = data_preparation('LTC')
BCH = data_preparation('BCH')
ETH = data_preparation('ETH')

# %%
BTC
```

```
# %%
B = np.array([[1,-1,0,0],
              [1,0,-1,0],
              [1,0,0,-1],
              [0,1,-1,0],
              [0,1,0,-1],
              [0,0,1,-1]]
             )

BTC0 = BTC
LTC0 = LTC
BCH0 = BCH
ETH0 = ETH

# %%
# calculate preference function rho
def rho(row):
    return -(row['spread']-row['mean'])/row['std']

# calculate target weight based on the utility for each asset
def calc_weight(row):
    long, short = {}, {}
    if row['BTC'] > 0:
        long['BTC'] = row['BTC']
    elif row['BTC'] < 0:
        short['BTC'] = row['BTC']

    if row['LTC'] > 0:
        long['LTC'] = row['LTC']
    elif row['LTC'] < 0:
        short['LTC'] = row['LTC']

    if row['BCH'] > 0:
        long['BCH'] = row['BCH']
    elif row['BCH'] < 0:
        short['BCH'] = row['BCH']

    if row['ETH'] > 0:
        long['ETH'] = row['ETH']
    elif row['ETH'] < 0:
        short['ETH'] = row['ETH']

    weights = {}
    long_sum = np.sum(list(long.values()))
    short_sum = -np.sum(list(short.values()))
    for i in long.keys():
        weights[i] = long[i]/long_sum
    for i in short.keys():
        weights[i] = short[i]/short_sum

    order = ["BTC","LTC", "BCH", "ETH"]
    weights = {k: weights[k] for k in order if k in weights}

    return weights

def recalc_rho(arr):
    return 1/4*B@B.T@arr
# u = 1/4*B.T*rho
# rho2 = B*u

# %%
def aggregate_kline(df, interval):
    new_df = pd.DataFrame()
    new_df['Open'] = df['Open'].resample(interval).first()
```

```
    new_df['High'] = df['High'].resample(interval).max()
    new_df['Low'] = df['Low'].resample(interval).min()
    new_df['Close'] = df['Close'].resample(interval).last()

    return new_df

# %%
windows = [120,180,240,360]
intervals = ['1h','2h','3h']
BTC = BTC0
LTC = LTC0
BCH = BCH0
ETH = ETH0

def backtest_v1(n):
    BTC_LTC = np.log(BTC['Open']/LTC['Open'])
    BTC_LTC.name = 'spread'
    BTC_LTC = pd.DataFrame(BTC_LTC)
    BTC_LTC['mean'] = BTC_LTC['spread'].rolling(n).mean()
    BTC_LTC['std'] = BTC_LTC['spread'].rolling(n).std()

    BTC_BCH = np.log(BTC['Open']/BCH['Open'])
    BTC_BCH.name = 'spread'
    BTC_BCH = pd.DataFrame(BTC_BCH)
    BTC_BCH['mean'] = BTC_BCH['spread'].rolling(n).mean()
    BTC_BCH['std'] = BTC_BCH['spread'].rolling(n).std()

    BTC_ETH = np.log(BTC['Open']/ETH['Open'])
    BTC_ETH.name = 'spread'
    BTC_ETH = pd.DataFrame(BTC_ETH)
    BTC_ETH['mean'] = BTC_ETH['spread'].rolling(n).mean()
    BTC_ETH['std'] = BTC_ETH['spread'].rolling(n).std()

    LTC_BCH = np.log(LTC['Open']/BCH['Open'])
    LTC_BCH.name = 'spread'
    LTC_BCH = pd.DataFrame(LTC_BCH)
    LTC_BCH['mean'] = LTC_BCH['spread'].rolling(n).mean()
    LTC_BCH['std'] = LTC_BCH['spread'].rolling(n).std()

    LTC_ETH = np.log(LTC['Open']/ETH['Open'])
    LTC_ETH.name = 'spread'
    LTC_ETH = pd.DataFrame(LTC_ETH)
    LTC_ETH['mean'] = LTC_ETH['spread'].rolling(n).mean()
    LTC_ETH['std'] = LTC_ETH['spread'].rolling(n).std()

    BCH_ETH = np.log(BCH['Open']/ETH['Open'])
    BCH_ETH.name = 'spread'
    BCH_ETH = pd.DataFrame(BCH_ETH)
    BCH_ETH['mean'] = BCH_ETH['spread'].rolling(n).mean()
    BCH_ETH['std'] = BCH_ETH['spread'].rolling(n).std()

    # establish initial preference function
    rho_df = pd.DataFrame()
    rho_df['BTC_LTC'] = BTC_LTC.apply(lambda row: rho(row), axis=1)
    rho_df['BTC_BCH'] = BTC_BCH.apply(lambda row: rho(row), axis=1)
    rho_df['BTC_ETH'] = BTC_ETH.apply(lambda row: rho(row), axis=1)
    rho_df['LTC_BCH'] = LTC_BCH.apply(lambda row: rho(row), axis=1)
    rho_df['LTC_ETH'] = LTC_ETH.apply(lambda row: rho(row), axis=1)
    rho_df['BCH_ETH'] = BCH_ETH.apply(lambda row: rho(row), axis=1)
    rho_df.dropna(inplace=True)

    # estimate utility weight for each asset
    utility = pd.DataFrame(1/4*(B.T@rho_df.values.T).T,columns=["BTC","LTC", "BCH", "ETH"]

    # estimate target weight
```

```
        weight = pd.DataFrame(utility.apply(lambda row: calc_weight(row), axis=1).to_list())
        weight.index = utility.index

        # get trading price
        price = pd.DataFrame()
        price['BTC'] = BTC['Close']
        price['LTC'] = LTC['Close']
        price['BCH'] = BCH['Close']
        price['ETH'] = ETH['Close']

        # calculate target position for each asset
        total = 100000
        target_pos = total*weight/(price.dropna()).shift()
        target_pos.dropna(inplace=True)

        # calculate pnl
        delta_pos = target_pos.diff()
        delta_pos.iloc[0] = target_pos.iloc[0]
        portfolio_value = (target_pos*price).sum(axis=1)
        transaction_value = -(delta_pos*price).sum(axis=1)
        pnl = portfolio_value+transaction_value.cumsum()
        cost = (delta_pos.abs()*price*0.0001).sum(axis=1)
        net_pnl = pnl-cost.cumsum()

        summary_df = pd.DataFrame()
        summary_df['cost'] = cost.cumsum()
        summary_df['pnl'] = pnl
        summary_df['net_pnl'] = net_pnl

        return summary_df

    def backtest_v2(n, interval):
        BTC_LTC = np.log(BTC['Open']/LTC['Open'])
        BTC_LTC.name = 'spread'
        BTC_LTC = pd.DataFrame(BTC_LTC)
        BTC_LTC['mean'] = BTC_LTC['spread'].rolling(n).mean()
        BTC_LTC['std'] = BTC_LTC['spread'].rolling(n).std()

        BTC_BCH = np.log(BTC['Open']/BCH['Open'])
        BTC_BCH.name = 'spread'
        BTC_BCH = pd.DataFrame(BTC_BCH)
        BTC_BCH['mean'] = BTC_BCH['spread'].rolling(n).mean()
        BTC_BCH['std'] = BTC_BCH['spread'].rolling(n).std()

        BTC_ETH = np.log(BTC['Open']/ETH['Open'])
        BTC_ETH.name = 'spread'
        BTC_ETH = pd.DataFrame(BTC_ETH)
        BTC_ETH['mean'] = BTC_ETH['spread'].rolling(n).mean()
        BTC_ETH['std'] = BTC_ETH['spread'].rolling(n).std()

        LTC_BCH = np.log(LTC['Open']/BCH['Open'])
        LTC_BCH.name = 'spread'
        LTC_BCH = pd.DataFrame(LTC_BCH)
        LTC_BCH['mean'] = LTC_BCH['spread'].rolling(n).mean()
        LTC_BCH['std'] = LTC_BCH['spread'].rolling(n).std()

        LTC_ETH = np.log(LTC['Open']/ETH['Open'])
        LTC_ETH.name = 'spread'
        LTC_ETH = pd.DataFrame(LTC_ETH)
        LTC_ETH['mean'] = LTC_ETH['spread'].rolling(n).mean()
        LTC_ETH['std'] = LTC_ETH['spread'].rolling(n).std()

        BCH_ETH = np.log(BCH['Open']/ETH['Open'])
        BCH_ETH.name = 'spread'
        BCH_ETH = pd.DataFrame(BCH_ETH)
```

```
    BCH_ETH['mean'] = BCH_ETH['spread'].rolling(n).mean()
    BCH_ETH['std'] = BCH_ETH['spread'].rolling(n).std()

    # establish initial preference function
    rho_df = pd.DataFrame()
    rho_df['BTC_LTC'] = BTC_LTC.apply(lambda row: rho(row), axis=1)
    rho_df['BTC_BCH'] = BTC_BCH.apply(lambda row: rho(row), axis=1)
    rho_df['BTC_ETH'] = BTC_ETH.apply(lambda row: rho(row), axis=1)
    rho_df['LTC_BCH'] = LTC_BCH.apply(lambda row: rho(row), axis=1)
    rho_df['LTC_ETH'] = LTC_ETH.apply(lambda row: rho(row), axis=1)
    rho_df['BCH_ETH'] = BCH_ETH.apply(lambda row: rho(row), axis=1)
    rho_df.dropna(inplace=True)

    # estimate utility weight for each asset
    utility = pd.DataFrame(1/4*(B.T@rho_df.values.T).T,columns=["BTC","LTC", "BCH", "ETH"]

    # estimate target weight
    weight = pd.DataFrame(utility.apply(lambda row: calc_weight(row), axis=1).to_list())
    weight.index = utility.index

    # get trading price
    price = pd.DataFrame()
    price['BTC'] = BTC['Close']
    price['LTC'] = LTC['Close']
    price['BCH'] = BCH['Close']
    price['ETH'] = ETH['Close']
    price = price.resample(interval).last()

    # calculate target position for each asset
    total = 100000
    target_pos = total*weight/(price.dropna()).shift()
    target_pos.dropna(inplace=True)
    target_pos = target_pos.resample(interval).mean().round()

    # calculate pnl
    delta_pos = target_pos.diff()
    delta_pos.iloc[0] = target_pos.iloc[0]
    portfolio_value = (target_pos*price).sum(axis=1)
    transaction_value = -(delta_pos*price).sum(axis=1)
    pnl = portfolio_value+transaction_value.cumsum()
    cost = (delta_pos.abs()*price*0.0001).sum(axis=1)
    net_pnl = pnl-cost.cumsum()

    summary_df = pd.DataFrame()
    summary_df['cost'] = cost.cumsum()
    summary_df['pnl'] = pnl
    summary_df['net_pnl'] = net_pnl

    return summary_df

def backtest_v3(n, interval):
    BTC_LTC = np.log(BTC['Open']/LTC['Open'])
    BTC_LTC.name = 'spread'
    BTC_LTC = pd.DataFrame(BTC_LTC)
    BTC_LTC['mean'] = BTC_LTC['spread'].rolling(n).mean()
    BTC_LTC['std'] = BTC_LTC['spread'].rolling(n).std()

    BTC_BCH = np.log(BTC['Open']/BCH['Open'])
    BTC_BCH.name = 'spread'
    BTC_BCH = pd.DataFrame(BTC_BCH)
    BTC_BCH['mean'] = BTC_BCH['spread'].rolling(n).mean()
    BTC_BCH['std'] = BTC_BCH['spread'].rolling(n).std()

    BTC_ETH = np.log(BTC['Open']/ETH['Open'])
    BTC_ETH.name = 'spread'
```

```
BTC_ETH = pd.DataFrame(BTC_ETH)
BTC_ETH['mean'] = BTC_ETH['spread'].rolling(n).mean()
BTC_ETH['std'] = BTC_ETH['spread'].rolling(n).std()

LTC_BCH = np.log(LTC['Open']/BCH['Open'])
LTC_BCH.name = 'spread'
LTC_BCH = pd.DataFrame(LTC_BCH)
LTC_BCH['mean'] = LTC_BCH['spread'].rolling(n).mean()
LTC_BCH['std'] = LTC_BCH['spread'].rolling(n).std()

LTC_ETH = np.log(LTC['Open']/ETH['Open'])
LTC_ETH.name = 'spread'
LTC_ETH = pd.DataFrame(LTC_ETH)
LTC_ETH['mean'] = LTC_ETH['spread'].rolling(n).mean()
LTC_ETH['std'] = LTC_ETH['spread'].rolling(n).std()

BCH_ETH = np.log(BCH['Open']/ETH['Open'])
BCH_ETH.name = 'spread'
BCH_ETH = pd.DataFrame(BCH_ETH)
BCH_ETH['mean'] = BCH_ETH['spread'].rolling(n).mean()
BCH_ETH['std'] = BCH_ETH['spread'].rolling(n).std()

# establish initial preference function
rho_df = pd.DataFrame()
rho_df['BTC_LTC'] = BTC_LTC.apply(lambda row: rho(row), axis=1)
rho_df['BTC_BCH'] = BTC_BCH.apply(lambda row: rho(row), axis=1)
rho_df['BTC_ETH'] = BTC_ETH.apply(lambda row: rho(row), axis=1)
rho_df['LTC_BCH'] = LTC_BCH.apply(lambda row: rho(row), axis=1)
rho_df['LTC_ETH'] = LTC_ETH.apply(lambda row: rho(row), axis=1)
rho_df['BCH_ETH'] = BCH_ETH.apply(lambda row: rho(row), axis=1)
rho_df.dropna(inplace=True)
rho_df = rho_df.where(rho_df.abs()>=1,0)

# estimate utility weight for each asset
utility = pd.DataFrame(1/4*(B.T@rho_df.values.T).T,columns=["BTC","LTC", "BCH", "ETH"]

# estimate target weight
weight = pd.DataFrame(utility.apply(lambda row: calc_weight(row), axis=1).to_list())
weight.index = utility.index

# get trading price
price = pd.DataFrame()
price['BTC'] = BTC['Close']
price['LTC'] = LTC['Close']
price['BCH'] = BCH['Close']
price['ETH'] = ETH['Close']
price = price.resample(interval).last()

# calculate target position for each asset
total = 100000
target_pos = total*weight/(price.dropna()).shift()
target_pos.dropna(inplace=True)
target_pos = target_pos.resample(interval).mean().round()

# calculate pnl
delta_pos = target_pos.diff()
delta_pos.iloc[0] = target_pos.iloc[0]
portfolio_value = (target_pos*price).sum(axis=1)
transaction_value = -(delta_pos*price).sum(axis=1)
pnl = portfolio_value+transaction_value.cumsum()
cost = (delta_pos.abs()*price*0.0001).sum(axis=1)
net_pnl = pnl-cost.cumsum()

summary_df = pd.DataFrame()
summary_df['cost'] = cost.cumsum()
```

```
    summary_df['pnl'] = pnl
    summary_df['net_pnl'] = net_pnl

    return summary_df

df1 = backtest_v1(n=60)
df2 = backtest_v2(n=360, interval="3h")
df3 = backtest_v3(n=120, interval="2h")

# %%
def plot_backtest_results(df, version):
    pnl_daily = df.resample('d').last()
    high_level = pnl_daily['net_pnl'].cummax()
    drawdown = pnl_daily['net_pnl']-high_level
    fig, (ax1, ax2) = plt.subplots(2, 1, sharex=True, figsize=(7, 6))

    # plot pnl
    ax1.plot(pnl_daily.index, pnl_daily['pnl'], label='pnl')
    ax1.plot(pnl_daily.index, pnl_daily['net_pnl'], label='net pnl')
    ax1.plot(pnl_daily.index, pnl_daily['cost'], label='cost')
    ax1.set_title('pnl')
    ax1.legend()

    # plot drawdown
    ax2.fill_between(drawdown.index, 0, drawdown, where=drawdown <= 0, step='pre', label=
    ax2.set_title('drawdown')

    plt.xticks(rotation=45)
    plt.tight_layout()
    plt.savefig(f'./backtest_result/{version}.png')
    plt.show()


# %%
plot_backtest_results(df1, "v1")
plot_backtest_results(df2, "v2")
plot_backtest_results(df3, "v3")

# %%
def calculate_statistics(df):
    delta_cost = df['cost'].diff()
    trade_counts = len(delta_cost[delta_cost != 0])

    df['balance'] = df['net_pnl'] + 100000
    total_return = ((df['balance'].iloc[-1]-100000)/100000)*100
    annual_return = total_return/3
    daily_balance = df["balance"].resample('d').last()

    high_level = daily_balance.cummax()
    drawdown = daily_balance-high_level
    ddpercent = drawdown/high_level*100
    max_drawdown = -ddpercent.min()
    if df['balance'].min() < 0:
        sharpe = None
    else:

        # pre_day_balance = daily_balance.shift(1)
        daily_return = daily_balance.pct_change()
        daily_return_mean = daily_return.mean()*100
        daily_return_std = daily_return.std()*100
        sharpe = (daily_return_mean-0.02/365)/daily_return_std*np.sqrt(365)

    return {"total trades": trade_counts,
            "annual return": annual_return,
            "sharpe": sharpe,
```

```
            "max drawdown": max_drawdown}

print(calculate_statistics(df1))
print(calculate_statistics(df2))
print(calculate_statistics(df3))
```