Pairs Trading - Cointegration

Motivation:

- Security A - (Something Correlated to it) is more mean-reverting. Similar securties should trade similarily. Hence, any discrepancies between the correlated pairs should converge
- When there is a discrepancy, we form a long-short spread trade and bet on the discrepancy converging
- Pairs trading is one of the most popular statistical arbitrage strategies in traditional markets. We want to test the performance of this strategy in the cryptocurrency market, as it is still relatively new and should be fertile grounds for finding market inefficiencies
- We will use cointegration to find similar pairs as it is a robust statistical approach for identifying long-term equilibrium relationships between assets

In [1]:
```
from datetime import datetime
import pandas as pd
import numpy as np
import seaborn as sns
import matplotlib.pyplot as plt
from matplotlib.pyplot import figure

import statsmodels.api as sm
from statsmodels.tsa.stattools import adfuller

import warnings
warnings.filterwarnings("ignore", category=FutureWarning)

from joblib import Parallel, delayed
from itertools import combinations
```

Data Cleaning

We selected the top 1,000 currencies sorted by volume from CoinGecko. Then, we filtered out coins with fewer than 90% of daily price data points between 2018 and 2024.

In [2]:
```
from_pickle = pd.read_pickle('Data/CoinGecko_px_vol_1D.pkl')
```

In [3]:
```
price_columns = [col for col in from_pickle.columns if 'price' in col]
crypto_px = from_pickle[price_columns]
crypto_px.columns = crypto_px.columns.droplevel(1)
```

In [4]:
```
crypto_px = crypto_px.loc['2018-01-01':]
crypto_px = crypto_px.drop(columns=['TUSD', 'DAI', 'WBTC', 'WETH', 'USDC', 'BSV'])
```

In [5]:
```python
# Calculate the total number of data points
total_data_points = len(crypto_px)

# Calculate the number of non-NA/null entries for each coin
non_null_counts = crypto_px.notnull().sum()

# Calculate the threshold for 90% of the data points
threshold = 0.90 * total_data_points

# Filter out columns with fewer than 90% of data points
crypto_px = crypto_px.loc[:, non_null_counts >= threshold]

crypto_px
```

Out[5]:

Date	BTC	XRP	BNB	DOGE	ADA	TRX	LTC	LINK	BCH	EOS
2018-01-01	14093.606831	2.310120	8.828099	0.009091	0.747140	0.051654	230.462120	0.751033	2426.970077	7.672278
2018-01-02	15321.932852	2.455290	9.090393	0.009335	0.807430	0.080893	255.048185	0.689388	2627.026940	9.504036
2018-01-03	15583.885538	3.125710	9.886323	0.009592	1.075401	0.098107	248.042194	0.704623	2630.511811	10.090184
2018-01-04	15976.365194	3.220050	9.675758	0.010098	1.179347	0.218139	244.834372	1.036826	2458.894372	11.713284
2018-01-05	18336.922980	2.931380	16.488523	0.013841	1.077821	0.231673	254.138525	0.996575	2551.321685	9.673192
...	...	...	...	...	...	...	...	...	...	...
2024-08-12	58804.234500	0.552884	503.472306	0.100570	0.328140	0.127716	59.691360	10.012537	330.440521	0.467477
2024-08-13	59350.074333	0.568612	518.752217	0.107647	0.338874	0.126661	61.414267	10.561648	354.155288	0.500104
2024-08-14	60601.223178	0.576440	523.553455	0.106469	0.340317	0.128873	63.411783	10.577495	352.234629	0.507208
2024-08-15	58739.193822	0.568488	523.842262	0.102529	0.335251	0.130504	63.906408	10.396984	338.005714	0.506509
2024-08-16	57624.116929	0.561137	519.989791	0.100275	0.325393	0.130136	65.236601	10.171003	334.436579	0.490075

2568 rows × 86 columns

We have 86 coins and will select pairs from among them.

In [6]:
```python
coins_ret = crypto_px / crypto_px.shift() - 1
```

In [7]:
```python
def compute_turnover(port):
    to = (port.fillna(0)-port.shift().fillna(0)).abs().sum(1)
    return to
```

In [8]:
```python
def compute_sharpe_ratio(rets):
    mean_rets = rets.mean()*252
    vol = rets.std()*np.sqrt(252)
    sharpe_ratio = mean_rets / vol
    return sharpe_ratio
```

In [9]:
```
def compute_stats(rets):
    stats={}
    stats['avg'] = rets.mean()*252
    stats['vol'] = rets.std()*np.sqrt(252)
    stats['sharpe'] = stats['avg']/stats['vol']
    stats['hit_rate'] = rets[rets>0].count() / rets.count()
    stats = pd.DataFrame(stats)
    return stats
```

In [10]:
```
def drawdown(px):
    return (px / (px.expanding(min_periods=1).max()) - 1)
```

In [11]:
```
def duration(px):
    peak = px.expanding(min_periods=1).max()
    res = pd.DataFrame(index=px.index,columns=px.columns)
    for col in px.columns:
        for dt in px.index:
            if px.loc[dt,col] >= peak.loc[dt,col]:
                 res.loc[dt,col] = 0
            else:
                res.loc[dt,col] = res.loc[:dt,col].iloc[-2] + 1
    return res
```

Pairs Selection

Pairs are selected and updated every half year, making the process more practical for real-time implementation. Here's how it works:

- **Rolling OLS Regression**: Every six months, perform Ordinary Least Squares (OLS) regression on the log prices of two securities over the past year to obtain the residuals:

$$\log(p_{i,t}) = \alpha + \beta \log(p_{j,t}) + \epsilon_t$$

- **Stationarity Test**: Apply the Augmented Dickey-Fuller (ADF) test to the residuals to check for stationarity. A p-value < 0.05 indicates that the residuals are stationary.
- **Selection Criteria**: For each coin, select the pair with the most negative test statistic from the ADF test, ensuring all selected pairs have p-values less than 0.05.

By updating pairs every six months using one year of price data, this approach adapts to changing market conditions and enhances the strategy's practicality, especially in the fast-moving cryptocurrency market.

In [12]:
```
def adf_for_pair(symbol_i, symbol_j, crypto_px):
    """
    Perform the ADF test on the residuals of the OLS regression between two assets.

    Parameters:
    symbol_i (str): The first asset's symbol.
    symbol_j (str): The second asset's symbol.
    in_sample_px (DataFrame): The DataFrame containing in-sample price data for the asse

    Returns:
    tuple: A tuple containing the pair (symbol_i, symbol_j) and a tuple of (p_value, tes
    """
    # Convert raw price to log price
    crypto_log_px = np.log(crypto_px)

    # Handle missing data
    X = crypto_log_px[symbol_i].fillna(0).values
    Y = crypto_log_px[symbol_j].fillna(0).values

    # OLS regression
    model = sm.OLS(Y, sm.add_constant(X)).fit()
    alpha = model.params[0]
    beta = model.params[1]
    residuals = Y - beta * X - alpha

    # ADF test on residuals
    adf_result = adfuller(residuals)
    p_value = adf_result[1]
    test_statistic = adf_result[0]

    return (symbol_i, symbol_j), (p_value, test_statistic)
```

In [13]:
```python
def select_pairs(crypto_px, significance_level=0.05, top_n=1, n_jobs=-1):
    """
    Compute ADF test results for all pairs, select the top `n` cointegrated pairs for ea
    and display the results along with the update date.

    Parameters:
    crypto_px (DataFrame): DataFrame containing price data of assets.
    significance_level (float): The significance level threshold for p-values to select 
    top_n (int): The number of top pairs to select for each coin.
    n_jobs (int): Number of parallel jobs to run.

    Returns:
    list: A sorted list of unique cointegrated pairs.
    """

    # Get all combinations of pairs
    symbols = crypto_px.columns.tolist()
    pairs = list(combinations(symbols, 2))

    # Parallel computation of ADF tests for all pairs
    adf_results = Parallel(n_jobs=n_jobs)(
        delayed(adf_for_pair)(symbol_i, symbol_j, crypto_px) for symbol_i, symbol_j in p
    )

    # Create a DataFrame with ADF results
    pairs, results = zip(*adf_results)
    adf_df = pd.DataFrame(results, columns=['p_value', 'test_statistic'], index=pairs)

    # Filter pairs with p-values less than the significance level
    filtered_df = adf_df[adf_df['p_value'] < significance_level]

    # Initialize a set to store unique pairs
    final_pairs_set = set()

    # Initialize a dictionary to store top pairs for each coin
    top_pairs_per_coin = {}

    # Iterate over all unique coins in the filtered pairs
    all_coins = set(sum([list(pair) for pair in filtered_df.index], []))
    for coin in all_coins:
        # Filter pairs where the coin is involved
        coin_pairs = filtered_df.loc[[(coin in pair) for pair in filtered_df.index]]

        # Sort the pairs by the ADF test statistic (smaller test statistic is better)
        sorted_pairs = coin_pairs.sort_values(by='test_statistic')

        # Select the top `n` pairs for this coin
        top_pairs = sorted_pairs.head(top_n)

        # Store the result in the dictionary
        top_pairs_per_coin[coin] = top_pairs

        # Add the selected pairs to the final set
        final_pairs_set.update(top_pairs.index.tolist())

    # Convert the set to a sorted list
    final_pairs = sorted(final_pairs_set)

    # Determine the update date as the end date of crypto_px + 1 day
    update_date = crypto_px.index[-1] + pd.Timedelta(days=1)
    print(f"Pairs Updated date: {update_date.strftime('%Y-%m-%d')}")

    # Display the top cointegrated pairs for each coin
    sorted_coins = sorted(top_pairs_per_coin.keys())
    for coin in sorted_coins:
        df = top_pairs_per_coin[coin]
        top_coins = [pair[1] if pair[0] == coin else pair[0] for pair in df.index]

    # Print the final pairs and the count
    print(f"Final pairs to be traded: {final_pairs}")
    print(f"Number of pairs to be traded: {len(final_pairs)}")
```

```
    return final_pairs
```

Trading Strategy

1. Signal Generation

- **Residual Calculation:**

$$\epsilon_t = \log(p_{i,t}) - (\beta_t \log(p_{j,t}) + \alpha_t)$$

  where:
  - $\log(p_{i,t})$ and $\log(p_{j,t})$ are the log prices of coins i and j at time t.
  - β_t is calculated as:

$$\beta_t = \mathrm{Corr}_t \times \frac{\mathrm{Vol}_{j,t}}{\mathrm{Vol}_{i,t}}$$

  with:
  - Corr_t is the 90-day rolling correlation between $\log(p_{i,t})$ and $\log(p_{j,t})$.
  - $\mathrm{Vol}_{i,t}$ and $\mathrm{Vol}_{j,t}$ are the 90-day rolling volatilities of $\log(p_{i,t})$ and $\log(p_{j,t})$, respectively.
  - α_t is calculated as:

$$\alpha_t = \mu_{\log(p_{j,t})} - \beta_t \cdot \mu_{\log(p_{i,t})}$$

  where:
  - $\mu_{\log(p_{j,t},90)}$ is the 90-day rolling mean of $\log(p_{j,t})$.
  - $\mu_{\log(p_{i,t},90)}$ is the 90-day rolling mean of $\log(p_{i,t})$.
- **Z-Score Calculation:**

$$z_t = \frac{\epsilon_t - \mu_t}{\sigma_t}$$

  where:
  - μ_t is the 90-day rolling mean of the spread.
  - σ_t is the 90-day rolling standard deviation of the spread.

2. Portfolio Construction

- **Entry Signals:**
  - **Short** coin i and **long** β_t units of coin j if $z_t > 1$.
  - **Long** coin i and **short** β_t units of coin j if $z_t < -1$.
- **Exit Signals:**
  - Close the position when z_t moves to any of the following thresholds:

$$z_t \geq -\text{threshold} \quad \text{or} \quad z_t \leq \text{threshold}$$

    where threshold is one of the values: 0.1, 0.2, 0.5, or 0.7.

In [14]:
```
def gen_signals(px, pairs, window=90):
    signal_df = {}
    for pair in pairs:
        asset_i, asset_j = pair

        # Forward-fill missing values and replace zeros with NaNs
        px_i = px[asset_i].replace(0, np.nan).ffill()
        px_j = px[asset_j].replace(0, np.nan).ffill()

        # Apply log transformation
        log_px_i = np.log(px_i)
        log_px_j = np.log(px_j)

        # Calculate rolling covariance and variance
        rolling_cov = log_px_i.rolling(window=window, min_periods=1).cov(log_px_j)
        rolling_var = log_px_i.rolling(window=window, min_periods=1).var()

        # Calculate beta and alpha
        beta = rolling_cov / rolling_var
        alpha = log_px_j.rolling(window=window).mean() - beta * log_px_i.rolling(window=

        # Calculate spread for time t using beta and alpha
        spread = log_px_i - (beta * log_px_j + alpha)

        # Calculate rolling mean and standard deviation of the spread using data up to t
        spread_mean = spread.rolling(window=window, min_periods=1).mean()
        spread_std = spread.rolling(window=window, min_periods=1).std()

        # Calculate the z-score for time t using spread(t), mean(t), and std(t)
        z_score = (spread - spread_mean) / spread_std

        # Store beta, alpha, spread, and z-score in a multi-level column DataFrame
        signal_df[(pair, 'beta')] = beta
        signal_df[(pair, 'alpha')] = alpha
        signal_df[(pair, 'spread')] = spread
        signal_df[(pair, 'z_score')] = z_score

    # Convert the dictionary to a DataFrame
    signal_df = pd.DataFrame(signal_df)

    return signal_df
```

In [15]:
```
def gen_port(signal_df, pairs, crypto_px, threshold=0.5):
    # Initialize a DataFrame with the same index and columns as crypto_px, filled with N
    pos = pd.DataFrame(index=signal_df.index, columns=crypto_px.columns)
    for pair in pairs:
        asset_i, asset_j = pair
        # Access z-scores and betas for this pair
        z_scores = signal_df[(pair, 'z_score')]
        betas = signal_df[(pair, 'beta')]

        # Set positions based on z-scores
        pos.loc[z_scores > 1, asset_i] = -1  # Short one unit of asset_i
        pos.loc[z_scores < -1, asset_i] = 1  # Long one unit of asset_i
        pos.loc[(z_scores.abs() <= threshold), asset_i] = 0  # Exit signal

        pos.loc[z_scores > 1, asset_j] = betas  # Long beta units of asset_j
        pos.loc[z_scores < -1, asset_j] = -betas  # Short beta units of asset_j
        pos.loc[(z_scores.abs() <= threshold), asset_j] = 0  # Exit signal

    # Forward-fill missing values
    pos = pos.ffill()
    # Normalize to ensure a fully-invested portfolio
    pos = pos.divide(pos.abs().sum(axis=1), axis=0).fillna(0)
    return pos
```

Performance Evaluation

We evaluate the strategy's performance across four exit thresholds: 0.1, 0.2, 0.5, and 0.7. For each threshold, the following key metrics are calculated:

- **Sharpe Ratio**: Risk-adjusted return of the strategy.

- **Transaction Costs**: Average costs incurred due to trading.
- **Holding Period**: Average number of days a position is held.
- **Turnover**: Average daily proportion of the portfolio that is traded.
- **Annualized Return**: Average yearly return of the strategy.
- **Annualized Volatility**: Standard deviation of returns on an annual basis, indicating risk.

Cryptocurrencies can have commissions of ~7bps. While total slippage is unknown and will depend on the trader's volume as well, let's assume another 13 bps. So total all-in execution costs will be 20 bps for market-orders.

In [16]:
```
# Define the thresholds and initialize the arrays for storing metrics
end_of_insample = pd.Timestamp('2018-12-31')
last_available_date = crypto_px.index[-1]
thresholds = [0.1, 0.2, 0.5, 0.7]
metrics = {
    'Sharpe Ratio': np.zeros(len(thresholds)),
    'Return': np.zeros(len(thresholds)),
    'Volatility': np.zeros(len(thresholds)),
    'Holding Period': np.zeros(len(thresholds)),
    'Turnover': np.zeros(len(thresholds)),
    'Transaction Costs': np.zeros(len(thresholds)),
}

# Set the start of the out-of-sample period
start_of_out_sample = end_of_insample + pd.DateOffset(days=1)

# Define half-year periods for updating pairs
update_dates = pd.date_range(start=start_of_out_sample, end=crypto_px.index[-1], freq='6

# Initialize an empty DataFrame to store the complete portfolio over all periods
full_portfolio = pd.DataFrame(index=crypto_px.loc[start_of_out_sample:].index, columns=c

# Loop over each threshold to calculate metrics
for i, threshold in enumerate(thresholds):
    print(f"Evaluating with exit threshold: {threshold}")
    for start_date in update_dates:
        # Define the end of the update period (6 months later)
        end_date = start_date + pd.DateOffset(months=6) - pd.DateOffset(days=1)

        # Adjust end_date if it exceeds the last available date
        if end_date > last_available_date:
            end_date = last_available_date

        # Select the in-sample period for pair selection
        insample_start = start_date - pd.DateOffset(years=1)
        insample_end = start_date - pd.DateOffset(days=1)

        # Re-select pairs based on the ADF test or any other criteria
        updated_pairs = select_pairs(crypto_px.loc[insample_start:insample_end])

        # Generate signals for the new pairs
        signal_df = gen_signals(crypto_px, updated_pairs, window=90)
        signal_df = signal_df.loc[start_date:end_date]

        # Generate portfolio for the selected pairs
        port = gen_port(signal_df, updated_pairs, crypto_px, threshold)

        # Store the generated portfolio in the full_portfolio DataFrame
        full_portfolio.loc[start_date:end_date, :] = port

    # Calculate out-of-sample daily returns for the entire period
    out_sample_ret = coins_ret.loc[start_of_out_sample:][full_portfolio.columns]
    strat_gross_ret = (full_portfolio.shift() * out_sample_ret).sum(axis=1)

    # Calculate net returns with transaction costs
    to = compute_turnover(full_portfolio)
    tcost_bps = 20  # (commissions + slippage)
    strat_net_ret = strat_gross_ret.subtract(to * tcost_bps * 1e-4, fill_value=0)

    # Compute Sharpe ratio
    sharpe_ratio = compute_sharpe_ratio(strat_net_ret)
    metrics['Sharpe Ratio'][i] = sharpe_ratio

    # Compute transaction costs
    total_tcost = to * tcost_bps * 1e-4
    metrics['Transaction Costs'][i] = total_tcost.mean()

    # Compute holding period (average number of days a position is held)
    metrics['Holding Period'][i] = 2/to.mean()

    # Store turnover
    metrics['Turnover'][i] = to.mean()

    # Compute return
```

```python
    returns = strat_net_ret.mean()
    metrics['Return'][i] = returns

    # Compute volatility
    volatility = strat_net_ret.std()
    metrics['Volatility'][i] = volatility

# Convert metrics to a DataFrame for easier visualization
metrics_df = pd.DataFrame(metrics, index=thresholds)
```

```
'MTL'), ('UQC', 'STRAX'), ('VGX', 'GLM'), ('VIC', 'XVG'), ('WAXP', 'REQ'), ('WAXP', 'TR
AC'), ('XLM', 'RLC'), ('XMR', 'ZIL'), ('XNO', 'PIVX'), ('XRP', 'ARDR'), ('ZEC', 'GFT'),
('ZEN', 'MED')]
Number of pairs to be traded: 67
Pairs Updated date: 2023-01-01
Final pairs to be traded: [('ADA', 'MKR'), ('ADA', 'TRX'), ('ADX', 'FUN'), ('AMB', 'PIV
X'), ('ANT', 'BORG'), ('ANT', 'FUN'), ('ARK', 'FUN'), ('ARK', 'IOST'), ('BAT', 'FUN'),
('BCH', 'TRAC'), ('BTC', 'GFT'), ('CVC', 'FUN'), ('DASH', 'CTXC'), ('DASH', 'TRAC'),
('DOGE', 'XMR'), ('ELF', 'MLN'), ('ELF', 'NULS'), ('ENJ', 'FUN'), ('EOS', 'FUN'), ('EO
S', 'XMR'), ('ETC', 'GFT'), ('GAS', 'FUN'), ('GLM', 'FUN'), ('ICX', 'FUN'), ('IOTA', 'F
UN'), ('LINK', 'DENT'), ('LRC', 'FUN'), ('LSK', 'FUN'), ('LSK', 'SNT'), ('LTC', 'ANT'),
('MANA', 'FUN'), ('MLN', 'MDT'), ('MTL', 'FUN'), ('NEO', 'ARDR'), ('NEO', 'BLZ'), ('NE
O', 'FUN'), ('NEO', 'GFT'), ('NEO', 'XEM'), ('NEO', 'XVG'), ('NMR', 'REQ'), ('OMG', 'FU
N'), ('ONT', 'DCR'), ('ONT', 'FUN'), ('POWR', 'FUN'), ('POWR', 'IDEX'), ('PRO', 'FUN'),
('QTUM', 'DGB'), ('QTUM', 'FUN'), ('QTUM', 'MED'), ('QTUM', 'STMX'), ('REN', 'FUN'),
('REQ', 'IDEX'), ('RLC', 'FUN'), ('RVN', 'FUN'), ('SBD', 'FUN'), ('SNX', 'BTG'), ('SN
X', 'FUN'), ('SNX', 'SWFTC'), ('STEEM', 'FUN'), ('STORJ', 'FUN'), ('STRAX', 'FUN'), ('S
YS', 'FUN'), ('THETA', 'DENT'), ('TRAC', 'FUN'), ('UQC', 'SWFTC'), ('UTK', 'FUN'), ('VG
X', 'FUN'), ('VIC', 'FUN'), ('WAVES', 'FUN'), ('WAXP', 'FUN'), ('XDC', 'FUN'), ('XLM',
'FUN'), ('XNO', 'FUN'), ('XRP', 'BNB'), ('ZEC', 'FUN'), ('ZEN', 'FUN'), ('ZIL', 'FUN'),
```

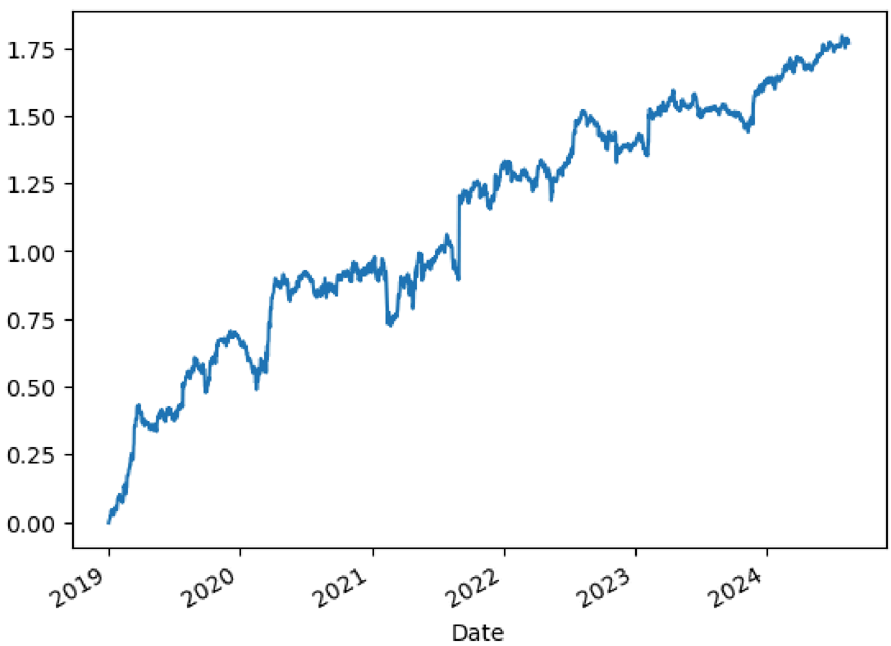
In [17]:
```python
print(metrics_df)
```

```
     Sharpe Ratio    Return  Volatility  Holding Period  Turnover  \
0.1      0.945908  0.000709    0.011894       20.229346  0.098866
0.2      0.878847  0.000683    0.012336       18.512020  0.108038
0.5      0.904451  0.000751    0.013189       16.181857  0.123595
0.7      0.967115  0.000859    0.014102       14.839511  0.134775

     Transaction Costs
0.1           0.000198
0.2           0.000216
0.5           0.000247
0.7           0.000270
```

The performance across different thresholds is relatively robust, with only minor variations.

In [18]:
```python
strat_net_ret.cumsum().plot()
```

Out[18]: <Axes: xlabel='Date'>



We also compare the performance of our strategy against our benchmark, specifically a buy-and-hold strategy for Bitcoin. We will evaluate key metrics including alpha and beta, maximum drawdowns and maximum drawdown duration.

In [19]:
```
buy_and_hold_btc = coins_ret['BTC'][start_of_out_sample:]
buy_and_hold_btc
```

Out[19]:
```
Date
2019-01-01   -0.030762
2019-01-02    0.027551
2019-01-03    0.020533
2019-01-04   -0.024701
2019-01-05    0.010259
                ...
2024-08-12   -0.034218
2024-08-13    0.009282
2024-08-14    0.021081
2024-08-15   -0.030726
2024-08-16   -0.018984
Name: BTC, Length: 2057, dtype: float64
```

In [20]:
```
full_sample_ret = pd.DataFrame({
    'strat_ret': strat_net_ret,
    'buy_and_hold_btc': buy_and_hold_btc
})

full_sample_stats = compute_stats(full_sample_ret)
print(full_sample_stats)
```

```
                       avg       vol    sharpe  hit_rate
strat_ret         0.216498   0.22386  0.967115  0.508994
buy_and_hold_btc  0.468641  0.541459  0.865516  0.516318
```

In [21]:
```
full_sample_ret.cumsum().plot()
```

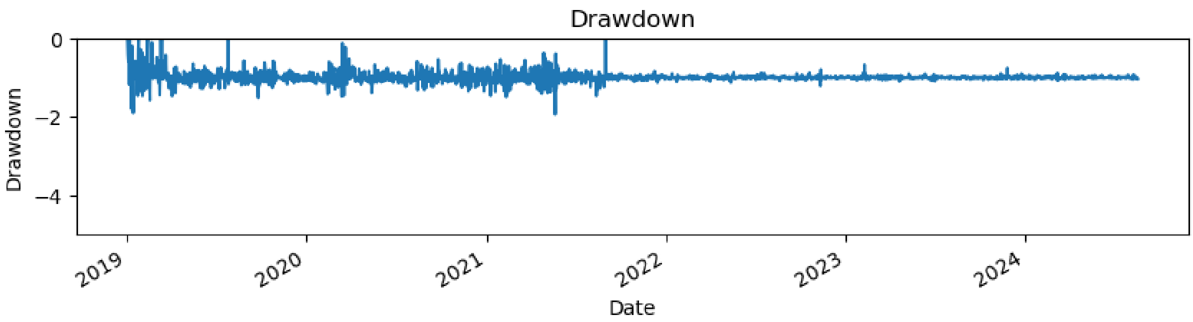
Out[21]: <Axes: xlabel='Date'>

In [22]:
```
corr = full_sample_ret.rolling(252).corr(full_sample_ret['buy_and_hold_btc'])
vol = full_sample_ret.rolling(252).std()

beta = (corr*vol).divide(vol['buy_and_hold_btc'], 0)

# Computing Point-in-Time Residual Returns
resid = full_sample_ret - beta.multiply(full_sample_ret['buy_and_hold_btc'], axis=0)
print(resid)
print(resid.corr())

# The information ratio
IR = resid.mean()/resid.std()*np.sqrt(252)
print(f"Information ratio is {IR['strat_ret']}")
```

```
             strat_ret  buy_and_hold_btc
Date
2019-01-01        NaN               NaN
2019-01-02        NaN               NaN
2019-01-03        NaN               NaN
2019-01-04        NaN               NaN
2019-01-05        NaN               NaN
...               ...               ...
2024-08-12  -0.009819      3.191891e-16
2024-08-13   0.004392     -8.673617e-17
2024-08-14  -0.001412     -2.012279e-16
2024-08-15  -0.001694      2.949030e-16
2024-08-16  -0.009178      1.804112e-16

[2057 rows x 2 columns]
                  strat_ret  buy_and_hold_btc
strat_ret          1.000000         -0.025434
buy_and_hold_btc  -0.025434          1.000000
Information ratio is 0.9016655515440292
```

The correlation of the residual returns with Bitcoin is -0.03

In [23]:
```
# Plot drawdown
dd = drawdown(full_sample_ret['strat_ret'])
plt.figure(figsize=(10, 2))
dd.plot()
plt.ylim(-5, 0)
plt.title('Drawdown')
plt.xlabel('Date')
plt.ylabel('Drawdown')
plt.show()
print(f"The max drawdown is {dd.min()}")
```




```
The max drawdown is -1.9360073186897857
```

In [24]:
```
ddd = duration(full_sample_ret.cumsum())
ddd
```

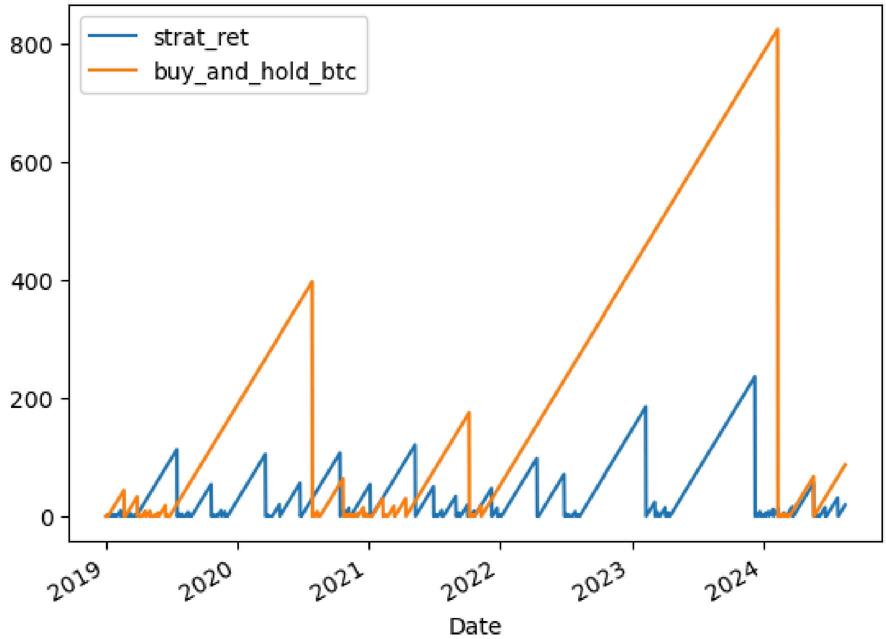
Out[24]:

	strat_ret	buy_and_hold_btc
Date		
2019-01-01	0	0
2019-01-02	0	0
2019-01-03	0	0
2019-01-04	0	1
2019-01-05	0	2
...	...	...
2024-08-12	15	83
2024-08-13	16	84
2024-08-14	17	85
2024-08-15	18	86
2024-08-16	19	87

2057 rows × 2 columns

In [25]:
```
ddd.plot()
```

Out[25]: <Axes: xlabel='Date'>



In [26]:
```
# maximum drawdown duration
ddd.max()
```

Out[26]:
```
strat_ret           236
buy_and_hold_btc    824
dtype: object
```

Overall, the cointegration strategy did not perform as well as expected. While cointegration is theoretically sound, its practical application can be complex and less effective. We will further explore correlation-based pairs trading in the next steps to seek more effective strategies.

In [ ]: