Problem addressed and technical model(s) applied

1. Project Goal: The primary objective is to implement a stat-arb strategy, as proposed in the paper by Avellaneda and Lee (2010), and evaluate its performance across a universe of 40 cryptocurrencies. This involves analyzing hourly price data of over 120 tokens, with a focus on the 40 with the largest market capitalizations, recorded over a period from February 19, 2021, to September 26, 2022.

2. Implementation Procedure: The procedure involves several key steps:

    - Factor Returns Computation: Computing factor returns for two risk factors at a specific time using a set of 40 largest market-cap tokens.
    - Principal Component Analysis (PCA): Applying PCA to the empirical correlation matrix of token returns to identify the principal components representing the risk factors. Regression Analysis: Performing linear regression on the hourly returns of each token to estimate regression coefficients and residual series.
    - Trading Signal Generation: Generating trading signals based on a set of parameters and the calculated s-score for each token.
    - Strategy Performance Evaluation: Evaluating the strategy's performance over a testing period, including calculating the Sharpe ratio and Maximum Drawdown (MDD).

3. Technical Models:
   PCA for Risk Factor Identification: PCA is used to extract the top principal components from the correlation matrix of token returns, which represent the underlying risk factors in the market.
   Linear Regression for Residual Analysis: This is applied to the hourly returns of each token to determine the relationship between the returns and the identified risk factors, along with the residuals.
   Statistical Metrics for Performance Evaluation: Metrics like the Sharpe ratio and Maximum Drawdown are used to assess the risk-adjusted returns and the potential for loss in the trading strategy.

4. Tasks and Deliverables: The project requires various tasks such as plotting cumulative return curves, eigen-portfolio weights, and evolution of s-scores for specific cryptocurrencies. The deliverables include ready-to-run codes, output files, and a comprehensive report discussing the implementation, outcomes, and practicality of the trading strategy.

Explanation of the functionalities of key classes/functions and their inputs/outputs

The `BackTesting.py` file contains a class `BackTesting` with its methods and functionalities. Here's an overview of the key components:

Class: `BackTesting`
- **Purpose**: To simulate the performance of the trading strategy over a specified period using historical data.
- **Attributes**:
  - `start_time`, `end_time`: Define the time frame for the backtesting.
  - `tokens_largest_cap`: List of tokens with the largest market capitalization.
  - `tokens_price`: Historical price data of the tokens.
  - `cash_value`: The initial capital for trading.
  - `window` (M): The time window for calculating statistical metrics.
  - `portfolio`: A dictionary to store the current portfolio status.
  - `portfolio_value`: To track the portfolio value over time.
- **Methods**:
  1. `__init__`: Initializes the backtesting with the specified parameters.
  2. `run_backtest`: Runs the backtest over the specified time frame, generating trading signals and executing trades.
  3. `trade`: Executes trades based on the generated signals.

Method: `run_backtest`
- **Inputs**: None (uses class attributes).
- **Process**:
  - Iterates through the time frame.
  - Creates `TradingStrategy` objects and generates trading signals.
  - Calls the `trade` method to execute trades based on signals.
  - Aggregates the signals and portfolio values over the period.
- **Outputs**: Updates class attributes with portfolio values and signals.

Method: `trade`
- **Inputs**:
  - `signal`: A DataFrame with trading signals for each token.
  - `time`: The current time for the trade.
- **Process**:
  - Iterates through each token and its trading signal.
  - Executes buying or selling based on the signal, adjusting the portfolio and cash value.
- **Outputs**: Updates the portfolio and cash values based on executed trades.

To fully understand the workings of the `BackTesting` class, we also need to look into the `TradingStrategy.py` and `Utility.py` files, which likely contain the `TradingStrategy` class and various utility functions used by `BackTesting`. I will analyze these files next.

The `TradingStrategy.py` file defines the `TradingStrategy` class with its methods and functionalities. Here's a summary:

Class: `TradingStrategy`
- **Purpose**: To implement the core logic of the statistical arbitrage trading strategy, including the calculation of factor returns, s-scores, and generating trading signals.
- **Attributes**:
  - `sbo`, `sso`, `sbc`, `ssc`: Thresholds for different s-scores used in signal generation (buy open, sell open, buy close, sell close).
  - `time`: The current time for which the strategy is being applied.
  - `tokens_price`: Price data of tokens.
  - `tokens_largest_cap`: List of tokens with the largest market capitalization.
  - `M`: The time window for statistical calculations.
- **Methods**:
  1. `__init__`: Initializes the trading strategy with specified parameters.
  2. `calculate_factor_returns`: Calculates factor returns based on the price data.
  3. `calculate_s_score`: Calculates the s-score for each token.
  4. `generate_signals`: Generates trading signals based on s-scores.

Method: `calculate_factor_returns`
- **Inputs**: None (uses class attributes).
- **Process**:
  - Selects a window of data based on `M` and `time`.
  - Calculates hourly returns, correlation matrix, eigenvalues, and eigenvectors.
  - Computes eigenportfolios and factor returns.
- **Outputs**: Updates class attributes with calculated data.

Method: `calculate_s_score`
- **Inputs**: None (uses class attributes).
- **Process**:
  - Calculates Ornstein-Uhlenbeck (OU) parameters for residuals.
  - Computes the s-score for each token.
- **Outputs**: Updates class attributes with s-scores.

Method: `generate_signals`
- **Inputs**: None (uses class attributes).
- **Process**:
  - Iterates over each token and its s-score.
  - Generates trading signals (buy open, sell open, close all, close long, close short, hold) based on s-score thresholds.
- **Outputs**: Returns a DataFrame with trading signals for each token.

The functionality of this class is closely linked with various utility functions likely defined in `Utility.py`. I will analyze this file next to provide a complete picture of how these components work together.

The `Utility.py` file contains the `Utility` class, which offers a suite of static methods to support various calculations and data processing tasks for the trading strategy. Here's a breakdown of its key functionalities:

Class: `Utility`
- **Purpose**: Provides utility functions for data manipulation, statistical calculations, and other necessary operations.
- **Attributes**: None (It's a static class and not meant to be instantiated).
- **Methods**: Static methods to perform specific tasks.

Key Methods:
1. **`select_window`**
   - **Inputs**:
     - `df`: DataFrame from which to select data.
     - `M`: Integer representing the number of hours for the time window.
     - `time`: Specific time point for data selection.
   - **Process**: Selects a slice of the DataFrame based on the specified time window.
   - **Outputs**: A DataFrame containing the selected data.

2. **`find_tokens`**
   - **Inputs**:
     - `df`: DataFrame containing token data.
     - `M`: Integer used to determine the frequency threshold.
   - **Process**: Identifies tokens that appear more than a certain threshold in the data.
   - **Outputs**: A list of frequently occurring tokens.

3. **`get_hourly_returns`**
   - **Inputs**:
     - `df`: DataFrame of token prices.
   - **Process**: Calculates the percentage change (hourly returns) for each token.
   - **Outputs**: A DataFrame with hourly returns.

4. **`calculate_correlation_matrix`**
   - **Inputs**:
     - `df`: DataFrame of returns or similar data.
   - **Process**: Normalizes the data and calculates the empirical correlation matrix.
   - **Outputs**: The correlation matrix.

5. **`principal_component_analysis`**
   - **Inputs**:

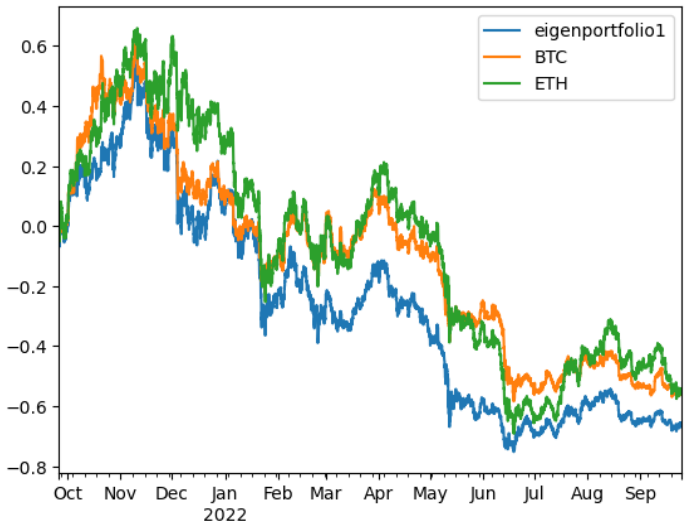
- `df`: DataFrame on which PCA is to be performed.
- `n_components`: Number of principal components to extract (default is 2).
- **Process**: Performs PCA to extract the principal components (eigenvectors) and their variances (eigenvalues).
- **Outputs**: Eigenvectors and eigenvalues of the PCA.

6. **`compute_eigenportfolios`**
- **Inputs**:
- `eigenvectors`: The eigenvectors from PCA.
- `hourly_returns`: DataFrame of hourly returns.
- `time`: Specific time for which the eigenportfolios are calculated.
- **Process**: Normalizes eigenvectors by the standard deviation of returns and computes eigenportfolios.
- **Outputs**: A DataFrame containing eigenportfolios.

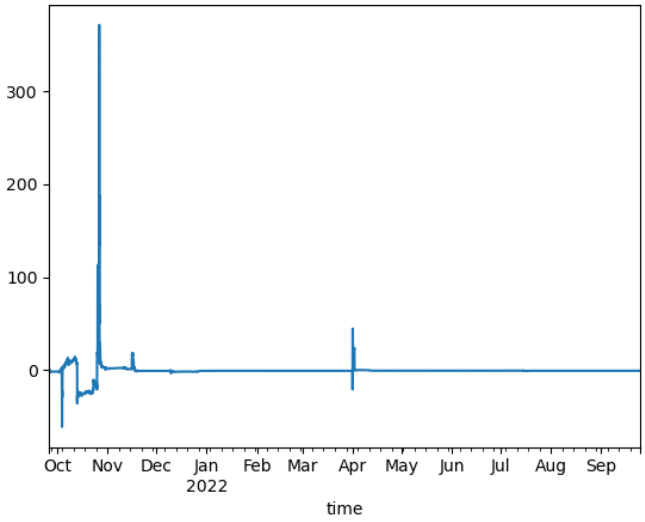
These utility functions are crucial for data processing and statistical analysis in the trading strategy. They support the `TradingStrategy` and `BackTesting` classes by providing the necessary calculations for factors, returns, correlation, PCA, and other key aspects of the statistical arbitrage strategy.

Outcome of the Implementation

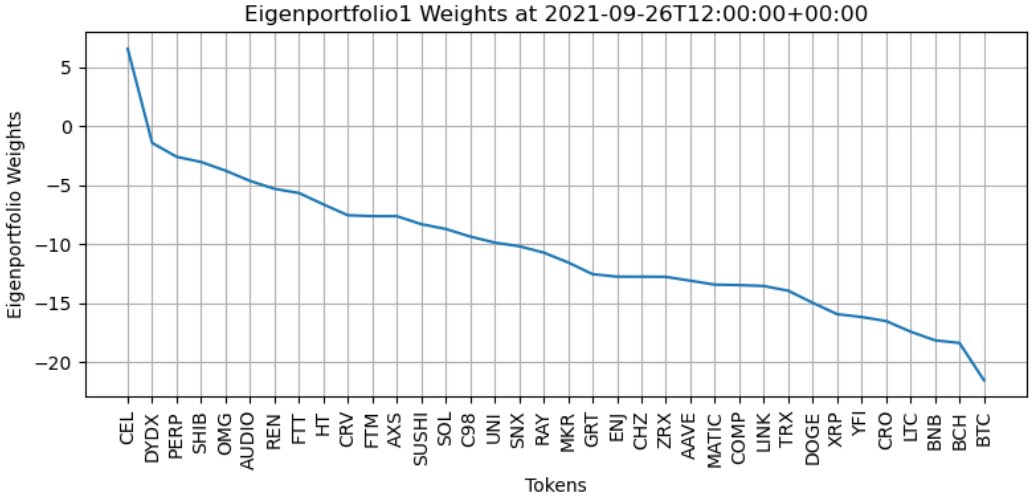
- The cumulative return curves of 4 assets over the testing period: the first eigen-portfolio, the second eigen-portfolio, BTC, and ETH.
The trend of the first eigen-portfolio, BTC and ETH

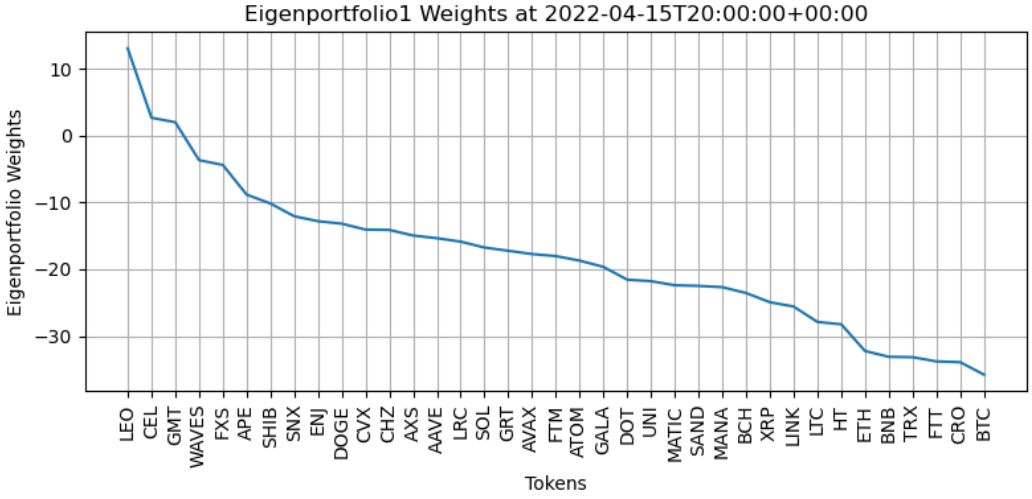


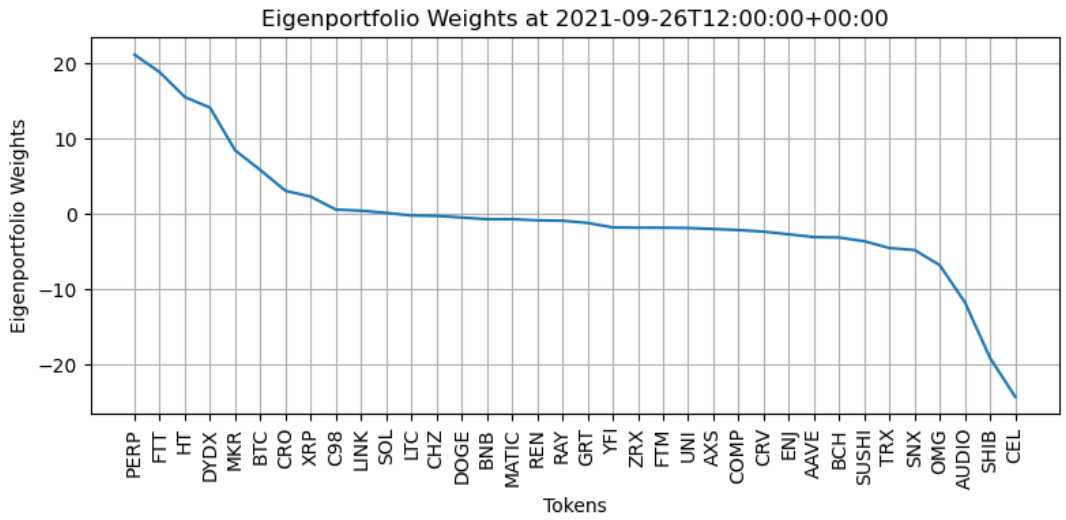
The trend of the second eigen-portfolio

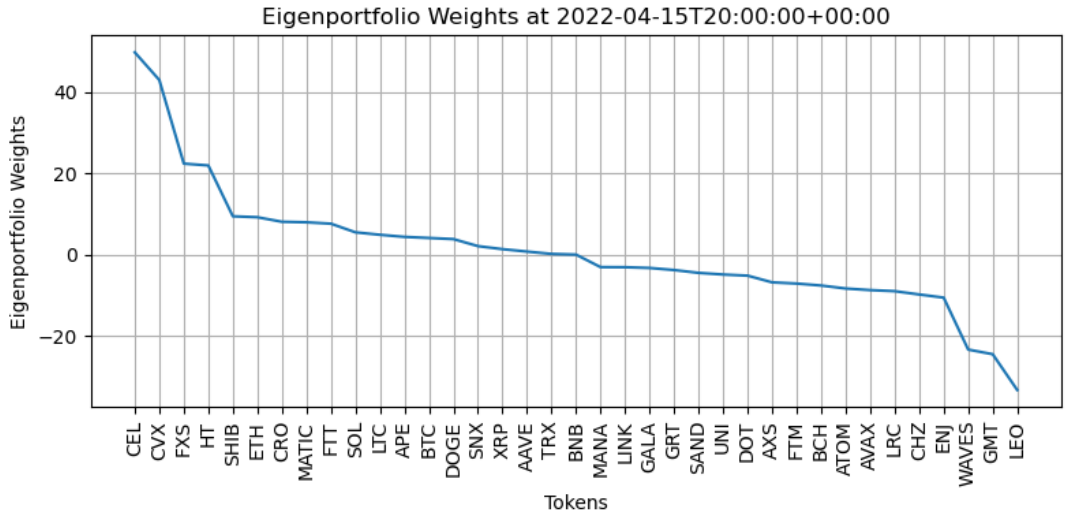


- Plot the eigen-portfolio weights of the two eigen-portfolios at 2021-09-26T12:00:00+00:00 and then at 2022-04-15T20:00:00+00:00

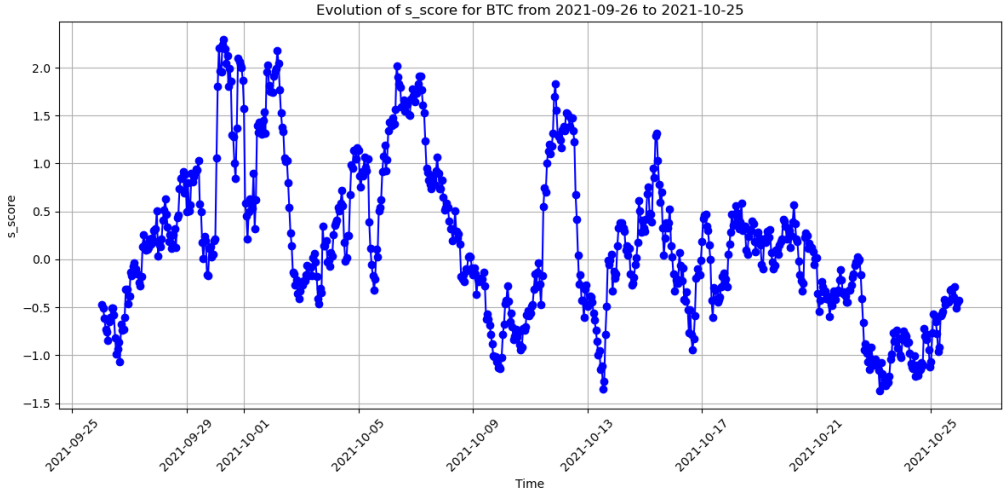


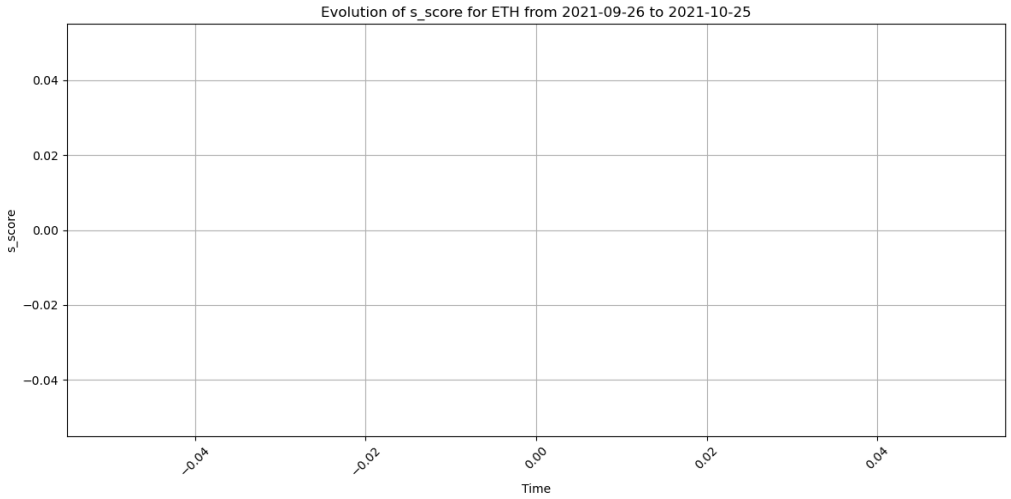




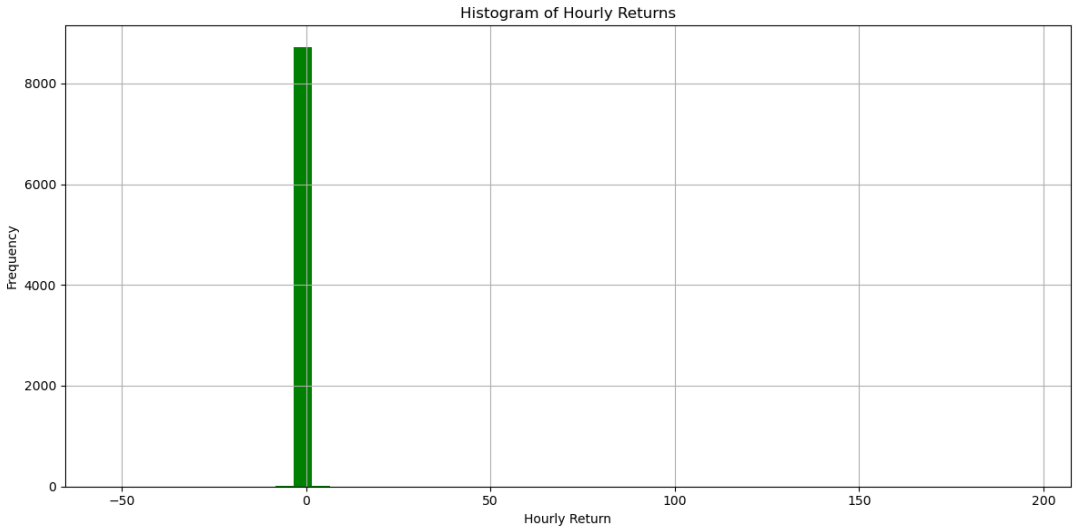


- Plot the evolution of s-score of BTC and ETH from 2021-09-26 00:00:00 to 2021-10-25 23:00:00 in two different figures.





- Plot the cumulative return curve of the implemented strategy and the histogram of the hourly returns



Sharpe Ratio

```
print("Here is the Sharpe ratio")
sharpe_ratio = backtest.calculate_sharpe_ratio()
print(sharpe_ratio)
```

✓ 0.0s

```
Here is the Sharpe ratio
0.627483597202618
```

Maximum Drawdown

```
print("Here is the maximum drawdown")
maximum_drawdown = backtest.calculate_maximum_drawdown()
print(maximum_drawdown)
```

✓ 0.0s

```
Here is the maximum drawdown
-5.1990804974490565
```

The csv files are stored in a folder called ThreeCSV.