Implementing a Statistical Arbitrage Strategy

Roy Gabriel
Michael Scott Smith

Data

The data was split into two categories: the top 40 crypto tokens based on market capitalization sorted by each hourly timestamp and the hourly price of over 120 crypto tokens. While the data was already sorted, further cleaning was needed. In order to handling missing data, we used forward fill because we are able to use some of the tokens that appeared in the top 40 market capitalization in the previous adjacent hours to fill the empty slots of the top 40 in the current hour.

In addition, due to missing data, as the moving window slides, it may experience an infinite return. This is because the price series would consist of several zeros before encountering a significant jump to say 50 USD or 100 USD. Thus, the return is very large according to the moving window. To account for this edge case, the algorithm does not trade those assets at the corresponding window. Intuitively, this makes sense because there may not exist enough data to achieve a statistically significant *s_score* signal. Moreover, the market just realized observations of the token price. Thus, it may not be as liquid as the algorithm assumes. There are several other reasons to list, which further cements the optimal choice to not trade those assets at the specific hour.

When standardization was needed (i.e., for normalized return series), we used the *StandardScaler* object from the *sklearn* package and specifically implemented the *fit_transform* method on the return series. While standardizing, the standard deviation of each token was saved to create the respective eigenportfolios.

Lastly, when appropriate, some data cleaning involved cleaning the dates column. The *Pandas* package was used, specifically the *to_datetime* method as well as a lambda function with some string manipulation.

Implementing the Procedure

We created several python files to implement the statistical arbitrage strategy: DataHandler.py, *Factors.py*, *StatArbStrategy.py*, *Trade.py*, and *main.py*.

DataHandler.py

This class creates the *DataHandler* class. When initialized, the class reads the two csv files containing the crypto token information and cleans the data as mentioned in the data section above. After the data is cleaned, the returns series is computed using the *Pandas pct_change()* method. Further data cleaning is done after the return series is obtained as mentioned in the data section. Lastly, it is noticeable that during the testing window from 2021-09-26 00:00:00 to

2021-10-25 23:00:00, ETH does not exist as a top 40 token. Thus, we implemented a quick fix solution. We replaced the smallest market cap token with ETH during this time window in order to plot the necessary s*_scores*.

This class was created in such a way that reading the csv files and the cleaning would only be done once, and the actual trading signal generation and trading would be done in a loop for each hour timestamp, yielding an O(n) time complexity, where n is the number of hourly timestamps.

Factors.py:

The *Factors.py* file includes the class *Factors*. The class then computes the factor returns by obtaining the corresponding eigenportfolios Q, using *sklearn's decomposition* library, specifically the *PCA* package, and the cumulative returns of the eigenportfolios.

StatArbStrategy.py

This python file defines the class *StatArbStrategy*. This class when initialized creates a *Factors* object and obtains the factor returns while saving the hourly returns and prices from the window of the current environment based on the parameter *window* (i.e., M = 240). Then, the residuals of the returns are estimated by running a linear regression for each of the top 40 tokens while saving the parameters. The residuals are then used in an autoregressive model to obtain the parameters *a* and *b*, which are then used to obtain the required variables to compute the *s_scores* for the trading signals (i.e., $\kappa_{c(i)}, m_{c(i)}, \sigma_{c(i)}, \sigma_{eq}^{c(i)}$) . There are times when the $b^2 > 1$, yielding an invalid $\sigma_{eq}^{c(i)}$. Those tokens were not traded during the corresponding hours. It is worth noting that we assumed the number of hours in a year were 8760, a constant used when calculating kappa and when annualizing the Sharpe Ratio metric.

The *s_scores* were then computed using the formula given in the research paper. We assumed $\bar{s}_{bo} = \bar{s}_{so} = 1.25,\ \bar{s}_{bc} = 0.75,\ \bar{s}_{sc} = 0.5$ as mentioned in the prompt, though they can be altered in the *generate_trading_signals* method. The function maps the conditions of the trading signals as outlined in the article as follows:

Table 1: Mapping the trading signals to its corresponding abbreviation.

Description	Abbreviation
Buy To Open	BTO
Sell To Open	STO
Close Long Position	CLO
Close Short Position	CSP

The linear regression models were implemented using *sklearn's linear_model* package, specifically, the *LinearRegression* class.

Most importantly, the *get_trading_signals* method returns a dictionary of all necessary outputs at a given observed time window (from start datetime to end datetime). It computes the eigenvectors corresponding to the largest two eigenvalues, their weights, the trading signals, and the *s_scores* as detailed in the project prompt.

Trade.py

The python file defines the class *Trade*. This class when initialized creates a *StatArbStrategy* object. This object is then used in the *run_strategy* method which uses the *get_trading_signals* method with a passed start datetime, end datetime, and a window size. When the trading signals are generated, the *trade* method maps the signals to their corresponding weights (1 denotes long, -1 denotes short, and 0 denotes no trade). In other words, we used *numpy's where* method to replace all signals that are *BTO* to 1, *STO* to -1, and 0 otherwise. After shifting the returns back one time step and multiplying by the trading signals, we obtain the forward-looking trades made at each time hour. This is an optimal approach in terms of execution speed since it uses the mathematical operators in the *Pandas* package, as opposed to looping through each time stamp and each column. There are a few other functions, such as the ones that compute the cumulative and average returns of the portfolio, Sharpe Ratio, and Maximum Drawdown.

The class also has plotting methods as desired by the prompt as well as a method to save the trading results into csv files.

main.py

This python file was used to wrap everything together and run the statistical arbitrage trading strategy given some inputs, like a start datetime and an end datetime.

Results

**All results pertain to a testing window from 2021-09-26 00:00:00 to 2022-09-25 23:00:00.*

The cumulative returns of the eigenportfolios corresponding to the two largest eigenvalues of the empirical correlation matrix of the top 40 tokens in each hour, ETH, and BTC, over the testing period can be seen in the Figure below.

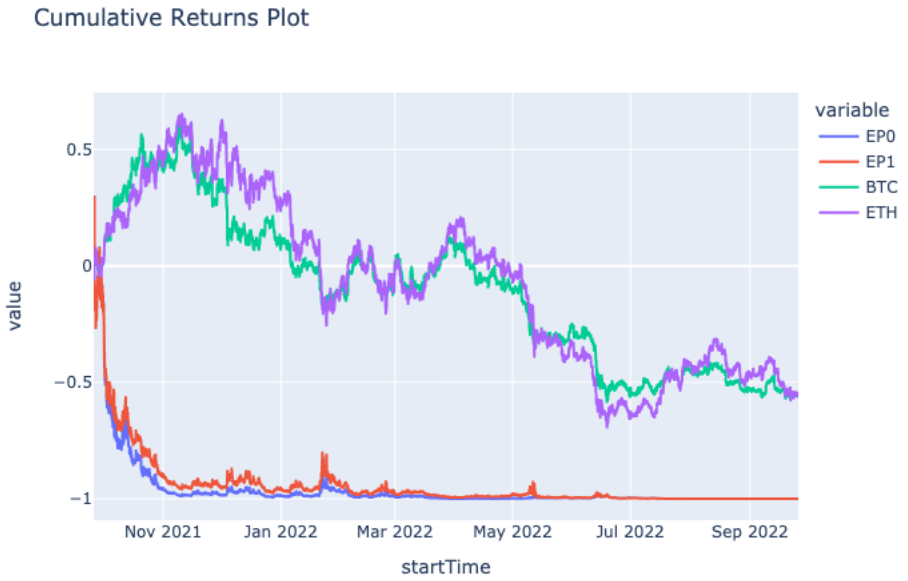

Figure 1: Cumulative Returns Plot.

As seen from the figure above, the cumulative returns of the two portfolios (EP0 and EP1) are highly correlated to one another, and slightly correlated to BTC and ETH, the generally largest two market cap tokens in our trading universe. Our eigenportfolios seem to have an exponential decline rather than a volatile negative decline as seen from BTC and ETH. Nevertheless, all eigenportfolios yielded negative overall cumulative return, and our eigenportfolios converged to -1, which shows a poorly chosen token universe, or that perhaps market cap is not the feature to use for PCA.

The eigenportfolios weights of the two eigenportfolios 2021-09-26T12:00:00+00:00 and 2022-04-15T20:00:00+00:00 can be seen in the Figure below.

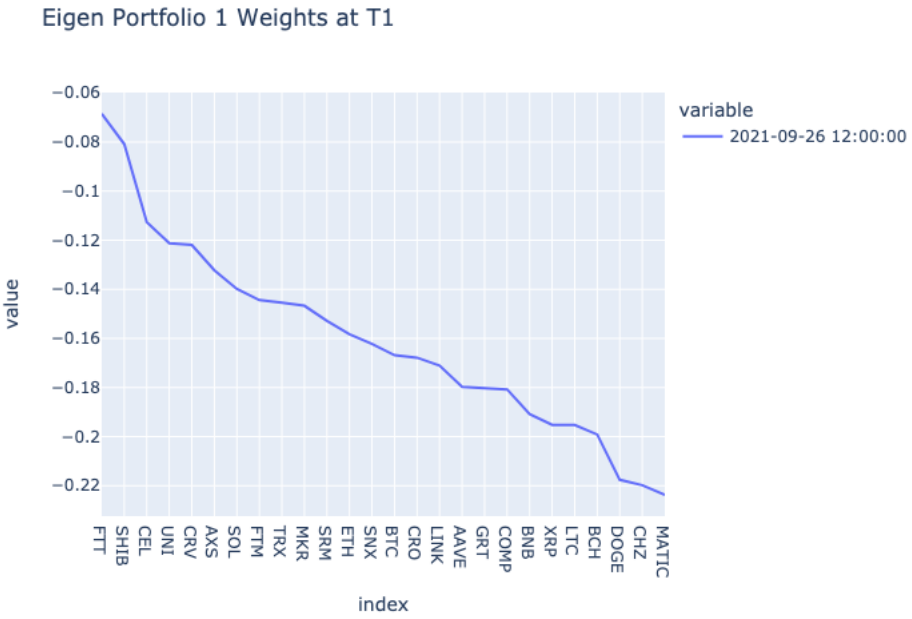

Figure 2: Eigen Portfolio 1 Weights at 2021-09-26 12:00:00.

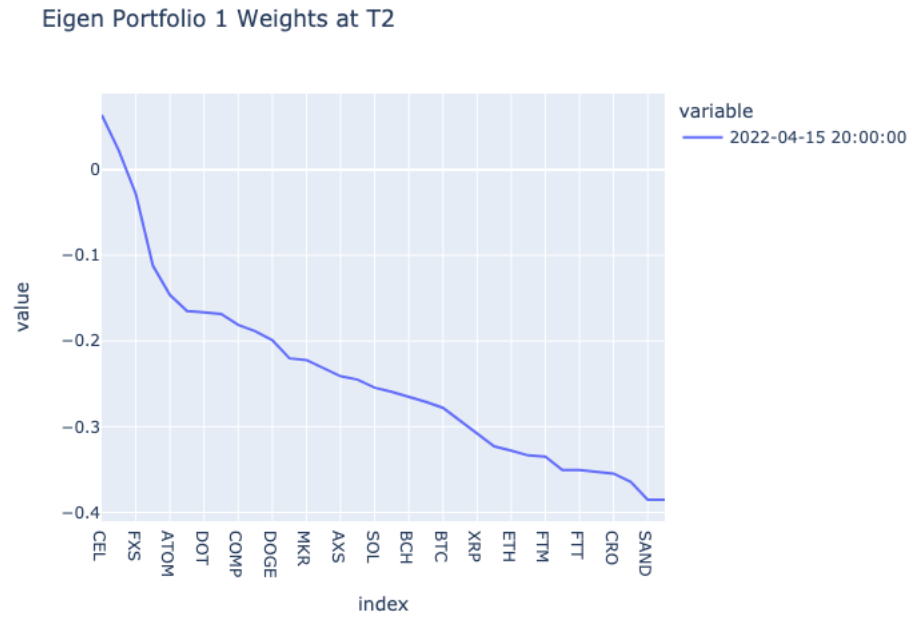

Figure 3: Eigen Portfolio 1 Weights at 2022-04-15 20:00:00.

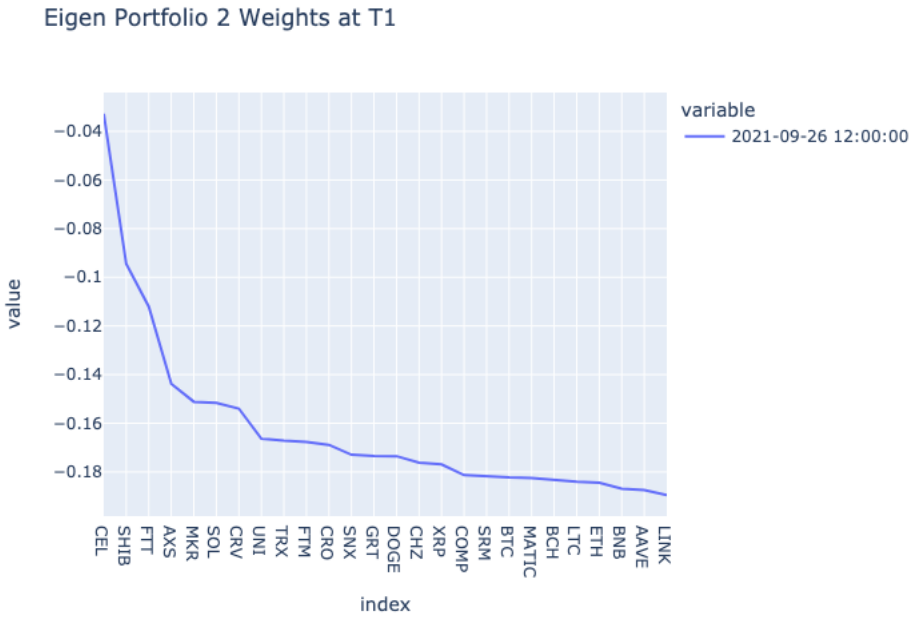

Figure 4: Eigen Portfolio 2 Weights at 2021-09-26 12:00:00.

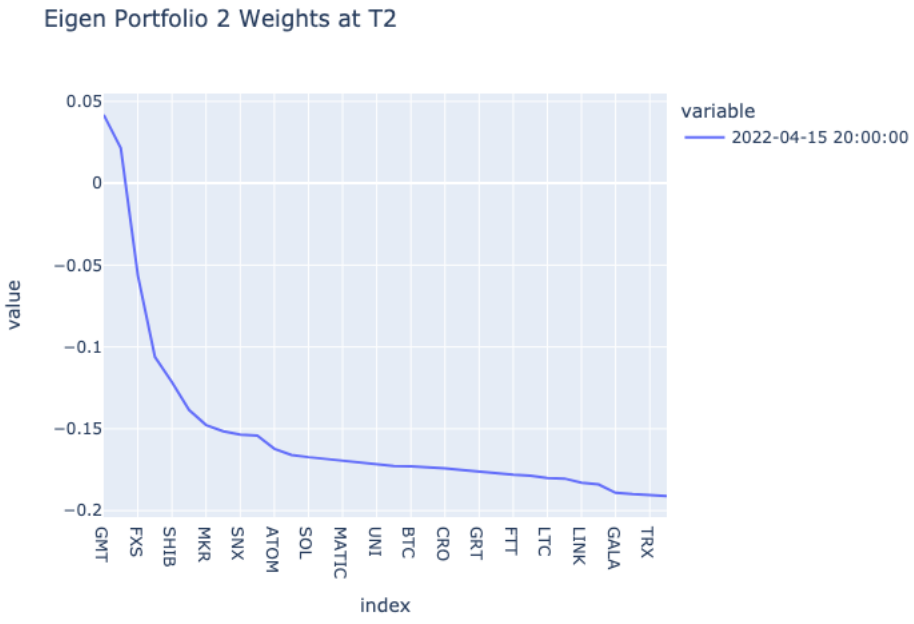

Figure 5: Eigen Portfolio 2 Weights at 2022-04-05 20:00:00.

As seen from the figures above, the weights match the behavior as that portrayed in the Statistical Arbitrage in US Equity Markets paper, though they are slightly less dispersed. Further, we oppositely match in terms of the signs in magnitude in the first and second eigenportfolios. Our eigenportfolios weights tend to me skewed to negative values. This makes sense since the research paper mentioned in the prompt refers to US Equity markets, which have been on a

bullish trend during the time window they ran their algorithm. However, our time window denotes a bearish crypto market.

The evolution of the *s_scores* of BTC and ETH fom 2021-09-26 00:00:00 to 2021-10-25 23:00:00 can be seen in the Figures below.

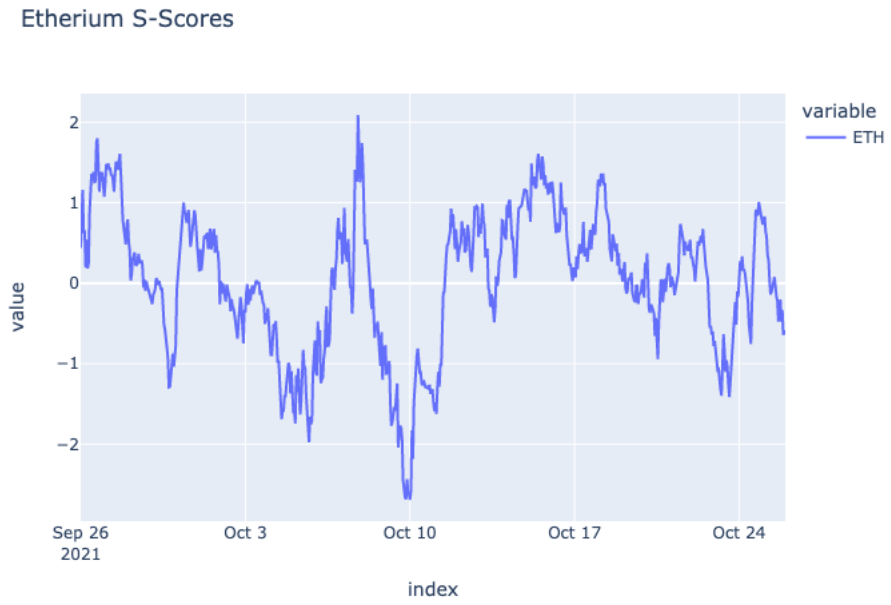

Figure 6: ETH S-Scores.

Figure 7: BTC S-Scores.

As seen from the figures above, the s_scores lie in the interval [-3, 3], which seems reasonable as per the research paper. It seems that the *s_scores* follow a stationary time series pattern and are heavily dispersed. This makes sense since we are looking at hourly data.

The cumulative return of the implemented strategy and the histogram of the hourly returns can be seen in the figures below.

Figure 8: Cumulative Strategy Returns.

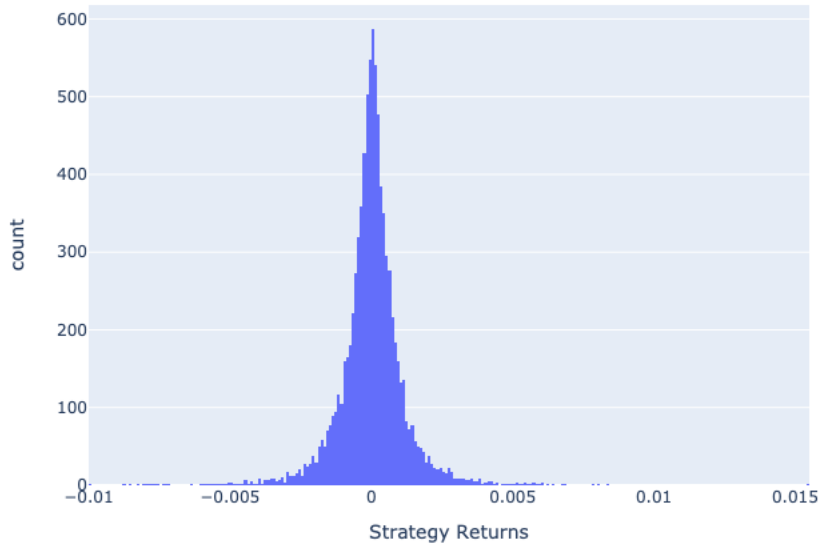

Figure 9: Histogram of hourly returns.

It seems our returns are somewhat centered around 0, with a very slight right skew. It would be worthwhile to test how significantly different than 0 it actually is. As for the cumulative return of the overall strategy, the returns are slightly positive, around 5%. It is worth noting that the most frequent hourly returns are centered around 0%. However, there are some instances when the hourly returns were +1.5% and -1%, though much less frequently.

Metrics:

The MDD plot can be seen below. It shows that our algorithm had its worst drawdown period during the beginning of September 2022 with a value of -3.19%. This is not as bad as if we were investing in ETH or BTC who had a worst maximum drawdown of roughly -50%.

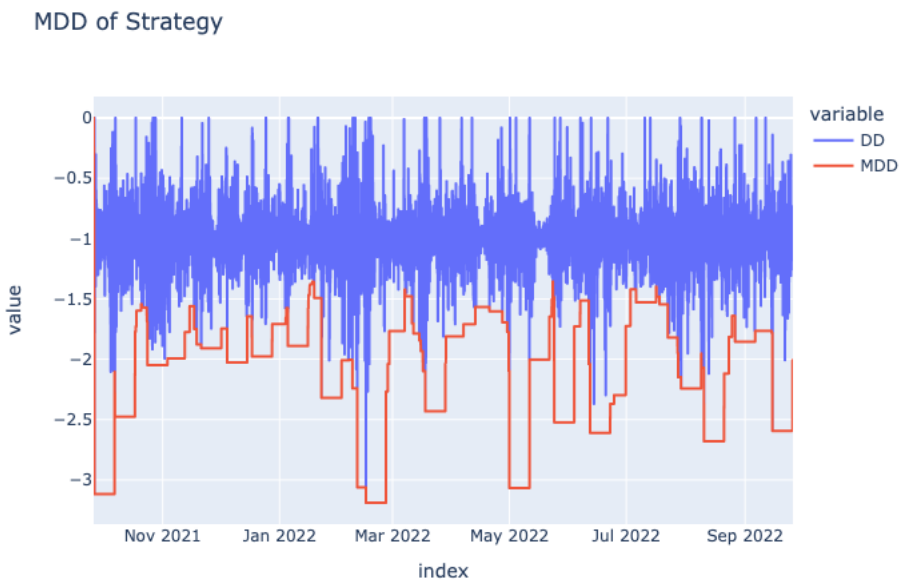

Figure 10: MDD of Strategy Returns

Further, as seen from the arbitrage strategy python console output below, the Sharpe Ratio was 0.54, and the MDD was -3.19%.

```
The Sharpe Ratio for Strategy is = 0.5413384024732039
The MDD for Strategy Returns = -3.1892279372590093
```

Figure 11: Python console output showing SR and MDD values.

Conclusions:

While it is an interesting idea to use such an arbitrage strategy, which yielded a positive though generally weak Sharpe Ratio, the universe of tokens needs to be taken into consideration. The

crypto currency market was going through a massive correction during the end of the time window, and during such times the statistical algorithms need to perform at their best, else they can blow up the account. However, we experienced the maximum drawdown periods during the end of the 2022 testing period, and it was less than the market maximum drawdown. Further, it is worth noting some assumptions made. We assumed all assets in our tradeable universe are tradeable long and short. We assumed we crossed the market at the price shown in the dataset, regardless of the bid-ask spread. We assumed the price data and market cap sorting was valid and void of human error. Lastly, we assumed we had sufficient capital and margin to execute the long-short arbitrage strategy.

Additional Considerations

A few other backtests need to be made, such as tweaking the *s* values to generate trading signals, the window (M = 240), number of tokens to keep as the largest market cap tokens (change the number 40), and perhaps feature engineer other variables. For example, instead of using the largest market capitalized tokens at a given hour, perhaps we can use the tightest markets (smallest bid-ask spreads), or largest volume (based on sentiment or magnitude), etc. Further metrics can be used such as statistically testing whether our returns are different than 0, computing alpha (using CAPM, Jensen's alpha, etc.), and using another benchmark besides 0%.