Equity Portfolio Optimization and Day Trading Using Markov Decision Process

Aim: Build a stochastic transition model in the form of a day trading bot using Markov Decision Process. The model should choose when to buy (1) and when to not buy (0) a stock. The model assumes a holding period of 24 hours for each stock where it is bought and sold at the auction close. £50 worth of each stock will be bought at a time.

Data for this project was sourced using Yahoo Finance and the yfinance API package in Python.

Methodology:

1. Identify three different stocks: two that are inversely correlated and another that is loosely correlated to both stocks based on rolling average of their daily returns
2. Download time series data for three different macroeconomic indicators: the S&P 500 index Price (GSPC), US 10 Treasury Bill (TNX) and the Gold Futures Price (GC=F) and analyze their daily return
3. Join stock data and macroeconomic data together and assess their correlation
4. Enrich and clean data ready for processing: splitting of the train and test data, identifying states, their respective actions and probabilities
5. Run the training data through a Markov Decision Process (assuming γ=0.9 and θ=0.00000001), using a Value Iteration algorithm to optimize a policy for each of the states.
6. Backtest the policy against the test data and evaluate results

1) Identify Three Stocks

The three different stocks will be chosen from a potential 4000 that are available from Yahoo Finance. The model is agnostic to which sectors the three stocks will be chosen from.

A random 50 stocks from the 4000 are selected and each pair's linear relationship is measure using Pearson's Correlation Coefficient:

$$\rho = \frac{\text{cov}(X,Y)}{\sigma_x \sigma_y}$$

The outcome can be seen in the form of a correlation heatmap found at the end of the document.

The three stocks being used in the model were identified using the following code:

```
import numpy as np
stock1 = corr_matrix.unstack().sort_values().index[0][0]
stock2 = corr_matrix.unstack().sort_values().index[0][1]

# Find stock loosely correlated to both
abs_corr = [sum(np.abs(corrs)) for corrs in zip(corr_matrix[stock1],
                                    corr_matrix[stock2])]
stock3index = np.argmin(abs_corr)
stock3 = corr_matrix[stock1].index[stock3index]
```

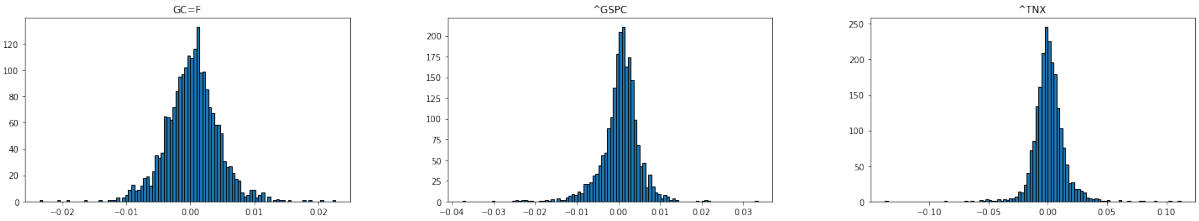
Stocks chosen were:

1. Enerflex Ltd. (EFXT) - a worldwide supplier of products and services to the global power generation and gas production industry, based in Calgary, Alberta
2. BlackRock MuniYield Quality Fund II, Inc. (MQT) - provide shareholders with as high a level of current income exempt from federal income taxes as is consistent with its investment policies and prudent investment management.
3. Eli Lilly and Company (LLY) - an American pharmaceutical company headquartered in Indianapolis, Indiana, with offices in 18 countries.

2) Macroeconomic Indicators

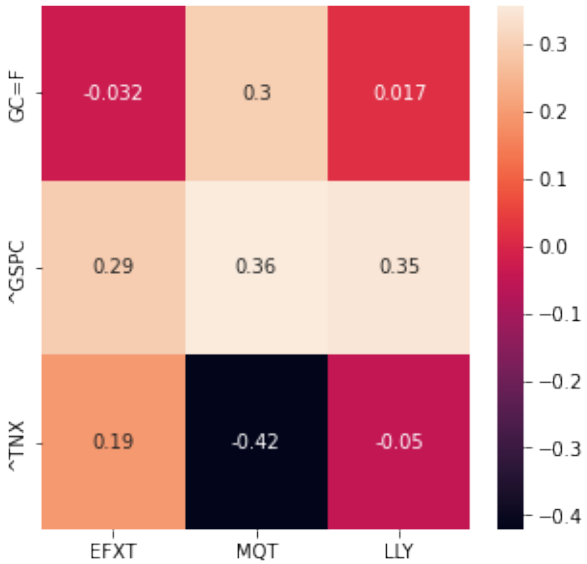
The S&P 500 index Price (GSPC), US 10 Treasury Bill (TNX) and the Gold Futures Price (GC=F) were chosen as macroeconomic indicators to be used within the model.

Their rolling daily returns over a period of time can be seen in the below histograms:



3) Analyze correlation between stock and macroeconomic indicators

The below highlightss the correlation between stocks and macroeconomic indicators:



4) Enrich data ready for processing

For the purpose of saving computational power and time, the daily returns of each of the data points where normalized to 0 or 1, where 1 is a positive return and 0 is not a positive daily return. The T+1 daily returns for the stock where also created as this is what the model is looking to predict. The data is split into a test and a training set.

```
from sklearn.model_selection import train_test_split

mix_df_binary = np.sign(mix_df).replace(-1, 0).dropna()
mix_df_binary[f'{stock1}+1'] = mix_df_binary[stock1].shift(1)
mix_df_binary[f'{stock2}+1'] = mix_df_binary[stock2].shift(1)
mix_df_binary[f'{stock3}+1'] = mix_df_binary[stock3].shift(1)
mix_df_binary = mix_df_binary.dropna()
mix_df_binary_train, mix_df_binary_test =
                                    train_test_split(mix_df_binary,
                                    test_size=0.2,shuffle=False)

stocks = [stock1, stock2, stock3]

mix_df_binary_train
```

Output:

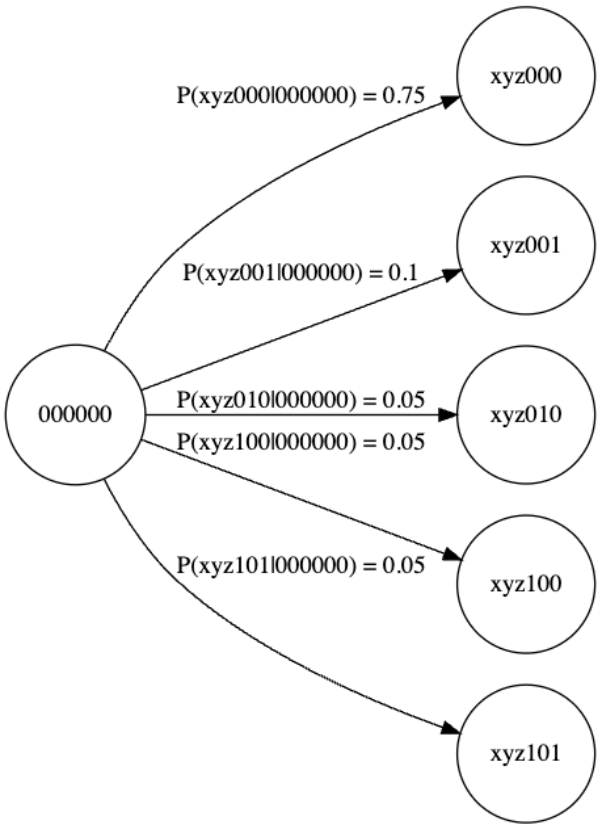
Date	GC=F	^GSPC	^TNX	EFXT	MQT	LLY	EFXT+1	MQT+1	LLY+1
2015-01-12	1.0	1.0	0.0	0.0	1.0	1.0	0.0	1.0	0.0
2015-01-13	1.0	1.0	0.0	0.0	1.0	0.0	0.0	1.0	1.0
2015-01-14	1.0	0.0	0.0	0.0	1.0	1.0	0.0	1.0	0.0
2015-01-15	1.0	0.0	0.0	0.0	1.0	0.0	0.0	1.0	1.0
2015-01-16	1.0	0.0	0.0	0.0	1.0	1.0	0.0	1.0	0.0

The probabilities for each unique T+1 state, given the current state, were then calculated to provide a probability matrix for the model to train on:

	GC=F	^GSPC	^TNX	EFXT	MQT	LLY	EFXT+1	MQT+1	LLY+1	prob
0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.750000
1	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	1.0	0.100000
2	0.0	0.0	0.0	0.0	0.0	0.0	0.0	1.0	0.0	0.050000
3	0.0	0.0	0.0	0.0	0.0	0.0	1.0	0.0	0.0	0.050000
4	0.0	0.0	0.0	0.0	0.0	0.0	1.0	0.0	1.0	0.050000

The above screenshot can be read as followed: given the model is in state where GC=F, GSPC, TNX, EFXT, MGT and LLY all had negative daily returns, the probability of the next state being an unknown return for the macro indicators (xyz) and a succeeding negative return for all three stocks. The second row shows a 0.1 probability for LLY to have a positive return the next day, TNX and EFXT having negative returns.

This can be seen here in the following graph:



2) Process Training Data Using Markov Decision Process (MDP)

The aim of the MDP is to take the optimal sequential decision under uncertainty. In this case, it is to decide whether it not to buy a stock based on the previous state and the reward for buying such stock. This can show mathematically as:

$$p(s_{t+1}, r_{t+1}|s_0, a_0, r_1, \dots, r_t, s_t, a_t) = p(s_{t+1}, r_{t+1}|s_t, a_t) \tag{1}$$

The reward for buying the stock will be 1 multiplied by the discount rate:

$$G_t = r_{t+1} + \gamma r_{t+2} + \gamma^2 r_{t+3} + \cdots = \sum_{k=0}^{\infty} \gamma^k r_{t+k+1} \tag{2}$$

Where γ is anything between 0 and 1. The discount rate determines whether to prioritize short or long-term rewards. A value of zero means that the metric only measures the next reward.

We can calculate the expected value of any given state under a given policy. This is called the state-value function:

$$V^{\pi}(s) = \mathbb{E}_{\pi}[G_t|s_t = s] \tag{3}$$

Once the state-values for each new state, given and action, the maximum expected value should be taken to determine the optimal policy. This can be denoted using the Bellman's Optimality Equation:

$$V_{k+1}(s) = \max_a \sum_{s',r} p(s', r|s, a) \left(r + \gamma V_k(s')\right) \tag{4}$$

See the code below for quantifying each state-value and taking the maximum value:

```
    #  Get expected / state values
    sub_df['V_pi'] = sub_df['prob'] * ((sub_df[f'{stock1}+1'] +
                                        sub_df[f'{stock2}+1'] +
                                        sub_df[f'{stock3}+1'] ) +
                                        (gamma * sub_df['V_pi_old']))
    v_pi_new = max(sub_df['V_pi'])
```

Now, given some arbitrary initial guess for the optimal state-value (e.g., zero), we can apply this rule over and over until the maximum difference in any update falls below some small threshold. The algorithm for doing this all called Value iteration:

Initialize array v arbitrarily (e.g., $v(s) = 0$ for all $s \in \mathcal{S}^+$)

Repeat
  $\Delta \leftarrow 0$
  For each $s \in \mathcal{S}$:
    $temp \leftarrow v(s)$
    $v(s) \leftarrow \max_a \sum_{s'} p(s'|s,a)[r(s,a,s') + \gamma v(s')]$
    $\Delta \leftarrow \max(\Delta, |temp - v(s)|)$
until $\Delta < \theta$ (a small positive number)

Output a deterministic policy, π, such that
  $\pi(s) = \arg\max_a \sum_{s'} p(s'|s,a)\left[r(s,a,s') + \gamma v(s')\right]$

The full code for optimizing the policy can be found in the appendix.

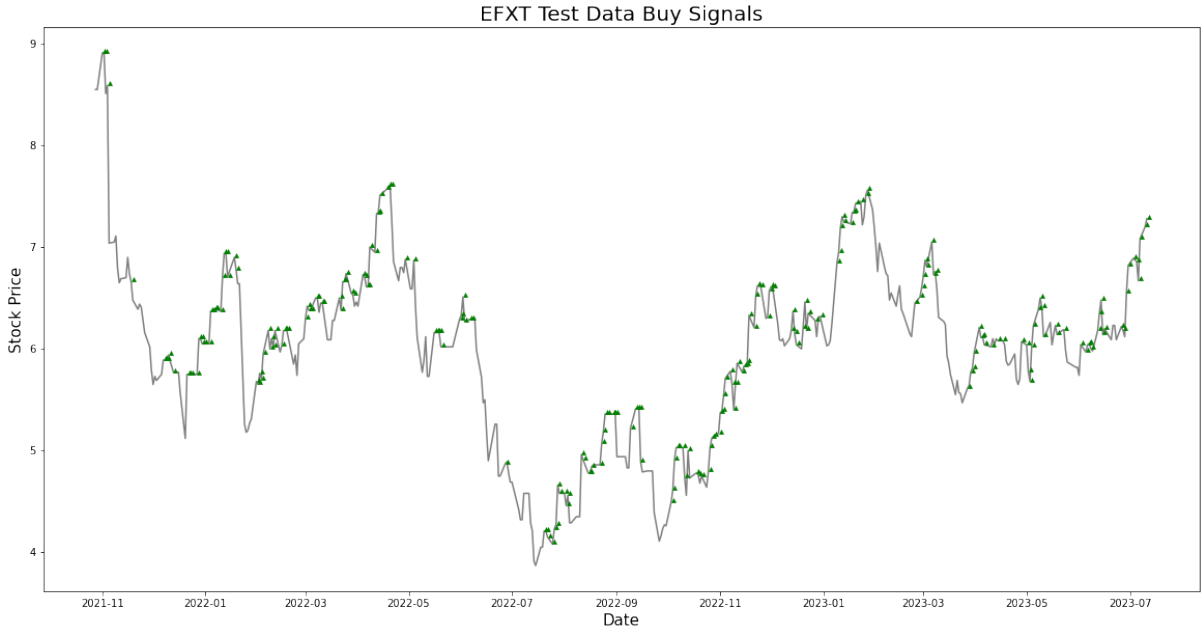
4) Enrich data ready for processing

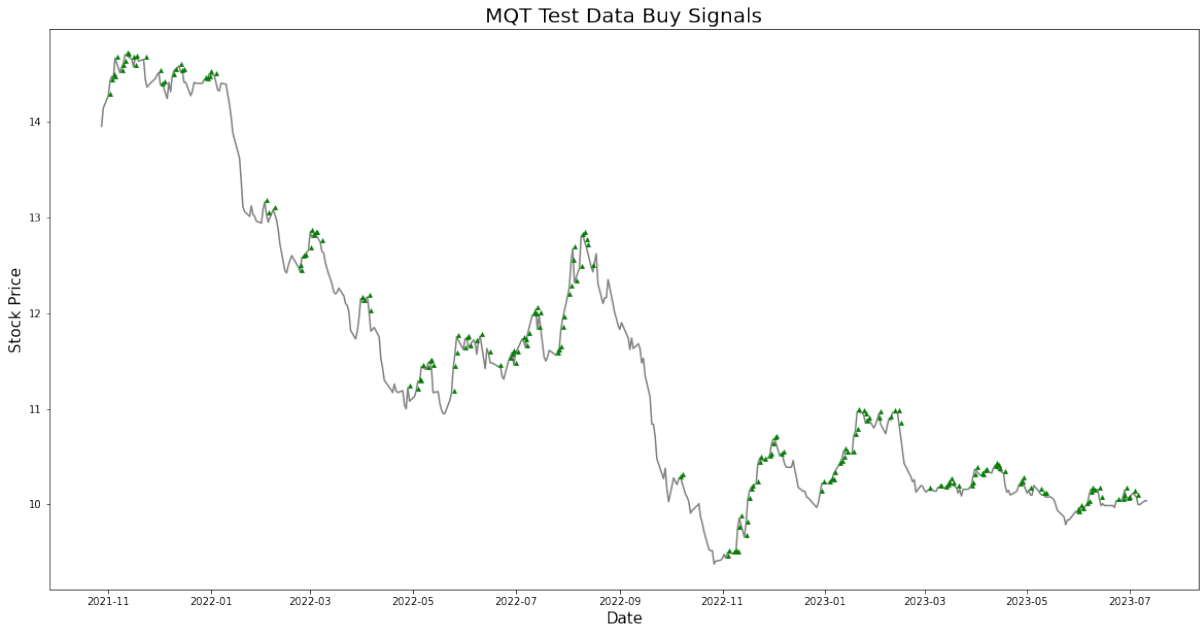
Once the training data has processed and the optimal actions / policy for each of the states have been identified. We map the policy back on to the testing data so that the data has an action for each trading day.

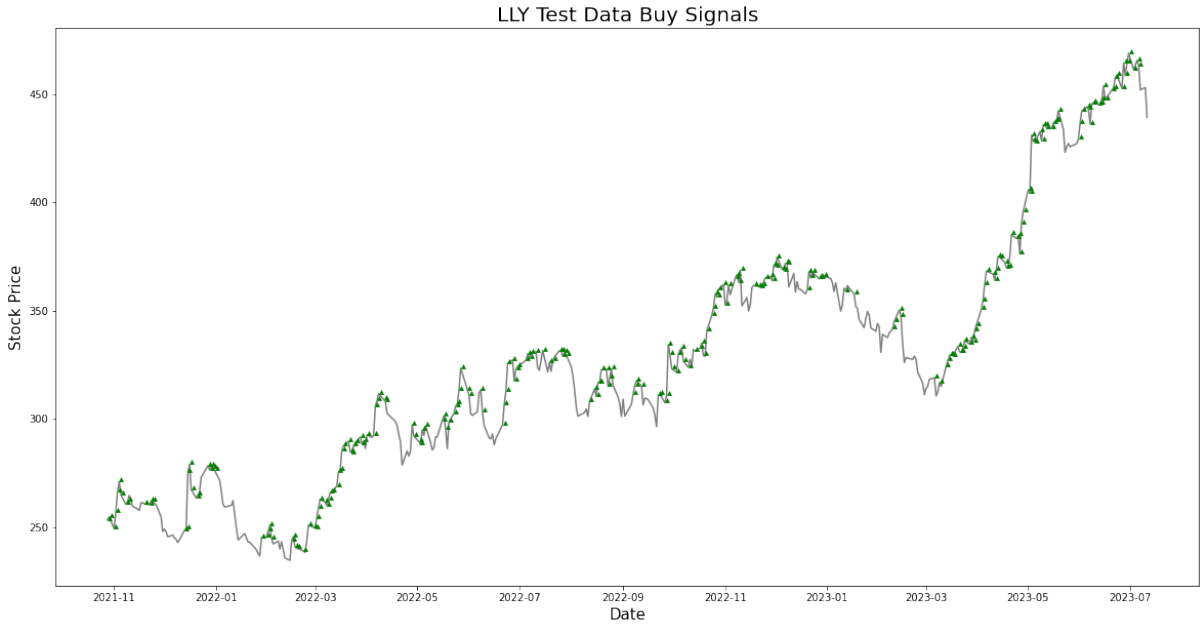
Once each trading day within the test data has an action the daily and cumulative returns can be calculated:

```
  for stock in stocks:
    ts_aciton[f'DReturns_{stock}'] =
                        ts_aciton[f'{stock}'].pct_change().shift(-1)
    ts_aciton[f'RR_{stock}'] = (50/ts_aciton[f'{stock}']) *
                                        ts_aciton[f'{stock}'] *
                                        ts_aciton[f'{stock}_A'] *
                                        ts_acitom[f'DReturns_{stock}']
    ts_aciton[f'CReturns_{stock}'] =
          ts_aciton[f'RReturns_{stock}'].cumsum()
ts_aciton['CreturnsALL'] = ts_aciton[f'CReturns_{stock1}'] +
                                    ts_aciton[f'CReturns_{stock2}'] +
                                    ts_aciton[f'CReturns_{stock3}']
```

Once completed, the plots for each stock were made (the green triangles represent buy signals):

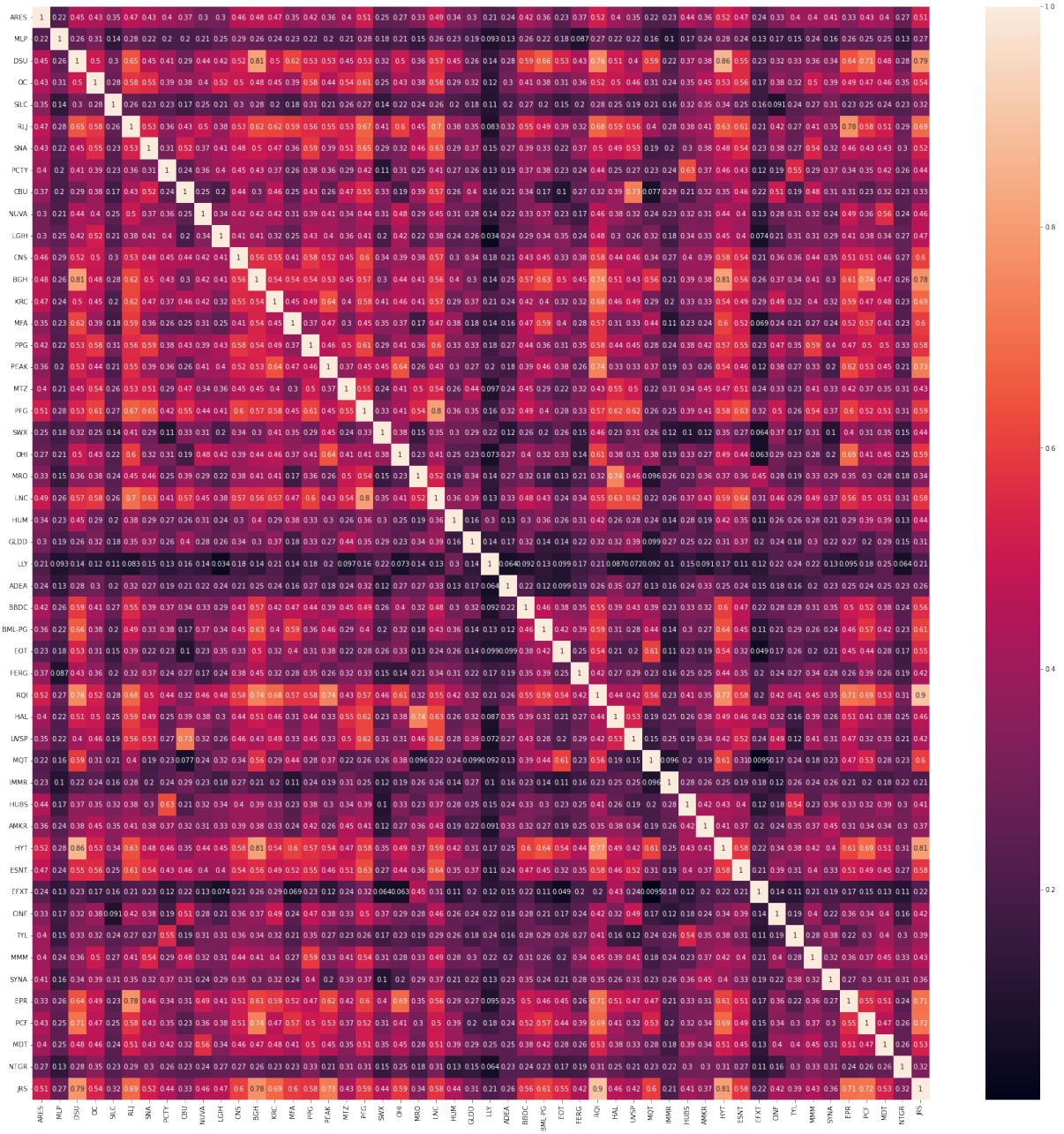






As you can see, the MDP trading strategy was optimized to not buy during times of ongoing negative returns in a stock. Potential improvements could be to accommodate a penalty when day trading in high levels of volatility or to incorporate some longer term mean reversion signals.

Figure 1: Correlation Heatmap



Markov Decision Process Algorithm

```
pm = prob_matrix
pm_cols = pm.columns
# Set constants
gamma = 0.9
theta = 0.00000001
#  Create a lookup of known states
pm['Lookup'] = pm[pm_cols[0]].astype(str) +
                                    pm[pm_cols[1]].astype(str) +
                                    pm[pm_cols[2]].astype(str) +
                                    pm[pm_cols[3]].astype(str) +
                                    pm[pm_cols[4]].astype(str) +
                                    pm[pm_cols[5]].astype(str)

# Starting optimal variables
pm['V_pi'] = 0
pm['V_pi_old'] = 0
pm['V_pi_diff'] = 0

# Get the unique known states in order to filter the df
uv = [''.join([str(y) for y in x])
      for x in pm[pm_cols[:6]].drop_duplicates().to_numpy()]

# Initialise optimal policy
optimal_policy = []

# Value Iteration algorithm
for i, v in enumerate(uv):
    # For each unique state, assign a starting optimal expected
                                        value and policy
    v_pol = None
    v_pi = 0

    # Filter probability matrix with the unique state
    sub_df = pm[pm.Lookup==v]

    # Iterate until abs(v_pi_new - v_pi) < theta
    while True:

        # Create copy of olf v_pi
        sub_df['V_pi_old'] = sub_df['V_pi']
        v_pi = max(sub_df['V_pi_old'])

        #  Get expected / state values
        sub_df['V_pi'] = sub_df['prob'] * ((sub_df[f'{stock1}+1'] +
                                            sub_df[f'{stock2}+1'] +
                                            sub_df[f'{stock3}+1'] ) +
                                             (gamma *
                                            sub_df['V_pi_old']))
        v_pi_new = max(sub_df['V_pi'])
        # Compare against theta, iterate again, or update optimal
                                            policy
        if abs(v_pi_new - v_pi) < theta:
            v_pol = sub_df[sub_df.V_pi==v_pi_new].head(1)
            v_pi = v_pi_new
            optimal_policy.append([v_pol[pm_cols[:9]], v_pi])
            break
```