DiscreteAllocation

Converting the continuous weights (output by optimisation methods) into an actionable allocation.

For example, if we have $10,000 that we would like to allocate. If we multiply the weights by this total portfolio value, the result will be dollar amounts of each asset. So if the optimal weight for Apple is 0.15, we need $1500 worth of Apple stock. However, Apple shares come in discrete units say $190, so we will not be able to buy exactly $1500 of stock. The best we can do is to buy the number of shares that gets us closest to the desired dollar value.

The optimisation problem is thus given by:

```
Minimize:\ r + ||wT - (x \circledcirc p)||_1\\
Subject\ to:\ r + x.p = T\\\\

Here:\\
T \in \mathbb{R} \ be\ the\ total\ value\ of\ the\ portfolio.\\
p \in \mathbb{R}^n \ be\ the\ array\ of\ latest\ prices\ for\ n\ assets.\\
w \in \mathbb{R}^n \ be\ the\ set\ of\ target\ weights.\\
x \in \mathbb{Z}^n \ be\ the\ integer\ allocation\ i.e.\ code\ result.\\
r \in \mathbb{R}^ \ be\ the\ remaining\ unallocated\ value.\\
\circledcirc \ denotes\ element-wise\ multiplication.\\
||\ ||_1\ represents\ the\ L_1\ norm, which\ is\ the\ sum\ of\ absolute\ values.\\
```

Essentially, we're trying to minimise both the amount of money leftover as well as the deviation from ideal values.

Code:

```
if short_ratio is None:
            self.short_ratio = sum((-x[1] for x in self.weights if x[1] < 0))
        else:
            self.short_ratio = short_ratio
```

Summing the weights of the assets that have negative weights (indicating a short position). This is done to determine the short ratio as a proportion of the total portfolio value.

When you go short on an asset, you effectively have a negative exposure to that asset's performance. For example, if the weight of a stock in a portfolio is -0.1, it means that you have a short position in that stock representing 10% of the total portfolio value.

Negative weights for short positions are used in mathematical models and algorithms to indicate the direction of the investment strategy. They help represent the fact that the investor is selling (shorting) the asset rather than buying it.

Greedy Algorithm:

It proceeds in two 'rounds'. In the first round, we buy as many shares as we can for each asset without going over the desired weight. In the Apple example, 1500/190 ≈ 7.89, so we buy 7 shares at a cost of $1330.

After iterating through all of the assets, we will have a lot of money left over (since we always rounded down).

In the **second round**, we calculate how far the current weights deviate from the existing weights for each asset. In the Apple example this would be a deviations of 0.15 - 0.133. Some assets will have a higher deviation from the ideal, so we will purchase shares of these first. We then repeat the process, always buying shares of the asset whose current weight is furthest away from the ideal weight.

Drawbacks:

- Money can be minimised but will be left-over
- Allocation method may deviate rather widely from the desired weights, particularly for companies with a high share price

Code:

```python
def greedy_portfolio(self, reinvest=False, verbose=False):
        """
        Convert continuous weights into a discrete portfolio allocation
        using a greedy iterative approach.
        """
        # Sort in descending order of weight
        self.weights.sort(key=lambda x: x[1], reverse=True)

        # If portfolio contains shorts
        if self.weights[-1][1] < 0:
            #Longs contains the tickers and weights of assets with non-negative weights
            #Shorts contains the tickers and absolute values of weights for assets with negative weights.
            longs = {t: w for t, w in self.weights if w >= 0}
            shorts = {t: -w for t, w in self.weights if w < 0}

            #Normalisation i.e. summing to 1
            long_total_weight = sum(longs.values())
            short_total_weight = sum(shorts.values())
            longs = {t: w / long_total_weight for t, w in longs.items()}
            shorts = {t: w / short_total_weight for t, w in shorts.items()}

            #Construct long-only discrete allocations for each
            short_val = self.total_portfolio_value * self.short_ratio
            long_val = self.total_portfolio_value
            if reinvest:
                long_val += short_val

            if verbose:
                print("\nAllocating long sub-portfolio...")
            da1 = DiscreteAllocation(
                longs, self.latest_prices[longs.keys()], total_portfolio_value=long_val
            )
            long_alloc, long_leftover = da1.greedy_portfolio()

            if verbose:
                print("\nAllocating short sub-portfolio...")
            da2 = DiscreteAllocation(
                shorts,
                self.latest_prices[shorts.keys()],
                total_portfolio_value=short_val,
            )
            short_alloc, short_leftover = da2.greedy_portfolio()
            short_alloc = {t: -w for t, w in short_alloc.items()}

            # Combine and return
            self.allocation = long_alloc.copy()
            self.allocation.update(short_alloc)
            self.allocation = self._remove_zero_positions(self.allocation)

            return self.allocation, long_leftover + short_leftover
```

- First step is to sort the assets in descending order based on their continuous weights (as it processes assets from highest to lowest weight, making it easier to allocate funds to assets with higher weights first). Next step is to check if the lowest weight in the sorted list is negative. If so, it implies that the portfolio includes **short positions**.
- Normalise the weights of longs and shorts so that each of them sums to one.
- The `greedy_portfolio` method is then called recursively on each sub-portfolio to obtain the discrete allocations for the long and short positions (this is now treated using the same 'long method' because the weights have been normalised).

First Round

```
# Otherwise, portfolio is long only and we proceed with greedy algo
        available_funds = self.total_portfolio_value
        shares_bought = []
        buy_prices = []

        #First round
        for ticker, weight in self.weights:
            price = self.latest_prices[ticker]
            #Attempt to buy the lower integer number of shares (can be zero).
            n_shares = int(weight * self.total_portfolio_value / price)
            cost = n_shares * price
            #As weights are all > 0 (long only) we always round down n_shares
            #so the cost is always <= simple weighted share of portfolio value,
            #so we can not run out of funds just here.
            assert cost <= available_funds, "Unexpectedly insufficient funds."
            available_funds -= cost
            shares_bought.append(n_shares)
            buy_prices.append(price)
```

Second Round

```python
# Second round
while available_funds > 0:
    # Calculate the equivalent continuous weights of the shares that
    # have already been bought
    current_weights = np.array(buy_prices) * np.array(shares_bought)
    current_weights /= current_weights.sum()
    ideal_weights = np.array([i[1] for i in self.weights])
    deficit = ideal_weights - current_weights

    # Attempt to buy the asset whose current weights deviate the most
    idx = np.argmax(deficit)
    ticker, weight = self.weights[idx]
    price = self.latest_prices[ticker]

    # If we can't afford this asset, search for the next highest deficit that we
    # can purchase.
    counter = 0
    while price > available_funds:
        deficit[idx] = 0  # we can no longer purchase the asset at idx
        idx = np.argmax(deficit)  # find the next most deviant asset

        # If either of these conditions is met, we break out of both while loops
        # hence the repeated statement below
        if deficit[idx] < 0 or counter == 10:
            break

        ticker, weight = self.weights[idx]
        price = self.latest_prices[ticker]
        counter += 1

    if deficit[idx] <= 0 or counter == 10:  # pragma: no cover
        # Dirty solution to break out of both loops
        break

    # Buy one share at a time
    shares_bought[idx] += 1
    available_funds -= price

self.allocation = self._remove_zero_positions(
    collections.OrderedDict(zip([i[0] for i in self.weights], shares_bought))
)

if verbose:
    print("Funds remaining: {:.2f}".format(available_funds))
    self._allocation_rmse_error(verbose)
return self.allocation, available_funds
```

- As long as there are available funds for further purchases.
- Calculates the current weights of the assets in the portfolio by multiplying the number of shares bought (`shares_bought`) by their corresponding purchase prices (`buy_prices`).
- Normalises the current weights (necessary for comparing them to the ideal weights).
- Calculates the deficit, which represents the deviation of the current weights from the ideal weights.