! This class has been made inactive. No posts will be allowed until an instructor reactivates the class.

note 172 views

Sharing "StrategyLearner" from mc3-p3

Professor mentioned that is now ok to share all the way to MC3-P3, if you would like to share your solution for MC3-P3 please post here.

NB: Professor mentioned that he was ok with code being shared here (as copy&paste here and not to link to it from an external website).

Thanks in advance for anyone sharing!

mc3-p3

Updated 1 year ago by Carlos Aguayo

followup discussions *for lingering questions and comments*

Resolved Unresolved

Carlos Aguayo 1 year ago

Here's my solution:

```
"""
Template for implementing StrategyLearner  (c) 2016 Tucker Balch

StrategyLearner
Carlos Aguayo
carlos.aguayo@gatech.edu
gtid 903055858

"""

import datetime as dt
import QLearner as ql
import pandas as pd
import util as ut
import numpy as np

import warnings  # http://stackoverflow.com/questions/15777951/how-to-suppress-pandas-future-warning
warnings.simplefilter(action="ignore", category=FutureWarning)


class StrategyLearner(object):

    # constructor
    def __init__(self, verbose = False):
        self.verbose = verbose
        self.learner = None

    @staticmethod
    def bollinger_value(price, sma, stddev):
        # bb_value[t] = (price[t] - SMA[t])/(2 * stdev[t])
        return (price - sma) / (2 * stddev)

    @staticmethod
    def momentum(price, n):
        # momentum[t] = (price[t]/price[t-N]) - 1
        return price / price.shift(n) - 1

    @staticmethod
    def volatility(price, n):
        # Volatility is just the stdev of daily returns.

        def compute_daily_returns(s):
            # https://www.udacity.com/course/viewer#!/c-ud501/l-4156938722/e-4185858982/m-4185858984
            daily_returns = s.copy()
            daily_returns[1:] = (s[1:] * 1.0 / s[:-1].values) - 1
            return daily_returns[1:]

        return pd.rolling_std(compute_daily_returns(price), n)

    @staticmethod
    def get_bins(data, steps):
        data = data.copy()
        steps -= 1

        stepsize = len(data) / steps
        data.sort()
        threshold = [0] * steps
        for i in range(steps):
            threshold[i] = data[(i + 1) * stepsize]
        return threshold
```

```python
    @staticmethod
    def to_technical_features(prices, symbol):
        # Bollinger Bands, Momentum, Volatility
        time_window = 20

        sma = pd.rolling_mean(prices, time_window)
        stddev = pd.rolling_std(prices, time_window)

        data = np.ndarray([len(prices), 3])

        data[:, 0] = StrategyLearner.bollinger_value(price=prices, sma=sma, stddev=stddev)[symbol]
        data[:, 1] = StrategyLearner.momentum(price=prices, n=5)[symbol]
        data[1:, 2] = StrategyLearner.volatility(price=prices, n=time_window)[symbol]
        data[:] = np.nan_to_num(data)

        return data

    def build_state(self, array_technical_features, holding):
        state = "".join(map(lambda discrete_value: str(int(discrete_value)), array_technical_features))
        return int(state + str(holding))

    # this method should create a QLearner, and train it for trading
    def addEvidence(self,
                    symbol="IBM",
                    sd=dt.datetime(2008, 1, 1),
                    ed=dt.datetime(2009, 1, 1),
                    sv=10000):

        num_states = 10**4  # 3 features with 9 values, times 3 because buy/sell/hold

        # https://piazza.com/class/ij9yiif53l27fs?cid=1640
        self.learner = ql.QLearner(num_states=num_states,
                                   num_actions=3,  # 3 actions, buy, sell or hold
                                   alpha=0.2,
                                   gamma=0.9,
                                   rar=0.98,
                                   radr=0.999,
                                   dyna=0,
                                   verbose=False)

        # example usage of the old backward compatible util function
        syms=[symbol]
        dates = pd.date_range(sd, ed)
        df = ut.get_data(syms, dates)  # automatically adds SPY
        prices = df[syms]  # only portfolio symbols

        # example use with new colname
        # volume_all = ut.get_data(syms, dates, colname="Volume")  # automatically adds SPY
        # volume = volume_all[syms]  # only portfolio symbols
        # volume_SPY = volume_all['SPY']  # only SPY, for comparison later
        # if self.verbose: print volume

        data = self.to_technical_features(prices, symbol)

        # TODO - Can't I do in one line?
        # TODO - I'd need to store the bins right?
        for i in range(data.shape[1]):
            data[:, i] = np.digitize(x=data[:, i], bins=self.get_bins(data[:, i], 10))

        round_lot = 100
        max_iterations = 100

        cumulative_return = np.ndarray(max_iterations)

        for iteration in range(0, max_iterations):
            # holding - 0
            # buy - 1
            # sell - 2
            df['cash'] = 0
            df['portfolio_value'] = 0
            df['cash'].ix[0] = sv
            df['portfolio_value'].ix[0] = sv

            initial_state = self.build_state(data[0], 0)
            action = self.learner.querysetstate(initial_state)
            prev_date = df.index[0]

            i = 1
            position = 0
            for date in df.index[1:]:
                entry = data[i]
                i += 1

                today_price = prices.ix[date, symbol]

                invalid = False

                if action == 1:  # buy
                    position += 1
                    if position > 1:
```

```
                position = 1
                invalid = True
        elif action == 2:  # sell
            position -= 1
            if position < -1:
                position = -1
                invalid = True

        if invalid:
            action = 0  # TODO: Can I use something like an enum instead of 0, 1 and 2?

        if action == 0:  # hold
            df.ix[date, 'cash'] = df.ix[prev_date, 'cash']
        elif action == 1:  # buy
            df.ix[date, 'cash'] = df.ix[prev_date, 'cash'] - today_price * round_lot
```

Alex K 1 year ago Thanks for sharing.

Resolved Unresolved

Andrew 11 months ago
I was hoping to get confirmation on the discrepancy in the feedback before posting this. However, with the class over, I get the sense that everyone is moving on, which is understandable. I have resolved what I believe was the source to my own satisfaction, so I'll go ahead and post it anyways, as a few have asked for it.

A few caveats, of course. As I've mentioned elsewhere, I'm not a "Python guy", so I'm sure there are better ways of doing some of this. I wouldn't even claim that this is the best way to do the project. It's also more than a bit messy - pieces of this grew organically over the last few projects and I didn't bother to clean things up much once I got it working well enough. The Indicators class in particular needs a rewrite, but you can at least see a number of the features I tried at one time or another on past projects. Of course, if I had to write it all over again I'd probably do everything differently anyways.

I've gone back and sprinkled some comments throughout. If anything isn't clear, feel free to ask questions.

```
"""
    StrategyLearner: Uses Q-Learning to learning trading policy.
"""

import datetime as dt
import math

import numpy as np
import pandas as pd

import QLearner as ql
from util import get_data


class StrategyLearner(object):

    def __init__(self, verbose=False):
        self._verbose = verbose
        self._q_learner = None

        # indicator features to use
        self._features = ["momentum", "momentum_10", "volume", "volatility", "10_day", "20_day", "5_day_spy"]
        # extra fields to pull from data (besides the defaults)
        self._fields = ['High', 'Low', 'Close', 'Volume']
        # support transactions with 100 or 200 shares
        self._shares = [100, 100, 200, 200, None]
        # actions: buy/sell 100 shares, buy/sell 200 shares, hold
        self._actions = ["BUY", "SELL", "BUY", "SELL", "HOLD"]

        self._indicators = None
        self._portfolio = None

    def addEvidence(self, symbol="IBM", sd=dt.datetime(2007, 12, 31), ed=dt.datetime(2009, 12, 31), sv=10000):

        # get data for our states (training period):
        # states - chronological list of indicator state-strings
        # state_map - map of state-strings to states
        # dates - dates for these states
        # price_dict - full price data as dictionary
        (states, state_map, dates, price_dict) = self._get_states_data(symbol, sd, ed)

        # count the total number of states
        num_states = len(state_map.keys())

        self._print_debug("Number of reachable states (train): " + str(num_states))

        # set up the Q-Learner
        self._q_learner = ql.QLearner(num_states, len(self._actions), rar=0.98, radr=0.999)

        # variables to track stopping criteria
        cr_last = None
        last_change = 0
        iterations = 0
```

```
        # continue to perform Q-learning iterations until stopping criteria for convergence
        while True:
            iterations += 1
            # perform trades and get the current cumulative returns
            cr = self._perform_trades(price_dict, symbol, sd, sv, dates, states, state_map)

            self._print_debug("Cumulative return at iteration " + str(iterations) + ": " + str(cr))

            # if the CR has changed, update the last stored value
            if not cr_last or (math.fabs(cr_last - cr) >= 0.001):
                cr_last = cr
                last_change = iterations

            # check the stopping criteria (CR hasn't changed for at least 10 iterations)
            if iterations - last_change > 10:
                break

        self._print_debug("Final training results: " + str(cr_last) + " after " + str(iterations) + " iterations")

    def testPolicy(self, symbol="IBM", sd=dt.datetime(2009, 12, 31), ed=dt.datetime(2011, 12, 31), sv=10000):

        # get data for our states (test period):
        # states - chronological list of indicator state-strings
        # state_map - map of state-strings to states
        # dates - dates for these states
        # price_dict - full price data as dictionary
        (states, state_map, dates, price_dict) = self._get_states_data(symbol, sd, ed, self._indicators)

        # count the total number of states
        num_states = len(state_map.keys())

        self._print_debug("Number of reachable states (test): " + str(num_states))

        # perform test trades for this period
        cr = self._perform_trades(price_dict, symbol, sd, sv, dates, states, state_map, True)

        self._print_debug("Cumulative return: " + str(cr))

        # return the generated trades for this period
        return self._portfolio.trades

    def _get_states_data(self, symbol, sd, ed, indicators=None):
        # get data for symbol
        dates = pd.date_range(sd, ed)
        # get price data, including SPY
        data_features = [get_data([symbol, "SPY"], dates, False)]
        # get data for the requested symbol and SPY
        columns = [symbol, "SPY"]
        # get data for each of the fields we need for the required features
        for field in self._fields:
            # columns for each field are in the form <field_name>_symbol
            data_features.append(get_data([symbol], dates, False, field)[symbol])
            columns.append(symbol + "_" + field)
            data_features.append(get_data(["SPY"], dates, False, field)["SPY"])
            columns.append("SPY_" + field)

        # combine the data into a single DataFrame
        data = pd.concat(data_features, axis=1)
        data.columns = columns
        # drop data for any date that SPY didn't trade
        data = data.dropna(subset=["SPY"])

        # create dictionary for prices (for performance reasons)
        price_dict = {}
        for row in data[symbol].iteritems():
            price_dict[row[0]] = row[1]

        # check if we've already created our indicators
        if indicators is None:
            self._indicators = Indicators(self._features)
            indicators = self._indicators

        # extract the indicators for this data set
        indicators.select(data, symbol)
        # discretize our features
        (dates, states, total_state_size) = indicators.discretize()

        # create a dictionary to store all the reachable states (for performance reasons)
        # probably not practical for a real-world system, but sufficient for this project
        state_map = {}

        # in addition to the features, there will be two other contributions to the state:
        # - two possible CR states (zero and below or greater than zero)
        # - three possible holding positions (NOT HOLDING, HOLDING LONG, HOLDING SHORT)
        # create these "sub-states" to add to our existing state data
        sub_states = []
        for i in range(2):
            for j in range(3):
                sub_states.append(str(j) + str(i))
```

```
        # combine our substates with our feature data to pre-determine all reachable states
        index = 0
        for state in states:
            for sub_state in sub_states:
                full_state = state + sub_state
                if full_state not in state_map.keys():
                    state_map[state + sub_state] = index
                    index += 1

        # this is the total state size (not the reachable state size)
        self._print_debug("Total state size: " + str(total_state_size * 6))

        return states, state_map, dates, price_dict

    def _perform_trades(self, prices, symbol, sd, sv, dates, states, state_map, query_only=False):
        # set-up the portfolio
        self._portfolio = Portfolio(prices, symbol, sd, sv)

        # check if this is the first pass through
        started = False
        # number of shares we're currently holding
        share_holding = 0
        # current CR
        cr = 0.0
        # check if we should impose a penalty
        penalize = False

        for (date, state_str) in zip(dates, states):

            # get the reward for the current date (will be 0.0 to start with, and ignored on the first iteration)
            (r, cr) = self._portfolio.get_reward(date)

            # determine our current CR state
            if cr <= 0.0:
                cr_state = "0"
            else:
                cr_state = "1"

            # determine our current holding position
            if share_holding == 0:
                position = "0"
            elif share_holding > 0:
                position = "1"
            else:
                position = "2"

            # add position and cr states to get the full state-string
            state_str += (position + cr_state)

            # get the mapped state
            state = state_map[state_str]

            # we may loop the current state multiple times if the Q-Learner continues to return an invalid action
            while True:
                # if this is the first day, or we're using the learned policy only (not updating), call the appropriate
                # method
                if not started or query_only:
                    # ignore states we don't know about - just hold (may occur for periods outside the training period)
                    if state >= self._q_learner.num_states:
                        self._print_debug("Skipping state: " + str(state))
                        # set the action to holding
                        action = 4
                    else:
                        # query the Q-Learner only (don't update)
                        action = self._q_learner.querysetstate(state)
                    started = True

                    if query_only:
                        # if we've selecting an invalid action, note it (will be ignored in portfolio)
                        if (action == 0) and (share_holding > 0):
                            self._print_debug("
```

Andrew 11 months ago There seems to be a limit to how much you can post in a single message. Adding the remaining code in follow-up post(s).

```
                    if query_only:
                        # if we've selecting an invalid action, note it (will be ignored in portfolio)
                        if (action == 0) and (share_holding > 0):
                            self._print_debug("Buy 100 when holding long")
                        elif (action == 1) and (share_holding < 0):
                            self._print_debug("Short 100 when holding short")
                        elif (action == 2) and (share_holding >= 0):
                            self._print_debug("Buy 200 when holding none or long")
                        elif (action == 3) and (share_holding <= 0):
                            self._print_debug("Short 200 when holding none short")

                else:
                    # if the Q-Learner selected an invalid action, penalize it
                    if penalize:
```

```
                    r = -100
                # update the Q-Learner with the current state and reward, then get the next action
                action = self._q_learner.query(state, r)

            # when training the learner, penalize it if it attempts to perform an invalid action
            if not query_only:
                if (action == 0) and (share_holding > 0):
                    penalize = True
                elif (action == 1) and (share_holding < 0):
                    penalize = True
                elif (action == 2) and (share_holding >= 0):
                    penalize = True
                elif (action == 3) and (share_holding <= 0):
                    penalize = True
                else:
                    penalize = False

            # if there's no penalty, we can continue
            if not penalize:
                break

        # update the portfolio and get the current share count
        share_holding = self._portfolio.update(date, self._actions[action], symbol, self._shares[action])

    # after processing all state data, return the final CR
    return cr

# simple debug method
def _print_debug(self, message):
    if self._verbose:
        print message


# copy of indicators class from previous project
class Indicators(object):

    # constructor
    def __init__(self, features):
        self._features = list(features)
        self._selected_indicators = None
        self._normalizers = None
        self._thresholds = None

    def select(self, data, symbol):

        # extract the specified features from the data set

        norms = []
        selected = []
        if not self._features or "stoch_rsi" in self._features:
            # stochastic RSI
            stoch_rsi, normalizer = Indicators._get_stochastic_rsi(data, symbol)
            selected.append(stoch_rsi)
            norms.append(normalizer)

        if not self._features or "williams_r" in self._features:
            # williams %r
            williams_r, normalizer = Indicators._get_williams_r(data, symbol)
            selected.append(williams_r)
            norms.append(normalizer)

        if not self._features or "aroon" in self._features:
            # normalized aroon oscillator (25 days)
            aroon, normalizer = Indicators._get_aroon(data, symbol)
            selected.append(aroon)
            norms.append(normalizer)

        if not self._features or "momentum" in self._features:
            # momentum (5 day)
            momentum, normalizer = \
                Indicators._get_momentum(data, symbol, 5, self._features.index("momentum"), self._normalizers)
            selected.append(momentum)
            norms.append(normalizer)

        if not self._features or "momentum_10" in self._features:
            # momentum (10 day)
            momentum, normalizer = Indicators._get_momentum(data, symbol, 10, self._features.index("momentum_10"),
                                                            self._normalizers)
            selected.append(momentum)
            norms.append(normalizer)

        if not self._features or "momentum_20" in self._features:
            # momentum (20 day)
            momentum, normalizer = Indicators._get_momentum(data, symbol, 20, self._features.index("momentum_20"),
                                                            self._normalizers)
            selected.append(momentum)
            norms.append(normalizer)

        if not self._features or "volume" in self._features:
            # volume
            volume, normalizer = Indicators._get_volume(data, symbol, self._features.index("volume"), self._normalizers)
```

```
            selected.append(volume)
            norms.append(normalizer)

        if not self._features or "volatility" in self._features:
            # volatility
            volatility, normalizer = \
                Indicators._get_volatility(data, symbol, self._features.index("volatility"), self._normalizers)
            selected.append(volatility)
            norms.append(normalizer)

        if not self._features or "5_day" in self._features:
            # 5_day ewma
            five_day, normalizer = Indicators._get_normalized_ema(data, symbol, 5)
            selected.append(five_day)
            norms.append(normalizer)

        if not self._features or "10_day" in self._features:
            # 10_day ewma
            ten_day, normalizer = Indicators._get_normalized_ema(data, symbol, 10)
            selected.append(ten_day)
            norms.append(normalizer)

        if not self._features or "20_day" in self._features:
            # 20_day ewma
            twenty_day, normalizer = Indicators._get_normalized_ema(data, symbol, 20)
            selected.append(twenty_day)
            norms.append(normalizer)

        if not self._features or "5_day_spy" in self._features:
            # 5_day_spy ewma
            twenty_day_spy, normalizer = Indicators._get_normalized_ema(data, "SPY", 5)
            selected.append(twenty_day_spy)
            norms.append(normalizer)

        if not self._features or "10_day_spy" in self._features:
            # 10_day_spy ewma
            twenty_day_spy, normalizer = Indicators._get_normalized_ema(data, "SPY", 10)
            selected.append(twenty_day_spy)
            norms.append(normalizer)

        if not self._features or "20_day_spy" in self._features:
            # 20_day_spy ewma
            twenty_day_spy, normalizer = Indicators._get_normalized_ema(data, "SPY", 20)
            selected.append(twenty_day_spy)
            norms.append(normalizer)

        indicators = pd.concat(selected, axis=1).dropna()
        indicators.columns = self._features

        self._selected_indicators = indicators
        self._normalizers = norms

    @property
    def selected_indicators(self):
        return self._selected_indicators

    @property
    def normalizers(self):
        return self._normalizers

    @property
    def features(self):
        return self._features

    def discretize(self):
        # to maintain consistency for all runs, make sure we use the same thresholds
        if self._thresholds is None:
            # no thresholds - find them
            self.find_thresholds()

        binned = []
        total_state_size = 1
        for feature in self._features:
            # get thresholds for the current feature
            thresholds = self._thresholds.get(feature)
            # we have one less bin then number of thresholds
            num_bins = len(thresholds) - 1
            # hack to deal with lack of volume data for ML4T-220
            if num_bins == 0:
                num_bins = 1

            # increment the total state size
            total_state_size *= num_bins

            # ensure all features are within the outer bounds
            self._selected_indicators[feature][self._selected_indicators[feature] < thresholds[0]] = thresholds[0]
            self._selected_indicators[feature][self._selected_indicators[feature] > thresholds[len(thresholds) - 1]] = \
                thresholds[len(thresholds) - 1]

            # just label our bins with the bin number
            labels = [i for i in range(num_bins)]
```

```
            # discretize the features
            (bin_features, feature_thresholds) = pd.cut(self._selected_indicators[feature], bins=thresholds,
                                                        labels=labels, right=True, include_lowest=True, retbins=True)

            binned.append(bin_features)

        # combine discretized features
        binned_ind = pd.concat(binned, axis=1)

        # combine the discretized features into a single state string for each day
        states = []
        for row in binned_ind.itertuples():
            state_str = ""
            for i in range(1, len(self._features) + 1):
                state_str += str(row[i])
            states.append(state_str)

        # return the dates, states and the total state size
        return binned_ind.index, states, total_state_size

    def find_thresholds(self):
        # set-up the threshold dictionary for each of the features
        self._thresholds = {}
        for feature in self._features:
            # find the number of unique values for the given feature
            num_unique = len(self._selected_indicators[feature].unique())
            # just use simple log of the number of unique values to determine the number of bins to use
            num_bins = int(math.log(num_unique, 2))
            # hack to deal with missing volume field for ML4T-220
            if num_bins == 0:
                num_bins = 1

            # just label our bins with the bin number
            labels = [i for i in range(num_bins)]

            # discretize the features using the number of bins (equal-width bins)
            (bin_features, feature_thresholds) = pd.cut(self._selected_indicators[feature]
```

Andrew 11 months ago And again...

```
            # discretize the features using the number of bins (equal-width bins)
            (bin_features, feature_thresholds) = pd.cut(self._selected_indicators[feature], bins=num_bins,
                                                        labels=labels, right=True, include_lowest=True, retbins=True)

            # set the thresholds for this feature
            self._thresholds[feature] = feature_thresholds

    @staticmethod
    def _get_stochastic_rsi(data, symbol):
        # calculate RSI from gains/losses
        gain_or_loss = data.diff()[symbol]
        gains, losses = gain_or_loss.copy(), gain_or_loss.copy()

        # calculate average gains over the window
        gains[gain_or_loss < 0] = 0
        avg_gain = pd.rolling_mean(gains, window=14)

        # calculate average losses over the window
        losses[gain_or_loss > 0] = 0
        # use absolute value of losses
        losses = losses.abs()
        avg_loss = pd.rolling_mean(losses, window=14)

        rs = (avg_gain / avg_loss)
        rsi = 100.0 - (100.0 / (1.0 + rs))

        return pd.rolling_apply(rsi, window=14, func=Indicators._calculate_stoch_rsi), None

    @staticmethod
    def _calculate_stoch_rsi(x):
        return (x[len(x) - 1] - x.min()) / (x.max() - x.min()) - 0.5

    @staticmethod
    def _get_momentum(data, symbol, window, index, normalizers):
        momentum = data[symbol]/data[symbol].shift(window)
        return Indicators._normalize_data(momentum, index, normalizers)

    @staticmethod
    def _get_aroon(data, symbol):
        aroon_up = pd.rolling_apply(data[symbol], window=25, func=Indicators._calculate_arron_up)
        aroon_down = pd.rolling_apply(data[symbol], window=25, func=Indicators._calculate_arron_down)

        return aroon_up - aroon_down - 0.5, None

    @staticmethod
    def _calculate_arron_up(x):
        return (25 - np.argmax(x))/25.0
```

```
    @staticmethod
    def _calculate_arron_down(x):
        return (25 - np.argmin(x))/25.0

    @staticmethod
    def _get_williams_r(data, symbol):
        index = np.array(range(len(data.index)))
        williams_r = pd.rolling_apply(index, window=14,
                                      func=lambda x: Indicators._calculate_williams_r(x, data, symbol))
        cur = pd.DataFrame(data=williams_r, index=data.index)
        return cur, None

    @staticmethod
    def _calculate_williams_r(x, data, symbol):
        cur_loc = int(x[len(x) - 1])
        cur_close = data.iloc[cur_loc][symbol + "_Close"]

        highest_high = data.iloc[x][symbol + "_High"].max()
        lowest_low = data.iloc[x][symbol + "_Low"].min()
        return (highest_high - cur_close) / (highest_high - lowest_low) - 0.5

    @staticmethod
    def _get_volume(data, symbol, index, normalizers):
        if data[symbol + "_Volume"].ix[1] == 1:
            return data[symbol + "_Volume"], None
        return Indicators._normalize_data(data[symbol + "_Volume"], index, normalizers)

    @staticmethod
    def _normalize_data(data, index, normalizers):
        normalizer = Normalizer(data.mean(), data.std())
        if normalizers is not None:
            normalizer = normalizers[index]

        return normalizer.normalize(data), normalizer

    @staticmethod
    def _get_volatility(data, symbol, index, normalizers):
        return Indicators._normalize_data(pd.rolling_std(data[symbol], window=10), index, normalizers)

    @staticmethod
    def _get_normalized_ema(data, symbol, window):
        sma = pd.ewma(data[symbol], span=window)
        std = pd.rolling_std(data[symbol], window=window)
        return (data[symbol] - sma) / (2 * std), None


class Normalizer(object):

    # constructor
    def __init__(self, mean, std):
        self._mean = mean
        self._std = std

    def normalize(self, values):
        return (values - self._mean) / (2 * self._std)


class Portfolio(object):

    def __init__(self, prices, symbol, start_date, sv=10000):
        self._prices = prices
        self._start_date = start_date
        self._sv = sv
        self._symbols = [symbol]

        # min/max shares allowed for any given day
        self._max_shares = 100
        self._min_shares = -100

        # starting positions
        self._current_positions = {"CASH": self._sv, symbol: 0}

        # track trades in a dictionary
        self._trades = {}

        # track the last CR calculated
        self._last_cr = sv

    def update(self, date, action, symbol, shares):

        # if the action is hold, do nothing

        # if the action is buy, update the portfolio
        if action == "BUY":
            # make sure it's a legal action
            if (self._current_positions[symbol] + shares) <= self._max_shares:
                # update number of shares
                self._current_positions[symbol] += shares
                # update current cash
                self._current_positions["CASH"] = self._current_positions["CASH"] - shares * self._prices[date]
```

```python
                # update trades dictionary
                self._trades[date] = shares

        # if the action is sell, update the portfolio
        elif action == "SELL":
            # make sure it's a legal action
            if (self._current_positions[symbol] - shares) >= self._min_shares:
                # update number of shares
                self._current_positions[symbol] -= shares
                # update current cash
                self._current_positions["CASH"] = self._current_positions["CASH"] + shares * self._prices[date]
                # update trades dictionary
                self._trades[date] = -shares

        # return the current asset position
        return self._current_positions[symbol]

    def get_reward(self, date):
        # get the reward for this date based on daily returns
        # first, simply calculate the current portfolio value
        current_totals = 0
        for entry in self._current_positions.keys():
            if entry == "CASH":
                current_totals += self._current_positions[entry]
            else:
                current_totals += self._current_positions[entry] * self._prices[date]

        # daily returns is based on the relationship of the current portfolio value and yesterdays portfolio value
        daily_rets = (current_totals / self._last_cr) - 1

        # cr is based on the current portfolio value and the starting value
        cr = (current_totals / self._sv) - 1

        # update the last CR
        self._last_cr = current_totals

        # return the reward and the CR
        return daily_rets, cr

    @property
    def trades(self):

        # generate a trades DataFrame from the stored trades dictionary
        index = np.array(sorted(self._prices.keys()))

        trades_df = pd.DataFrame(index=index, columns=self._symbols)
        trades_df = trades_df.fillna(value=0.)
        for key in self._trades.keys():
            # we're only training/testing on one symbol
            trades_df[self._symbols[0]][key] = self._trades[key]

        return trades_df
```

Andrew 11 months ago I tried to start the each post slightly overlapping the previous...hopefully it's not too hard to follow.