AI Alpaca Trading Bot

Introduction

This notebook demonstrates the process of creating an ensemble trading strategy and testing it on the Dow Jones 30 index. The ensemble is composed of three Deep Reinforcement Learning (DRL) algorithms - Advantage Actor-Critic (A2C), Proximal Policy Optimization (PPO), and Deep Deterministic Policy Gradient (DDPG). The code used in this notebook is based on the FinRL-Library which is a Python library for financial reinforcement learning developed by AI4Finance-LLC.

Install Required Packages

We begin by installing the packages required to run this notebook. These packages are:

- `setuptools==64.0.2` : A package for downloading and installing Python packages.
- `swig` : A package required by `wrds` package.
- `wrds` : A package for downloading data from the Wharton Research Data Services.
- `git+https://github.com/AI4Finance-LLC/FinRL-Library.git` : The FinRL-Library package.

In [ ]:
```
!pip3 install setuptools==64.0.2
!apt-get install swig
!pip3 install wrds
!pip3 install git+https://github.com/AI4Finance-LLC/FinRL-Library.git
```

Importing Libraries

The first line of the script imports the warnings module, which provides a way to handle warnings that may be encountered during the execution of the script. The second line of the script filters out warnings to avoid clutter in the output.

The next lines of the script import the following libraries:

- `pandas` (`pd`) and `numpy` (`np`) for data analysis and manipulation.
- `matplotlib` for creating visualizations of the data.
- `datetime` for handling date and time information.

Importing Required Modules

The following modules are then imported:

- `DOW_30_TICKER` from `finrl.config_tickers` to specify a list of tickers for the Dow Jones Industrial Average.
- `YahooDownloader` from `finrl.meta.preprocessor.yahoodownloader` to download financial data from Yahoo Finance.
- `FeatureEngineer` and `data_split` from `finrl.meta.preprocessor.preprocessors` for data pre-processing.
- `StockTradingEnv` from `finrl.meta.env_stock_trading.env_stocktrading` to define a custom environment for stock trading.
- `DRLAgent` and `DRLEnsembleAgent` from `finrl.agents.stablebaselines3.models` for reinforcement learning agents.
- `backtest_stats`, `backtest_plot`, `get_daily_return`, and `get_baseline` from `finrl.plot` for creating plots and calculating performance metrics.
- `pprint` for pretty-printing objects.

Setting Configuration Variables

The last few lines of the script set configuration variables for data pre-processing, model training, and testing. These include:

- `sys.path.append("../FinRL-Library")` to add the FinRL-Library directory to the system path.
- `check_and_make_directories` from `finrl.main` to create directories for data storage, model training, and testing results.
- `DATA_SAVE_DIR`, `TRAINED_MODEL_DIR`, `TENSORBOARD_LOG_DIR`, and `RESULTS_DIR` for specifying the paths to the data storage, model training, and testing results directories.
- `INDICATORS` to specify a list of technical indicators to be used in feature engineering.
- `TRAIN_START_DATE`, `TRAIN_END_DATE`, `TEST_START_DATE`, `TEST_END_DATE`, `TRADE_START_DATE`, and `TRADE_END_DATE` to specify the start and end dates for training, testing, and trading periods.

In [ ]:
```
import warnings
warnings.filterwarnings("ignore")
import pandas as pd
import numpy as np
import matplotlib
import matplotlib.pyplot as plt
%matplotlib inline
matplotlib.use('Agg')
import datetime

from finrl.config_tickers import DOW_30_TICKER
from finrl.meta.preprocessor.yahoodownloader import YahooDownloader
from finrl.meta.preprocessor.preprocessors import FeatureEngineer, data_spli
```

```
from finrl.meta.env_stock_trading.env_stocktrading import StockTradingEnv
from finrl.agents.stablebaselines3.models import DRLAgent,DRLEnsembleAgent
from finrl.plot import backtest_stats, backtest_plot, get_daily_return, get_

from pprint import pprint

import sys
sys.path.append("../FinRL-Library")

import itertools

import os
from finrl.main import check_and_make_directories
from finrl.config import (
    DATA_SAVE_DIR,
    TRAINED_MODEL_DIR,
    TENSORBOARD_LOG_DIR,
    RESULTS_DIR,
    INDICATORS,
    TRAIN_START_DATE,
    TRAIN_END_DATE,
    TEST_START_DATE,
    TEST_END_DATE,
    TRADE_START_DATE,
    TRADE_END_DATE,
)

check_and_make_directories([DATA_SAVE_DIR, TRAINED_MODEL_DIR, TENSORBOARD_LO
```

```
2023-04-19 21:52:24.986548: I tensorflow/core/util/port.cc:110] oneDNN custom
operations are on. You may see slightly different numerical results due to fl
oating-point round-off errors from different computation orders. To turn them
off, set the environment variable `TF_ENABLE_ONEDNN_OPTS=0`.
2023-04-19 21:52:25.022337: I tensorflow/core/platform/cpu_feature_guard.cc:1
82] This TensorFlow binary is optimized to use available CPU instructions in
performance-critical operations.
To enable the following instructions: AVX2 AVX512F AVX512_VNNI FMA, in other
operations, rebuild TensorFlow with the appropriate compiler flags.
2023-04-19 21:52:25.677180: W tensorflow/compiler/tf2tensorrt/utils/py_utils.
cc:38] TF-TRT Warning: Could not find TensorRT
```

The `DOW_30_TICKER` variable contains a list of 30 stock tickers of companies that are part of the Dow Jones Industrial Average.

The code defines four date variables for training and testing purposes, namely `TRAIN_START_DATE`, `TRAIN_END_DATE`, `TEST_START_DATE`, and `TEST_END_DATE`.

Then, the code creates a DataFrame object `df` using the `YahooDownloader` class from the `finrl` package. The `YahooDownloader` object takes four parameters, namely `start_date`, `end_date`, `ticker_list`, and `fetch_data()`. The `start_date` and `end_date` parameters are set to `TRAIN_START_DATE` and `TEST_END_DATE`, respectively. The `ticker_list` parameter is set to `DOW_30_TICKER`, which is the list of stock tickers imported earlier. The `fetch_data()` method fetches historical stock price data from Yahoo Finance for the specified ticker list and date range.

After creating the df `DataFrame`, the code prints the first five rows of the `DataFrame` using the `head()` method, followed by the last five rows using the `tail()` method, and then the shape of the `DataFrame` using the `shape` attribute.

Next, the code sorts the `df` `DataFrame` by date and ticker using the `sort_values()` method and prints the first five rows of the sorted DataFrame.

The code then prints the number of unique tickers in the `DataFrame` using the `unique()` method applied to the tic column of the `DataFrame`.

Finally, the code prints the count of each ticker in the `DataFrame` using the `value_counts()` method applied to the tic column of the `DataFrame`.

In [ ]:

```
print(DOW_30_TICKER)

TRAIN_START_DATE = '2009-04-01'
TRAIN_END_DATE = '2022-01-01'
TEST_START_DATE = '2022-01-01'
TEST_END_DATE = '2023-04-01'

df = YahooDownloader(start_date = TRAIN_START_DATE,
                     end_date = TEST_END_DATE,
                     ticker_list = DOW_30_TICKER).fetch_data()

df.head()
```

```
['AXP', 'AMGN', 'AAPL', 'BA', 'CAT', 'CSCO', 'CVX', 'GS', 'HD', 'HON', 'IBM',
'INTC', 'JNJ', 'KO', 'JPM', 'MCD', 'MMM', 'MRK', 'MSFT', 'NKE', 'PG', 'TRV',
'UNH', 'CRM', 'VZ', 'V', 'WBA', 'WMT', 'DIS', 'DOW']
[*********************100%***********************]  1 of 1 completed
[*********************100%***********************]  1 of 1 completed
[*********************100%***********************]  1 of 1 completed
[*********************100%***********************]  1 of 1 completed
[*********************100%***********************]  1 of 1 completed
[*********************100%***********************]  1 of 1 completed
[*********************100%***********************]  1 of 1 completed
[*********************100%***********************]  1 of 1 completed
[*********************100%***********************]  1 of 1 completed
[*********************100%***********************]  1 of 1 completed
[*********************100%***********************]  1 of 1 completed
[*********************100%***********************]  1 of 1 completed
[*********************100%***********************]  1 of 1 completed
[*********************100%***********************]  1 of 1 completed
[*********************100%***********************]  1 of 1 completed
[*********************100%***********************]  1 of 1 completed
[*********************100%***********************]  1 of 1 completed
[*********************100%***********************]  1 of 1 completed
[*********************100%***********************]  1 of 1 completed
[*********************100%***********************]  1 of 1 completed
[*********************100%***********************]  1 of 1 completed
[*********************100%***********************]  1 of 1 completed
[*********************100%***********************]  1 of 1 completed
[*********************100%***********************]  1 of 1 completed
[*********************100%***********************]  1 of 1 completed
[*********************100%***********************]  1 of 1 completed
[*********************100%***********************]  1 of 1 completed
[*********************100%***********************]  1 of 1 completed
[*********************100%***********************]  1 of 1 completed
[*********************100%***********************]  1 of 1 completed
Shape of DataFrame:  (103242, 8)
```

Out[ ]:

	date	open	high	low	close	volume	tic	day
0	2009-04-01	3.717500	3.892857	3.710357	3.303859	589372000	AAPL	2
1	2009-04-01	48.779999	48.930000	47.099998	35.911705	10850100	AMGN	2
2	2009-04-01	13.340000	14.640000	13.080000	11.688809	27701800	AXP	2
3	2009-04-01	34.520000	35.599998	34.209999	26.850748	9288800	BA	2
4	2009-04-01	27.500000	29.520000	27.440001	19.726316	15308300	CAT	2

In [ ]:

```
df.tail()
```

Out[ ]:

	date	open	high	low	close	volume	tic	day
103237	2023-03-31	471.519989	476.000000	470.100006	472.589996	3971300	UNH	4
103238	2023-03-31	223.600006	225.839996	223.289993	225.460007	9507200	V	4
103239	2023-03-31	38.790001	39.049999	38.549999	38.256863	22796000	VZ	4
103240	2023-03-31	34.820000	34.840000	34.259998	34.580002	6705800	WBA	4
103241	2023-03-31	146.580002	148.440002	146.470001	147.449997	6954400	WMT	4

In [ ]:

```
df.shape
```

Out[ ]:

```
(103242, 8)
```

In [ ]:

```
df.sort_values(['date','tic']).head()
```

Out[ ]:

	date	open	high	low	close	volume	tic	day
0	2009-04-01	3.717500	3.892857	3.710357	3.303859	589372000	AAPL	2
1	2009-04-01	48.779999	48.930000	47.099998	35.911705	10850100	AMGN	2
2	2009-04-01	13.340000	14.640000	13.080000	11.688809	27701800	AXP	2
3	2009-04-01	34.520000	35.599998	34.209999	26.850748	9288800	BA	2
4	2009-04-01	27.500000	29.520000	27.440001	19.726316	15308300	CAT	2

In [ ]:

```
df.tic.unique()
df.tic.value_counts()
```

Out[ ]:
```
AAPL    3525
AMGN    3525
WMT     3525
WBA     3525
VZ      3525
V       3525
UNH     3525
TRV     3525
PG      3525
NKE     3525
MSFT    3525
MRK     3525
MMM     3525
MCD     3525
KO      3525
JPM     3525
JNJ     3525
INTC    3525
IBM     3525
HON     3525
HD      3525
GS      3525
DIS     3525
CVX     3525
CSCO    3525
CRM     3525
CAT     3525
BA      3525
AXP     3525
DOW     1017
Name: tic, dtype: int64
```

The following code block initializes the INDICATORS list with the names of four technical indicators: `macd`, `rsi_30`, `cci_30`, and `dx_30`.

Next, an instance of the `FeatureEngineer` class is created with the following parameters:

- `use_technical_indicator=True` to specify that technical indicators will be used in feature engineering.
- `tech_indicator_list=INDICATORS` to specify the list of technical indicators to be used.
- `use_turbulence=True` to specify that turbulence index will be used as a feature.
- `user_defined_feature=False` to specify that no additional user-defined features will be used.

The `preprocess_data` method of the `FeatureEngineer` instance is then called with the `df` parameter, which contains financial data in the form of a Pandas `DataFrame`. The resulting preprocessed data is then copied to a new `DataFrame` and missing values are filled with zeros using the `fillna(0)` method. Any infinite values are also replaced with zeros using the `replace(np.inf,0)` method.

The `sample` method is then called on the processed `DataFrame` to display a random sample of five rows of the preprocessed data.

The `stock_dimension` variable is then initialized to the number of unique stock tickers in the processed `DataFrame`, while `state_space` is initialized to a calculated value based on the number of stocks, technical indicators, and other features used. The `print` statement at the end of the script outputs the values of `stock_dimension` and `state_space`.

In [ ]:
```
INDICATORS = ['macd',
               'rsi_30',
               'cci_30',
               'dx_30']

print("==============Preprocessing Data===========")

fe = FeatureEngineer(use_technical_indicator=True,
                     tech_indicator_list = INDICATORS,
                     use_turbulence=True,
                     user_defined_feature = False)

processed = fe.preprocess_data(df)
processed = processed.copy()
processed = processed.fillna(0)
processed = processed.replace(np.inf,0)

print(processed.sample(5))

stock_dimension = len(processed.tic.unique())
state_space = 1 + 2*stock_dimension + len(INDICATORS)*stock_dimension
print(f"Stock Dimension: {stock_dimension}, State Space: {state_space}")
```

```
==============Preprocessing Data===========
Successfully added technical indicators
Successfully added turbulence index
              date        open        high         low       close    volume  \
99965   2022-12-08  155.889999  156.419998  153.449997  153.020752   1632400
93254   2022-01-06   78.790001   79.580002   77.949997   75.847565  11359200
34445   2013-12-17   81.790001   81.849998   80.690002   61.838070  11416400
65179   2018-03-06   43.950001   44.049999   43.590000   37.185398  10010500
101252  2023-02-13   27.870001   28.549999   27.719999   28.549999  32347500

         tic  day      macd     rsi_30      cci_30      dx_30  turbulence
99965    AXP    3  2.000852  52.908038   54.956991  11.740558   10.416006
93254    MRK    3  0.286666  53.312368  122.295699  17.469565   44.801264
34445     PG    1 -0.060503  47.563807 -155.948069  19.552919   73.600199
65179     KO    1 -0.479148  43.946574  -52.719030  17.244998   32.062424
101252  INTC    0  0.040394  50.470899  -32.385753   1.496642    9.711087
Stock Dimension: 29, State Space: 175
```

The `env_kwargs` dictionary contains the configuration of the `StockTradingEnv`. Here are the definitions of the variables in the dictionary:

- `hmax` : The maximum number of shares that can be traded per action.
- `initial_amount` : The amount of cash with which the agent starts trading.
- `buy_cost_pct` : The cost of buying stocks. This is a percentage of the total value of the stocks purchased.
- `sell_cost_pct` : The cost of selling stocks. This is a percentage of the total value of the stocks sold.
- `state_space` : The dimension of the state space of the environment. It is calculated as `1 + 2 * stock_dimension + len(INDICATORS) * stock_dimension` , where `stock_dimension` is the number of unique stock tickers in the dataset and `INDICATORS` is the list of technical indicators used to preprocess the data.
- `stock_dim` : The number of unique stock tickers in the dataset.
- `tech_indicator_list` : The list of technical indicators used to preprocess the data.
- `action_space` : The dimension of the action space of the environment. It is equal to stock_dimension.
- `reward_scaling` : A scaling factor used to normalize the reward.
- `print_verbosity` : The level of verbosity of the environment.

The `rebalance_window` and `validation_window` variables determine the duration of the rebalance and validation windows, respectively. The rebalance window is the number of days after which the model is retrained, while the validation window is the number of days used for validation and trading.

The `DRLEnsembleAgent` object is used to train and evaluate the ensemble trading strategy. It takes in the preprocessed data, training and validation periods, rebalance and validation windows, and environment configuration as input arguments.

The `A2C_model_kwargs` , `PPO_model_kwargs` , and `DDPG_model_kwargs` Dictionaries contain the hyperparameters for the A2C, PPO, and DDPG models, respectively. The hyperparameters include the learning rate, batch size, number of steps, entropy coefficient, and buffer size.

The `timesteps_dict` dictionary contains the number of training steps for each model. The number of steps is set to 1 in this example.

The `df_summary` `DataFrame` contains the summary statistics for the ensemble trading strategy. The statistics include the Sharpe ratio, annual return, maximum drawdown, and total number of trades.

The `df_trade_date` `DataFrame` contains the unique trade dates for the trading period. The `df_account_value` `DataFrame` contains the account value for each trading day, as well as the portfolio value, daily return, and total return. These values are stored in separate CSV files for each rebalance period.

In [ ]:
```
env_kwargs = {
    "hmax": 100,
```

```
    "initial_amount": 1000000,
    "buy_cost_pct": 0.001,
    "sell_cost_pct": 0.001,
    "state_space": state_space,
    "stock_dim": stock_dimension,
    "tech_indicator_list": INDICATORS,
    "action_space": stock_dimension,
    "reward_scaling": 1e-4,
    "print_verbosity":5
}

rebalance_window = 63 #63 # rebalance_window is the number of days to retrai
validation_window = 63 #63 # validation_window is the number of days to do v

ensemble_agent = DRLEnsembleAgent(df=processed,
                 train_period=(TRAIN_START_DATE,TRAIN_END_DATE),
                 val_test_period=(TEST_START_DATE,TEST_END_DATE),
                 rebalance_window=rebalance_window,
                 validation_window=validation_window,
                 **env_kwargs)


A2C_model_kwargs = {
                    'n_steps': 5,
                    'ent_coef': 0.005,
                    'learning_rate': 0.0007
                    }

PPO_model_kwargs = {
                    "ent_coef":0.01,
                    "n_steps": 2, #2048
                    "learning_rate": 0.00025,
                    "batch_size": 128
                    }

DDPG_model_kwargs = {
                      #"action_noise":"ornstein_uhlenbeck",
                      "buffer_size": 1, #10_000
                      "learning_rate": 0.0005,
                      "batch_size": 64
                    }

timesteps_dict = {'a2c' : 1, #10_000 each
                 'ppo' : 1,
                 'ddpg' : 1
                 }
```

The code block performs an ensemble strategy run using an instance of the `DRLEnsembleAgent` class called `ensemble_agent`. This ensemble agent is trained to combine the predictions of multiple Deep Reinforcement Learning (DRL) models for better performance in stock trading.

The `run_ensemble_strategy` method of the `DRLEnsembleAgent` instance is called with the following parameters:

- `A2C_model_kwargs`, `PPO_model_kwargs`, and `DDPG_model_kwargs`: dictionaries containing keyword arguments that will be used to instantiate A2C, PPO, and DDPG models, respectively. These arguments can include hyperparameters such as learning rate, discount factor, number of hidden layers, etc.
- `timesteps_dict`: a dictionary specifying the number of timesteps for training and testing each model. This can be useful for comparing performance of models with different training lengths.

The `run_ensemble_strategy` method executes the ensemble strategy run and returns a summary of the results, which is stored in the `df_summary` DataFrame. This summary includes statistics such as total return, Sharpe ratio, maximum drawdown, and other performance metrics for the ensemble strategy.

In [ ]:

```
df_summary = ensemble_agent.run_ensemble_strategy(A2C_model_kwargs,
                                                  PPO_model_kwargs,
                                                  DDPG_model_kwargs,
                                                  timesteps_dict)

df_summary
```

```
============Start Ensemble Strategy============
============================================
turbulence_threshold:  200.79827641989235
======Model training from:  2009-04-01 to  2022-01-03
======A2C Training========
{'n_steps': 5, 'ent_coef': 0.005, 'learning_rate': 0.0007}
Using cpu device
Logging to tensorboard_log/a2c/a2c_126_5
======A2C Validation from:  2022-01-03 to  2022-04-04
A2C Sharpe Ratio:  -0.10828496901537975
======PPO Training========
{'ent_coef': 0.01, 'n_steps': 2, 'learning_rate': 0.00025, 'batch_size': 128}
Using cpu device
Logging to tensorboard_log/ppo/ppo_126_5
------------------------------------
| time/              |             |
|    fps             | 72          |
|    iterations      | 1           |
|    time_elapsed    | 0           |
|    total_timesteps | 2           |
| train/             |             |
|    reward          | 0.035176605 |
------------------------------------
======PPO Validation from:  2022-01-03 to  2022-04-04
PPO Sharpe Ratio:  -0.08833196158451757
======DDPG Training========
{'buffer_size': 1, 'learning_rate': 0.0005, 'batch_size': 64}
Using cpu device
Logging to tensorboard_log/ddpg/ddpg_126_5
======DDPG Validation from:  2022-01-03 to  2022-04-04
======Best Model Retraining from:  2009-04-01 to  2022-04-04
======Trading from:  2022-04-04 to  2022-07-06
============================================
turbulence_threshold:  200.79827641989235
======Model training from:  2009-04-01 to  2022-04-04
======A2C Training========
{'n_steps': 5, 'ent_coef': 0.005, 'learning_rate': 0.0007}
Using cpu device
Logging to tensorboard_log/a2c/a2c_189_4
======A2C Validation from:  2022-04-04 to  2022-07-06
A2C Sharpe Ratio:  -0.17209157351865084
======PPO Training========
{'ent_coef': 0.01, 'n_steps': 2, 'learning_rate': 0.00025, 'batch_size': 128}
Using cpu device
Logging to tensorboard_log/ppo/ppo_189_4
------------------------------------
| time/              |             |
|    fps             | 86          |
|    iterations      | 1           |
|    time_elapsed    | 0           |
|    total_timesteps | 2           |
| train/             |             |
|    reward          | 0.016925406 |
------------------------------------
======PPO Validation from:  2022-04-04 to  2022-07-06
PPO Sharpe Ratio:  -0.3579357179174672
```

```
======DDPG Training========
{'buffer_size': 1, 'learning_rate': 0.0005, 'batch_size': 64}
Using cpu device
Logging to tensorboard_log/ddpg/ddpg_189_4
======DDPG Validation from:  2022-04-04 to  2022-07-06
======Best Model Retraining from:  2009-04-01 to  2022-07-06
======Trading from:  2022-07-06 to  2022-10-04
============================================
turbulence_threshold:  200.79827641989235
======Model training from:  2009-04-01 to  2022-07-06
======A2C Training========
{'n_steps': 5, 'ent_coef': 0.005, 'learning_rate': 0.0007}
Using cpu device
Logging to tensorboard_log/a2c/a2c_252_3
======A2C Validation from:  2022-07-06 to  2022-10-04
A2C Sharpe Ratio:  -0.22462784195434582
======PPO Training========
{'ent_coef': 0.01, 'n_steps': 2, 'learning_rate': 0.00025, 'batch_size': 128}
Using cpu device
Logging to tensorboard_log/ppo/ppo_252_3
------------------------------------
| time/              |             |
|    fps             | 52          |
|    iterations      | 1           |
|    time_elapsed    | 0           |
|    total_timesteps | 2           |
| train/             |             |
|    reward          | 0.017820721 |
------------------------------------
======PPO Validation from:  2022-07-06 to  2022-10-04
PPO Sharpe Ratio:  -0.3428364503971467
======DDPG Training========
{'buffer_size': 1, 'learning_rate': 0.0005, 'batch_size': 64}
Using cpu device
Logging to tensorboard_log/ddpg/ddpg_252_3
======DDPG Validation from:  2022-07-06 to  2022-10-04
======Best Model Retraining from:  2009-04-01 to  2022-10-04
======Trading from:  2022-10-04 to  2023-01-04
Ensemble Strategy took:  2.8501511295636495  minutes
```

Out[ ]:

	Iter	Val Start	Val End	Model Used	A2C Sharpe	PPO Sharpe	DDPG Sharpe
0	126	2022-01-03	2022-04-04	PPO	-0.108285	-0.088332	-0.119326
1	189	2022-04-04	2022-07-06	A2C	-0.172092	-0.357936	-0.207065
2	252	2022-07-06	2022-10-04	DDPG	-0.224628	-0.342836	-0.102599

This code block performs an analysis of the performance of the trading strategy over a test period. The first step is to identify the unique trading dates within the test period using the `unique_trade_date` variable. This is achieved by filtering the processed DataFrame to include only dates that are greater than `TEST_START_DATE` and less than or equal to `TEST_END_DATE`, and then selecting only the unique dates using the `unique()` method.

The `df_trade_date` DataFrame is then created to store these unique trading dates in a column named `datadate`. An empty DataFrame called `df_account_value` is also initialized to store the account value data from each rebalancing period.

A loop is then executed to read the account value data from the rebalancing periods and concatenate it into `df_account_value`. The loop iterates over each rebalancing period, which has a length of `rebalance_window + validation_window`. The `pd.read_csv()` function reads the CSV file that contains the account value data for the corresponding rebalancing period and saves it to a temporary DataFrame called `temp`. The `df_account_value` DataFrame is then concatenated with `temp` using the `pd.concat()` function to append the data from the current rebalancing period to the overall DataFrame. The `ignore_index=True` parameter ensures that the indices of the original DataFrames are not used in the concatenated DataFrame.

Finally, the Sharpe ratio of the trading strategy is calculated using the formula `sharpe = (252**0.5)*df_account_value.account_value.pct_change(1).mean()/df_accou`
The Sharpe ratio is a measure of risk-adjusted return that is commonly used to evaluate investment strategies. It is calculated as the ratio of the average excess return earned over the risk-free rate per unit of volatility or standard deviation of returns. In this case, the daily returns of the trading strategy are used to calculate the Sharpe ratio. The Sharpe ratio is printed to the console using the `print()` function.

In [ ]:
```
unique_trade_date = processed[(processed.date > TEST_START_DATE)&(processed.

df_trade_date = pd.DataFrame({'datadate':unique_trade_date})
df_account_value = pd.DataFrame()

for i in range(rebalance_window+validation_window, len(unique_trade_date)+1,
    temp = pd.read_csv('results/account_value_trade_{}_{}.csv'.format('ensem
    df_account_value = pd.concat([df_account_value, temp], ignore_index=True

sharpe=(252**0.5)*df_account_value.account_value.pct_change(1).mean()/df_acc

print('Sharpe Ratio: ',sharpe)
```

```
Sharpe Ratio:  0.28436988550196607
```

Following code block aims to plot the account value over time for the rebalancing periods in the `df_account_value` DataFrame. To achieve this, `df_account_value` is joined with `df_trade_date` on the datadate column. The `validation_window` number of rows from the beginning of `df_trade_date` are skipped using the `df_trade_date[validation_window:]` slicing syntax. The `reset_index()` method is called on the sliced DataFrame to reset the index to start from zero, and the `drop=True` parameter is used to drop the original index column.

The resulting DataFrame is stored back in `df_account_value`. This ensures that both DataFrames have the same number of rows, which is required for plotting.

Next, the `head()` method is called on `df_account_value` to display the first few rows of the DataFrame. This provides an overview of the data, including the account value and the corresponding dates for each rebalancing period.

Finally, the `account_value` column of `df_account_value` is selected and plotted using the `plot()` method. This generates a line plot of the account value over time, with the x-axis representing the dates and the y-axis representing the account value. The plot provides a visual representation of the performance of the trading strategy over the rebalancing periods. It can be used to identify trends, patterns, and anomalies in the account value data.

In [ ]:

```
df_account_value=df_account_value.join(df_trade_date[validation_window:].res

df_account_value.head()
```

Out[ ]:

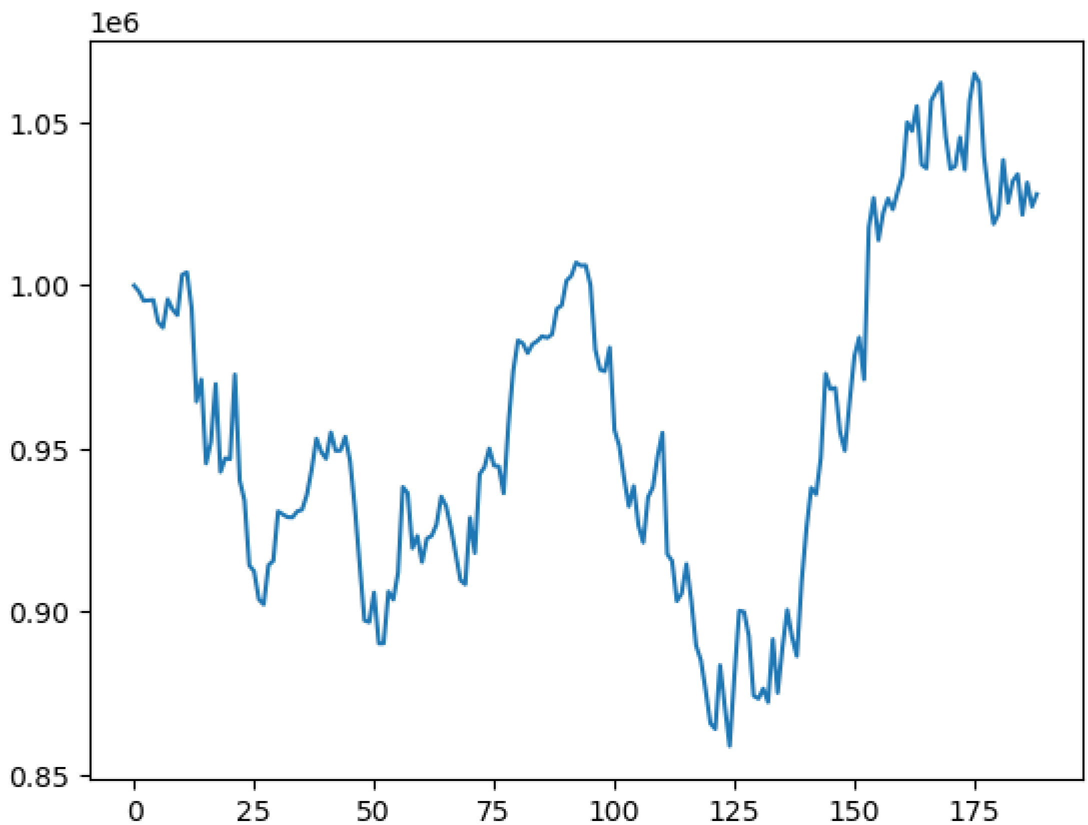
	account_value	date	daily_return	datadate
0	1000000.000000	2022-04-04	NaN	2022-04-04
1	998183.683387	2022-04-05	-0.001816	2022-04-05
2	995306.231272	2022-04-06	-0.002883	2022-04-06
3	995436.087707	2022-04-07	0.000130	2022-04-07
4	995559.217485	2022-04-08	0.000124	2022-04-08

In [ ]:

```
%matplotlib inline
df_account_value.account_value.plot()
```

Out[ ]:

```
<AxesSubplot:>
```



Backtesting is the process of evaluating a trading strategy using historical data to see how it would have performed in the past. It is an essential step in developing and refining trading strategies and can help traders to identify potential risks and opportunities.

The `backtest_stats()` function is called on the `df_account_value` DataFrame to calculate the performance statistics for the trading strategy. This function takes the account value data as input and calculates various performance metrics such as total return, annualized return, Sharpe ratio, and maximum drawdown. The resulting performance statistics are stored in the `perf_stats_all` variable.

The `perf_stats_all` variable is then converted to a pandas DataFrame using the `pd.DataFrame()` function. This converts the performance statistics into a tabular format that is easier to read and analyze.

Finally, the backtest results are printed to the console using the `print()` function. This provides a summary of the performance of the trading strategy, including the various performance metrics calculated by the `backtest_stats()` function. The current date and time are also calculated using the `datetime.datetime.now()` function and the `strftime()` method to format the output.

In [ ]:

```
print("==============Get Backtest Results===========")
now = datetime.datetime.now().strftime('%Y%m%d-%Hh%M')

perf_stats_all = backtest_stats(account_value=df_account_value)
perf_stats_all = pd.DataFrame(perf_stats_all)
```

```
==============Get Backtest Results===========
Annual return          0.037583
Cumulative returns     0.028057
Annual volatility      0.201040
Sharpe ratio           0.284370
Calmar ratio           0.255195
Stability              0.120396
Max drawdown          -0.147272
Omega ratio            1.049088
Sortino ratio          0.416577
Skew                        NaN
Kurtosis                    NaN
Tail ratio             1.148323
Daily value at risk   -0.025102
dtype: float64
```

This code block calculates the performance statistics for a baseline trading strategy and compares it with the performance of the trading strategy used in the previous code block.

The `get_baseline()` function is called to download the historical price data for the Dow Jones Industrial Average (^DJI) index, which is commonly used as a benchmark for the performance of the stock market. This function takes the start and end dates as input and returns a DataFrame containing the historical price data for the specified time period.

The `backtest_stats()` function is then called on the `baseline_df` DataFrame to calculate the performance statistics for the baseline trading strategy. This function takes the price data as input and calculates various performance metrics such as total return, annualized return, Sharpe ratio, and maximum drawdown. The resulting performance statistics are stored in the stats variable.

Comparing the backtest results of the baseline strategy with the performance of the trading strategy used in the previous code block can help to evaluate the effectiveness of the trading strategy relative to the overall market. If the trading strategy outperforms the baseline strategy, it may indicate that the strategy has a significant edge in the market. Conversely, if the trading strategy underperforms the baseline strategy, it may suggest that the strategy needs further optimization or refinement.

In [ ]:
```
print("==============Get Baseline Stats===========")
baseline_df = get_baseline(
        ticker="^DJI",
        start = df_account_value.loc[0,'date'],
        end = df_account_value.loc[len(df_account_value)-1,'date'])

stats = backtest_stats(baseline_df, value_col_name = 'close')
```

```
==============Get Baseline Stats===========
[*********************100%***********************]  1 of 1 completed
Shape of DataFrame:  (188, 8)
Annual return          -0.067521
Cumulative returns     -0.050817
Annual volatility       0.207899
Sharpe ratio           -0.234486
Calmar ratio           -0.368916
Stability               0.002484
Max drawdown           -0.183024
Omega ratio             0.961703
Sortino ratio          -0.327108
Skew                         NaN
Kurtosis                     NaN
Tail ratio              0.927461
Daily value at risk    -0.026386
dtype: float64
```

This code block compares the backtest results obtained from the trading strategy to the performance of the Dow Jones Industrial Average (DJIA) over the same period.

The `backtest_plot` function takes three arguments:

- `df_account_value` : A DataFrame containing the account value over the period of the backtest.
- `baseline_ticker` : A string indicating the ticker symbol for the baseline index. In this case, it is set to '^DJI', which represents the DJIA.
- `baseline_start` and `baseline_end` : Strings representing the start and end dates for the baseline index data. In this case, they are set to the start and end dates of the trading period.

The function plots two lines on the same graph:

- The first line represents the account value of the trading strategy over the backtest period.
- The second line represents the value of the baseline index (DJIA) over the same period.

This allows for a direct comparison of the performance of the trading strategy with that of the benchmark index.

In [ ]:
```
print("==============Compare to DJIA===========")
# S&P 500: ^GSPC
# Dow Jones Index: ^DJI
# NASDAQ 100: ^NDX
backtest_plot(df_account_value,
              baseline_ticker = '^DJI',
              baseline_start = df_account_value.loc[0,'date'],
              baseline_end = df_account_value.loc[len(df_account_value)-1,'c
```

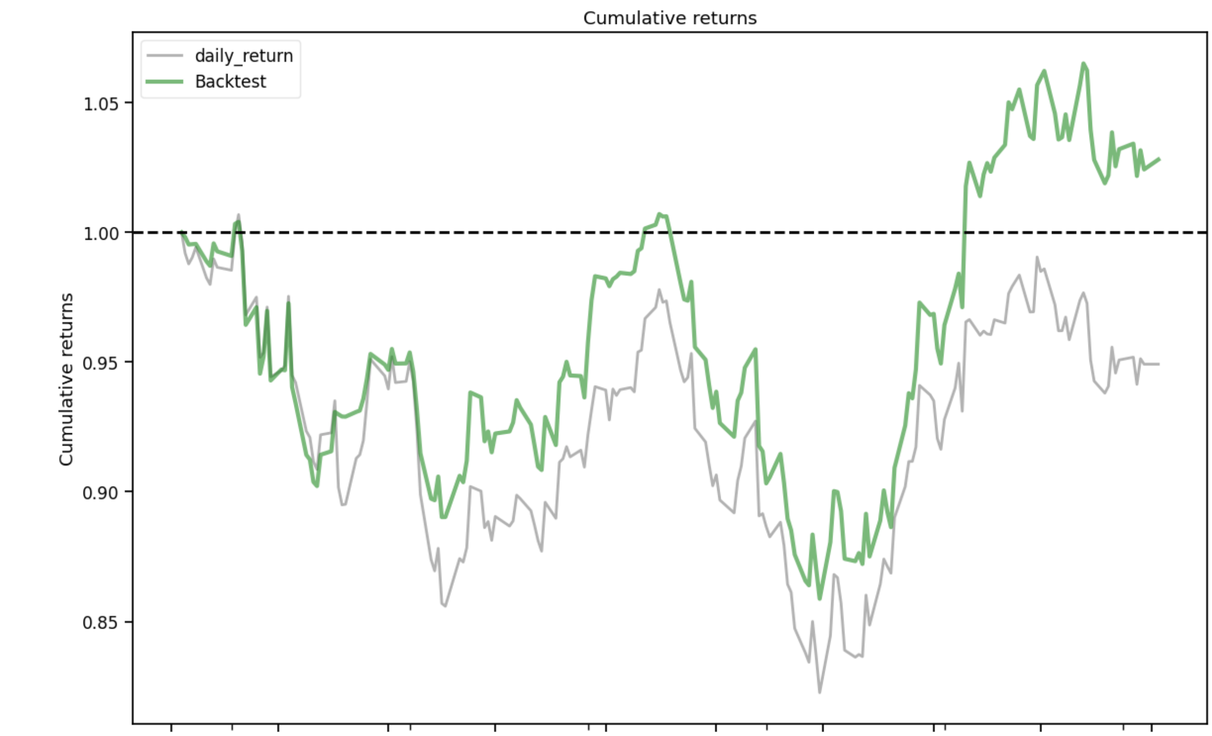
```
==============Compare to DJIA===========
[*********************100%***********************]  1 of 1 completed
Shape of DataFrame:  (188, 8)
```

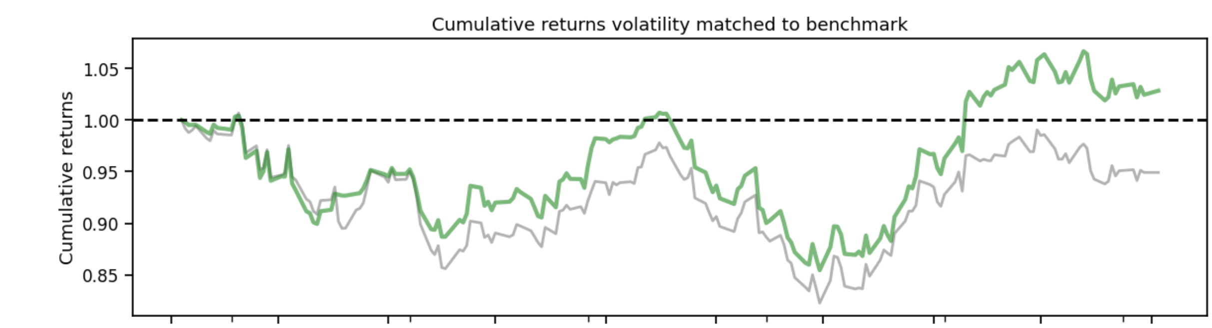
Start date	2022-04-04
End date	2023-01-03
Total months	9

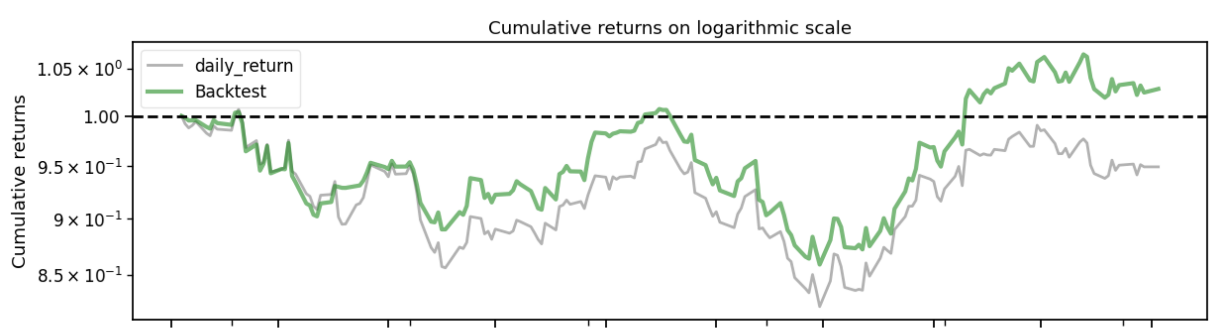
	Backtest
Annual return	3.758%
Cumulative returns	2.806%
Annual volatility	20.104%
Sharpe ratio	0.28
Calmar ratio	0.26
Stability	0.12
Max drawdown	-14.727%
Omega ratio	1.05
Sortino ratio	0.42
Skew	NaN
Kurtosis	NaN
Tail ratio	1.15
Daily value at risk	-2.51%
Alpha	0.11
Beta	0.91

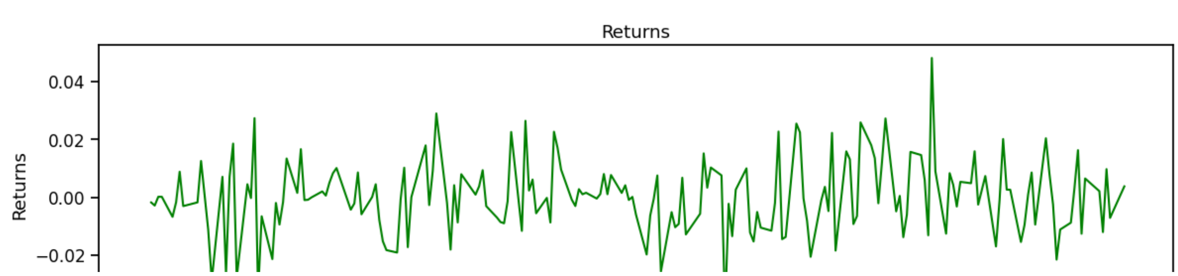
Worst drawdown periods	Net drawdown in %	Peak date	Valley date	Recovery date	Duration
0	14.73	2022-08-16	2022-09-30	2022-11-10	63
1	11.33	2022-04-20	2022-06-16	2022-08-16	85
2	4.34	2022-12-13	2022-12-19	NaT	NaN
3	2.51	2022-12-02	2022-12-09	2022-12-13	8
4	1.81	2022-11-25	2022-11-29	2022-11-30	4

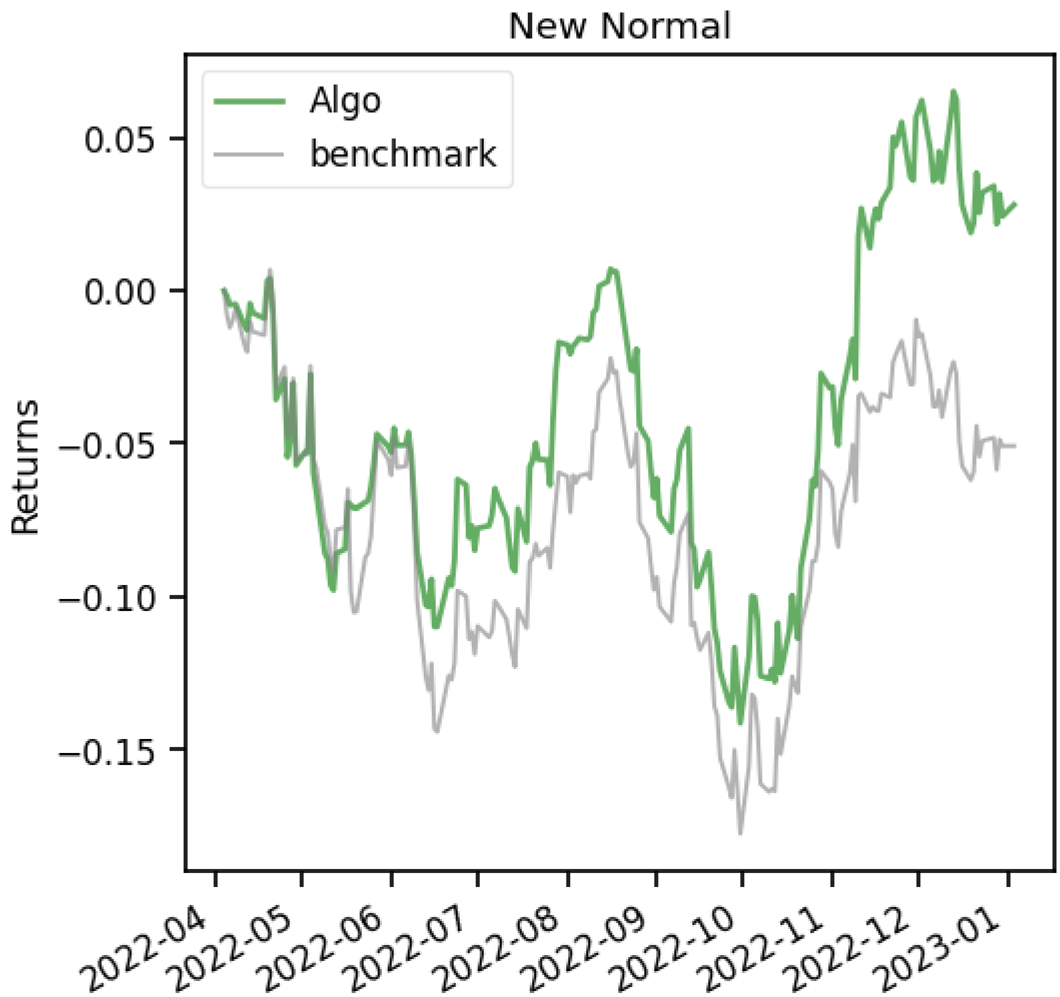
Stress Events	mean	min	max
New Normal	0.02%	-3.90%	4.81%


Cumulative returns
daily_return
Backtest
Cumulative returns
1.05
1.00
0.95
0.90
0.85


Cumulative returns volatility matched to benchmark
Cumulative returns
1.05
1.00
0.95
0.90
0.85


Cumulative returns on logarithmic scale
daily_return
Backtest
Cumulative returns
1.05×10^0
1.00
9.5×10^{-1}
9×10^{-1}
8.5×10^{-1}


Returns
Returns
0.04
0.02
0.00
−0.02


New Normal
Algo
benchmark
Returns
0.05
0.00
−0.05
−0.10
−0.15
2022-04
2022-05
2022-06
2022-07
2022-08
2022-09
2022-10
2022-11
2022-12
2023-01