UniFi

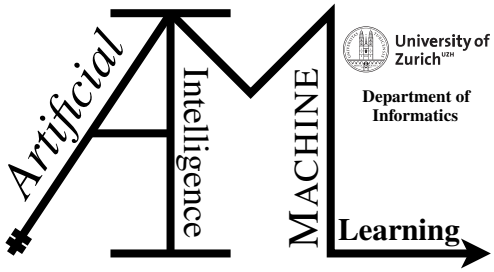
A Unified Framework for Portfolio Management

Master's Project

Barış Özakar
Doğan Parlak
Emine Didem Durukan
20-743-365, 20-743-316, 20-745-733

Submitted on
August 5 2022

Master's Project Supervisor
Prof. Dr. Manuel Günther



Master's Project

Author:
Barış Özakar
Doğan Parlak
Emine Didem Durukan,
baris.oezakar@uzh.ch
dogan.parlak@uzh.ch
eminedidem.durukan@uzh.ch

Project period: 03/15/2022 - 08/05/2022

Artificial Intelligence and Machine Learning Group
Department of Informatics, University of Zurich

Acknowledgements

We would like to thank Prof. Dr. Manuel Günther and Dr. Mario Sikic for their support, and guidance during the course of the project.

Abstract

In the literature, different approaches have been proposed in order to apply portfolio management in a "smart" way. A subset of approaches uses deep machine-learning methods that attempt to predict price movements or trends, see Heaton et al. (2017), Jiang et al. (2017). Another methodology that has been used frequently in the portfolio management field is Deep Reinforcement Learning. Deep RL methods aim to directly optimize the policy, without explicitly predicting future prices, see Moody and Saffell (2001), Dempster and Leemans (2006), Jiang et al. (2017).

Although there has been a remarkable number of approaches to solve the portfolio management problem, which still remains as a challenge is to compare them in a unified manner. The proposed approaches for the problem of portfolio management have their own way of preparing and collecting data, different protocols and metrics in the application of back-testing. Hence, comparisons of these approaches currently are just on the level of ideas, it is very hard to pinpoint performance differences with common financial metrics since there is not a unified framework around the portfolio management problem.

Hence we propose the framework "UniFi": A framework that provides all of the functionalities required to accomplish portfolio allocation in a user-friendly manner. It consists of several machine learning and deep reinforcement learning models accompanied by data collection, preparation and backtesting utilities. UniFi can also be tailored depending on the needs of the user. It is flexible enough for users to implement their own mechanisms.

Contents

Chapter 1

Problem Definition & Related Work

Portfolio management is the decision making process of continuously reallocating funds into financial investment products, aiming to maximize investment return while restraining the risk, see Haugen (1986). There are several approaches to the problem of portfolio management which are built with different combinations of methodologies and algorithms, making use of different datasets and timeframes.

Some of the existing deep machine-learning approaches attempt to predict price movements or trends, see Heaton et al. (2017). However, price predictions are not market actions, converting them into actions requires additional layer of logic, see Jiang et al. (2017). Other approaches to the portfolio management problem directly optimize the policy, without explicitly predicting future prices. These are model-free and fully machine-learning schemes, see Moody and Saffell (2001), Dempster and Leemans (2006). Moreover, deep RL is lately drawing much attention due to its remarkable achievements in playing video games V. Mnih (2015). A general-purpose continuous action and state space deep RL framework, the actor-critic Deterministic Policy Gradient Algorithms, was recently introduced, see V. Mnih (2015), Lillicrap et al. (2015). The paper by Jiang et al proposes an RL framework specially designed for the task of portfolio management, using the Ensemble of Identical Independent Evaluators (EIIE) topology, see Jiang et al. (2017). Finally, FinRL is an open-source framework to help practitioners establish a development pipeline of trading strategies based on deep reinforcement learning (DRL). Their goal is to design a Deep Reinforcement Learning trading strategy which includes: preprocessing market data, building a training environment, managing trading states, and backtesting trading performance Yang (2022).

Although there has been a remarkable number of approaches to solve the portfolio management problem, which still remains as a challenge is to compare them in a unified manner.

The proposed approaches for the problem of portfolio management each have their own way of preparing and collecting data, different protocols and metrics in the application of back-testing. Hence, comparisons of these approaches currently are just on the level of ideas, not on financial performance measured with common financial metrics. There is a lack of unified framework which provides

- Different portfolio allocation approaches, may that be Reinforcement learning based or conventional machine learning models,
- The necessary utilities like data collection, processing, model training and evaluation all together under one system,
- Extensibility such that custom utilities can be implemented within the framework,

- Systematic comparison protocols for different approaches according to their performance

Hence, a unified framework for collecting, preparing, training, testing and evaluating portfolio management agents has been sought after in the field. Therefore, our goal with UniFi is to create a platform which provides all of the functionalities required to accomplish portfolio allocation in a user-friendly and extensible manner, while allowing for systematical comparisons of different approaches according to their performance.

By using UniFi, researchers can collect and process the necessary financial data, create agents for portfolio allocation and evaluate the resulting agent against multiple other agents and sector standard benchmarks.

UniFi can also be tailored depending on the needs of the user. The UniFi framework provides flexibility for users to implement their own mechanism and integrate it into the system.

Finally, for the entire plethora of possible combinations of agents implemented using UniFi, we provide systematic evaluation protocols that compare agents not only on the level of ideas but on the level of agent performances.

Chapter 2

The UniFi Framework

In this chapter first we give a high-level overview of the framework and explain the layers of the system in detail.

2.1 High Level Overview

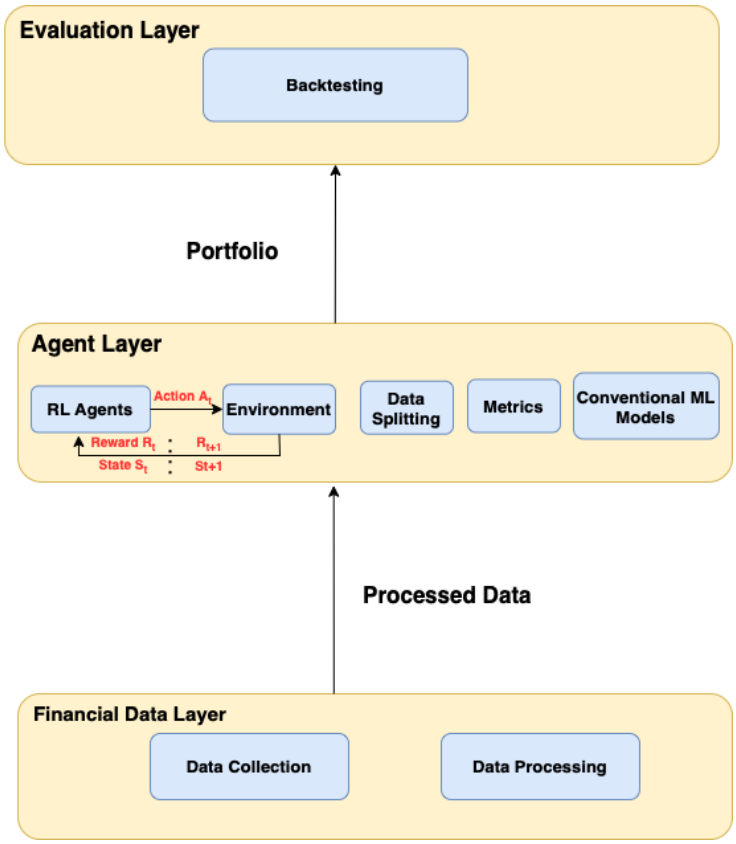


Figure 2.1: Framework Layers

The proposed framework is structured in three layers: Financial Data Layer, Agent Layer, and the Evaluation Layer as seen in the Figure 2.1. Starting from the bottom, Financial Data Layer consists of classes related to the collection and the processing of data as shown in Figure 2.2. The data collection utilities include downloading data via an API, for example Yahoo Finance API, and importing data directly from a CSV, JSON, or XML file. The framework is structured in a way that if needed, one could implement their data importing strategy and integrate it into the framework.

Another set of operations we support in the Financial Data Layer is data processing. The feature engineering functionality in this layer helps the users to extend their dataset either with default sector standard features such as technical indicators, return values, covariance, etc. or with user-defined features. The framework also provides additional functionalities to clean the data. Here again, the system is flexible enough for the implementations of additional functionalities depending on the need of the user.

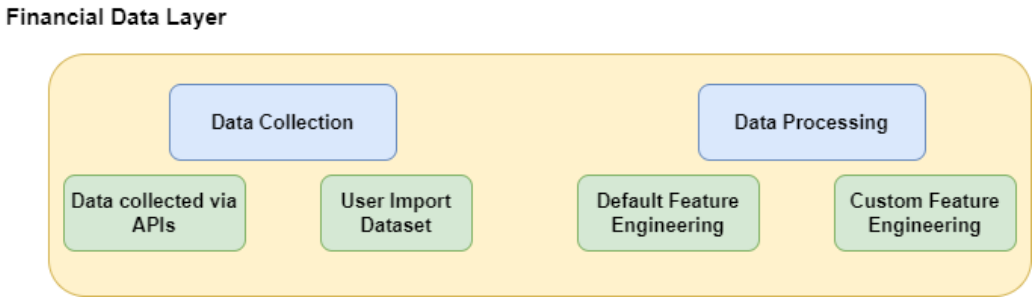


Figure 2.2: Financial Data Layer

One level up, we have the Agent Layer (Figure 2.4). Th Agent layer supports conventional machine learning models and deep reinforcement learning agents next to the environment, data splitting, and regression metrics implementations as seen in the Figure 2.1. The uniFi framework provides Support Vector Regressor Smola and Schölkopf (2004), Huber Regressor Huber (1973), Random Forest Regressor Genuer et al. (2008) , Decision Tree Regressor Ayyadevara (2018), and Linear Regression Weisberg (2005). For Deep RL agents, system currently supports A2C, DDPG Lillicrap et al. (2015), PPO Schulman et al. (2017), and TD3. The Agent Layer also consists of implementation of the PortfolioEnvironment for Deep RL agents to learn. Currently, it has a default environment for the purposes of the portfolio management problem. Once again, the user is provided the ability to implement or add their agents and/or environments.

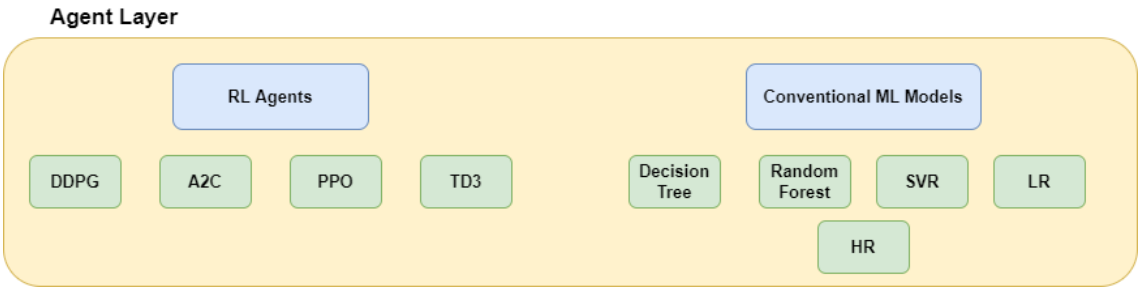


Figure 2.3: Agent Layer,First Part

In addition, the Agent layer includes methods for data splitting and regression metrics. Since the data we are tackling is time-series data, conventional randomized train and test split schemes do not work. Hence, the uniFi framework supports tailored functionalities for specifically dealing with time-series data such as the "Blocking Time Series Splitter" and the sequential "Time Splitter". Moreover, metrics to be used for regression are also supported. Once again, different approaches for data splitting or regression metrics can be implemented and integrated to the system if needed.

Figure 2.4: Agent Layer,Second Part

Finally, we have the Evaluation Layer. The evaluation Layer consists of functionalities for users to evaluate their trained strategies. The uniFi framework provides methods for backtest statistics. Moreover, it also provides functionalities to visualize the evaluation results through several plots.

In conclusion, these three layers interact in a way that in the end, the whole system creates a unified platform where different approaches can be compared.

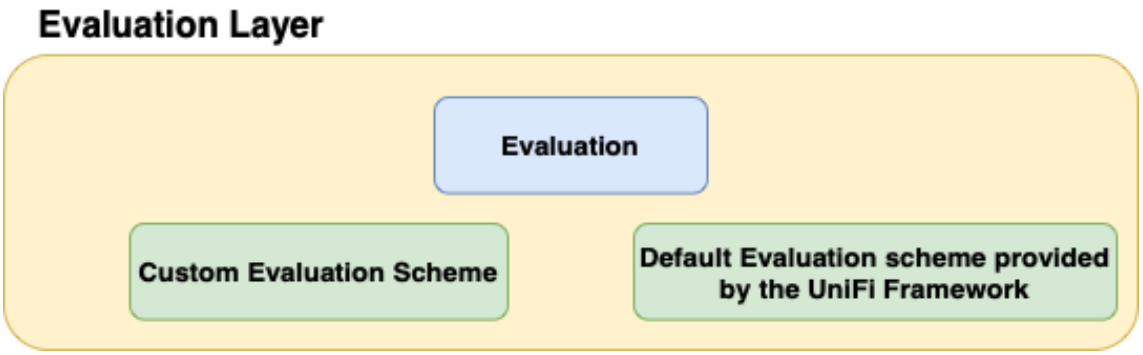


Figure 2.5: Evaluation Layer

2.2 Use Case Diagram

The primary actor for the uniFi framework is the user (i.e. researcher, scientific assistants). There are several goals that the user wishes to accomplish by interacting with the system which can be identified as "use cases". The base use cases are the main goals that the users of the system wishes to accomplish. For the UniFi framework, the base use cases include preparing the data, building the portfolio management model, training the model and evaluating the model. The interactions between the primary actor and use cases is defined as relationships. The Figure 2.6 shows several association, include, and extend relationships.

These relationships are identified as following:

- **Association** is the relationship between actors and base use cases. In the Figure 2.6, these are shown as lines connecting the base use cases with our primary actor.
- **Generalization** is the relationship that represents inheritance between a parent use case and a child use case, where each child use case shares the common behaviors of the parent use case.
- **Include** is the relationship between base use cases and included use cases. Included use case is a use case that always takes place with the related base use case.
- **Extend** provides an option to extend the behavior of the base use case. Extended use case is a use case that takes place only at particular scenarios with the related base use case.

1. Base use case: **Prepare data**
    - Child use case: Download data
    - Child use case: Import data
    - Child use case: Prepare train “data” (for ML)
    - Child use case: Prepare “environment” (for RL)
    - Extend use case: Extend data, ie. add features
        - Child use case: Extend data with readily implemented indicators
        - Child use case: Extend data with custom features
2. Base use case: **Create model** for the portfolio management agent
    - Extend use case: Build readily implemented ML model (SVR, RF, LR, DT)
    - Extend use case: Build readily implemented RL model (A2C, PPO, DDPG)
    - Extend use case: Build custom model
    - Included use case: Get model parameters
3. Base use case: **Train model**
    - Included use case: Get training parameters
4. Base use case: **Evaluate model**
    - Included use case: Perform backtest
    - Included use case: Plot financial visuals
    - Extend use case: Perform comparison against benchmark

An example use of the system will be:

1. The user wants to download data (from the Yahoo Finance API).
2. The user wants to extend the downloaded data with default technical indicators (from the StockStats library).
3. The user proceeds to prepare the environment for the RL Agents. The framework will use the OpenAI Gym and Stable Baselines libraries to build the relevant portfolio environment.
4. The user prepared his environment for Deep RL agents and now they want to build an RL Agent to train. In this case, the uniFi framework will build the relevant RL model (A2C, PPO, DDPG from Stable Baselines).
5. The user trains the model (with the interactions between the agent and the environment observations).
6. Upon building and the user wants to evaluate the trained model against a benchmark. The benchmark is a baseline portfolio that allows the user to gauge the performance of the trained agent. The framework provides exogenous market indexes and sector standard passive portfolio strategies as readily implemented benchmarks. The UniFi framework will perform backtest and calculate and plot the financial statistics by utilizing the Pyfolio library.

The machine learning scheme is different from the Deep RL scheme. Yet, the above example will differ only in the 3rd and 4th steps.

3. The user prepares an ML dataset with train and test splits.
4. The user creates an ML model (SVR, RF, LR, DT from Scikit-Learn)

System also supports customizing the framework depending on the need of the research project. As seen in the figure 2.6, user can build custom models, import custom datasets and add custom features to the dataset, and create different backtesting schemes.

2.3 Layers of the Framework

2.3.1 Financial Data Layer

Financial Data Layer consists of every processes that is related to data. These include: data collecting and data processing operations as seen in Figure 2.2. Data collection can be done in two different ways in the uniFi framework.

- **Downloading the Data**: Data can be downloaded using an API. Currently, the uniFi framework supports Yahoo Finance API.
- **Import Custom Dataset**: Users also have the option to import their own dataset. JSON,CSV, and XML file types are supported by the framework.

The structure of the Data Collector module can be seen in Figure 2.7. Every approach for data collection have been implemented as a class in order to maintain the modularity of the framework and to increase the flexibility of the system, as they inherit from the *Data Collector* abstract base class. *Data Collector* class provides the abstract method *"collect()"*. Method *"collect()"* has

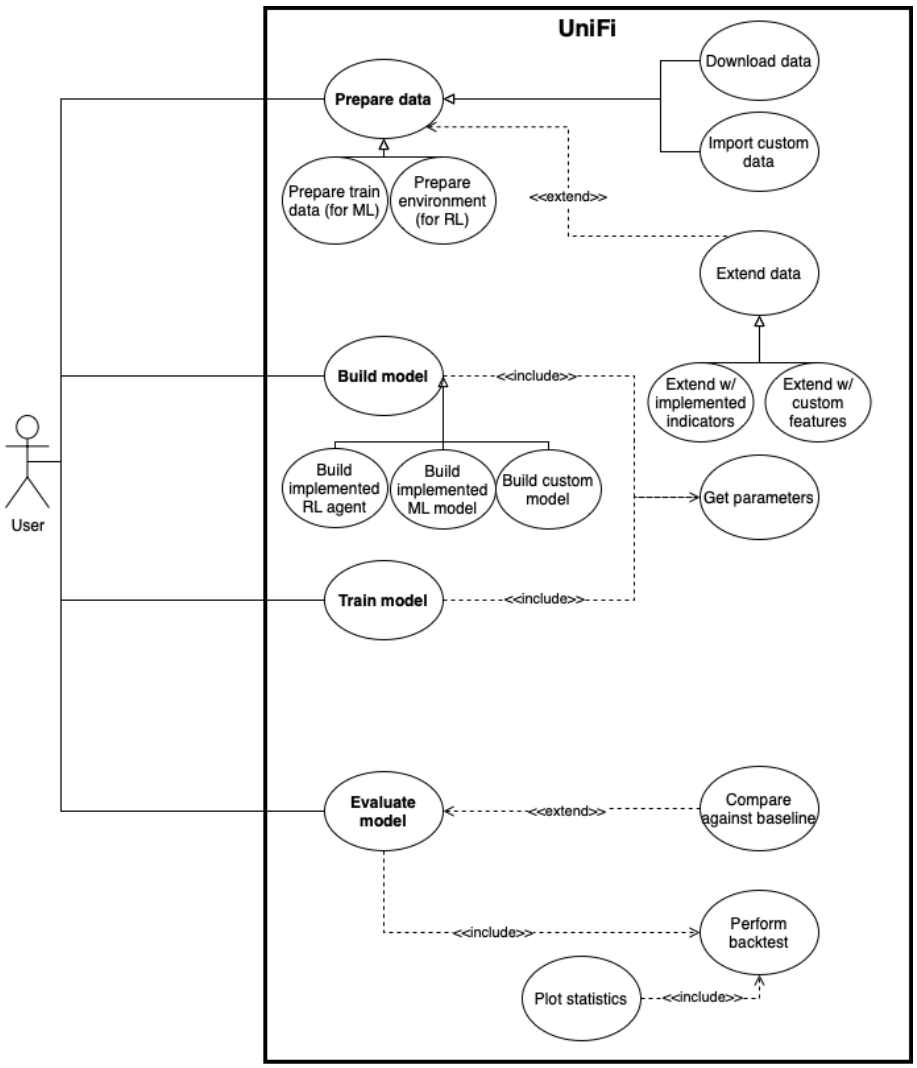


Figure 2.6: Use Case Diagram

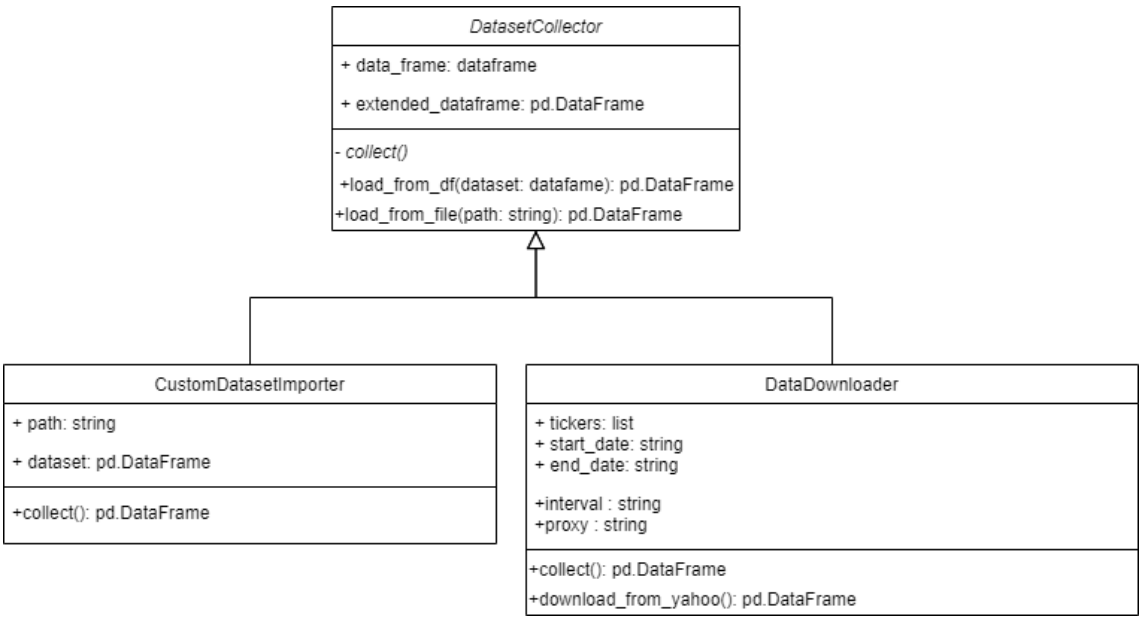


Figure 2.7: Data Collector Class Diagram

been implemented in every child class according to the purpose of the respective class. Methods *"load_from_df()"* loads the data from a dataframe and *"load_from_file()"* loads the data from a file.

First approach to data collection is to import a custom dataset. For this aim, we introduce the *CustomDatasetImporter* class. The *CustomDatasetImporter* class provides methods for users to import their data. So far, The uniFi framework supports JSON,XML, and CSV file types. The user only needs to give the path of the file to be loaded as input to the *CustomDatasetImporter*, and it will load the data.

For users to download data, we provide the *DataDownloader* class which provides functionalities to download data from Yahoo Finance API. Class attributes of *DataDownloader* class are the following:

- **Tickers:** List of tickers to be downloaded.
- **Start Date:** Starting date of the data.
- **End Date:** Ending date of the data.
- **Interval:** Interval to sample the data.
- **Proxy:** Flag for whether to proxy URL scheme when downloading downloading the data.

"download_from_yahoo()" method downloads the data via Yahoo Finance API, and *"collect()"* method loads the downloaded data.

Due to the framework's modular and flexible structure, the extension of the system is fairly easy. If a user wants to collect his data in a different way, they can do so by just creating another child class that follows the same structure with other classes, and integrate it into the system. An example for an extension can be seen in the Figure 2.8.

Besides data collection methods, Financial Data Layer also provides methods for feature engineering.

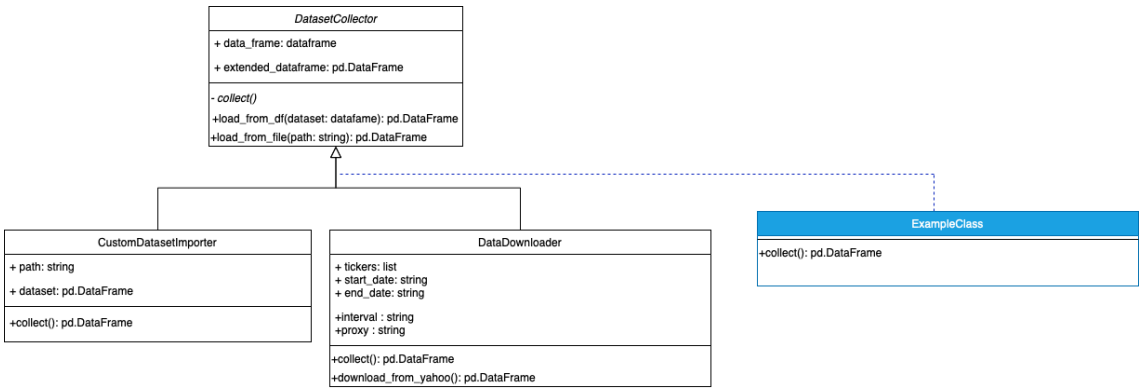


Figure 2.8: Data Collector Extension

The structure can be seen in Figure 2.9. The *"FeatureEngineer"* abstract base class contains *"extend_data()"* abstract method which has to be implemented in every child class. Furthermore, every approach has its own class and implements the *"extend_data()"* according to its objective.

There are two different ways to do feature engineering in the uniFi framework:

- **Default Feature Engineering:** The framework comes with default feature engineering methods. These methods include adding technical indicators, adding vix (volatility index, see Whaley (2009)), adding turbulence (see Zhang (2020)), and adding covariances and returns. It also provides a method for preparing the data for machine learning.

- **Custom Feature Engineer:** Users can extend their data with their own features.

"DefaultFeatureEngineer" class contains all the functionalities to extend the data with default features. Attributes of "DefaultFeatureEngineer" class are the following:

- **use_default**: Flag indicating whether to use default features or not.

- **use_covar**: Flag indicating whether to use covariances as features or not.

- **use_return**: Flag indicating whether to use returns as features or not.

- **lookback**: Lookback value, is the number of periods of historical data used for observation and calculation. This is mainly used for calculating technical indicators. See tra (2022)

- **use_vix**: Flag indicating whether to use vix as features or not.

- **tech_indicator**: List that contains technical indicators.

- **feature_list**: List of features.

Depending on the values of the above-mentioned parameters, feature engineering will be applied.

The UniFi Framework also creates an environment where users can use their own features to do feature engineering. "CustomFeatureEngineer" class provides the functionalities for users to apply their tailored feature engineering process. The function *"add_user_defined_features()"* adds the user defined features to current data, and *"extend_data()"* method extends the current data.

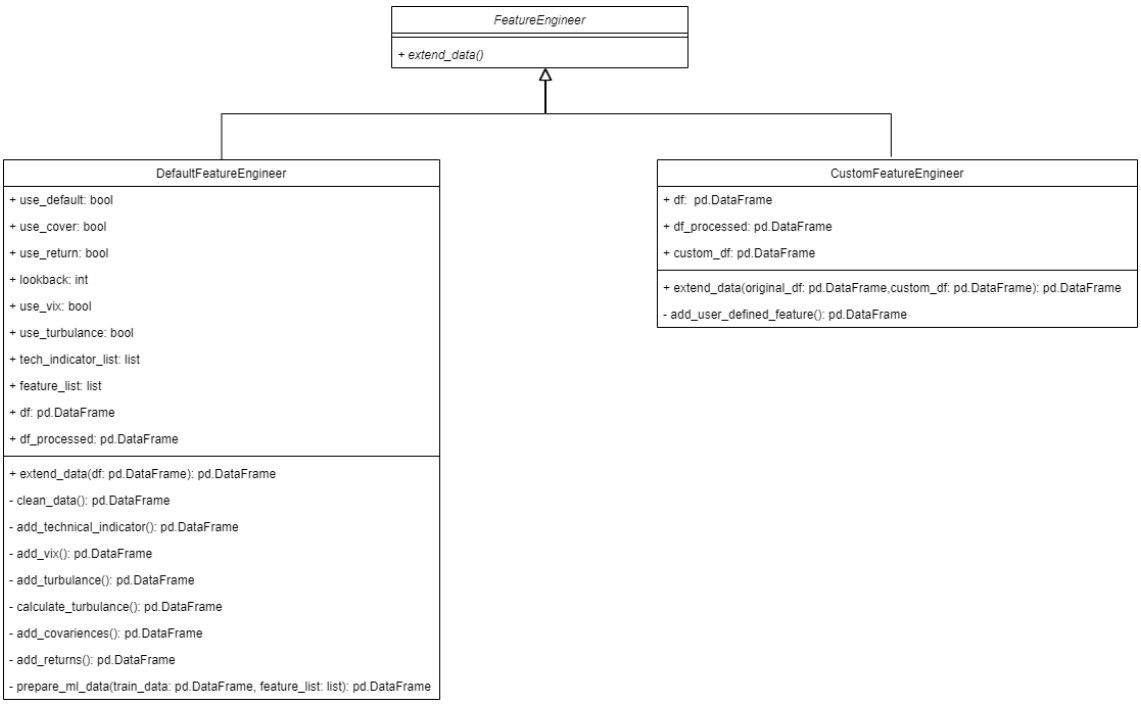


Figure 2.9: Data Processing Class Diagram

2.3.2 Agent Layer

Agent Layer consists of the following modules (see Figure 2.11):

- **Conventional Agent Module:** This module provides conventional machine learning module. Currently UniFi supports: Huber Regressor, Linear Regression, Decision Tree, Random Forest and Support Vector Regressor Agents.
- **Reinforcement Learning Agent Module:** RL Agent module consists of reinforcement learning agents which are: TD3, A2C, PPO, and DDPG.
- **Environment Module:** Provides methods for creating an environment for RL agents to interact. UniFi comes with a default envrionment for portfolio management.
- **Data Splitter Module:** Provides functionalities for data splitting.
- **Metrics Module:** Contains several metrics to assess the performance of the regression task with the help of "sckit-learn" library

Agent layer contains several algorithms with their corresponding methodologies to apply portfolio allocation. There is an abstract base class called Agent which forces Reinforcement Learning agents and Conventional agents to implement the fundamental methods *"train_model"*, *predict*, *"save_model"*, *"load_model"*. A high-level structure of the Agent Layer can be seen in Figure 2.10.

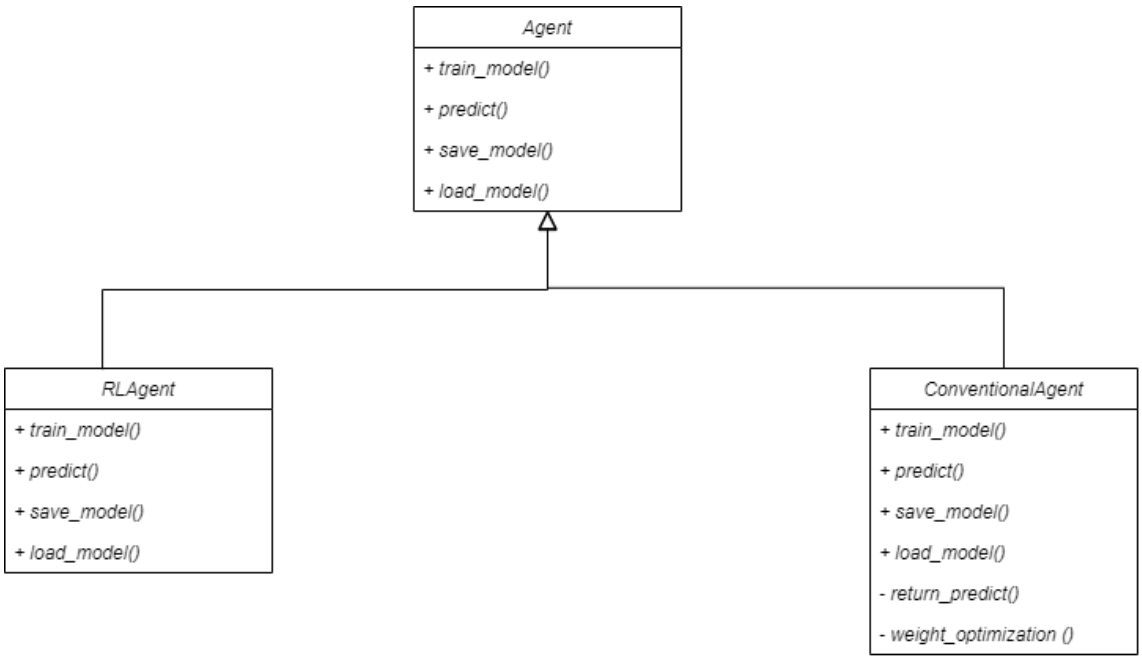


Figure 2.10: Agent Layer Class Diagram

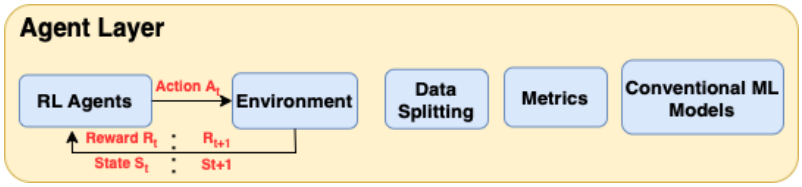


Figure 2.11: Agent Layer

In order to assess performance of return prediction step of conventional agents, metrics section is implemented with several types of performance metrics. Furthermore, there is Data Splitter section which implements TimeSeriesSplitter and BlockingTimeSeriesSplitter classes. Finally, there is environment section which is used to provide environment for training and testing purposes of Reinforcement Learning agents. This section contains an abstract base class Environment which forces a Portfolio environment to include *"reset"*, *step*, *"render"*, *"get_env"* methods.

Data Splitter

Splitting the data is an essential section of the framework for training and testing. Data splitting needs to be handled with caution when it comes to time series data since shuffling the data breaks time dependence. Therefore, UniFi provides an extensible Data Splitter module with Abstract base class implementations to split the data into train and test sets. The UniFi currently has two splitting schemes implemented under TimeSeriesSplitter and BlockingTimeSeriesSplitter classes. TimeSeriesSplitter class provides utilities for splitting the data sequentially, enabling the user

to choose a time interval which returns the data in the specified range. It also provides utility methods for cross-validation splits as shown in Figure 2.12.

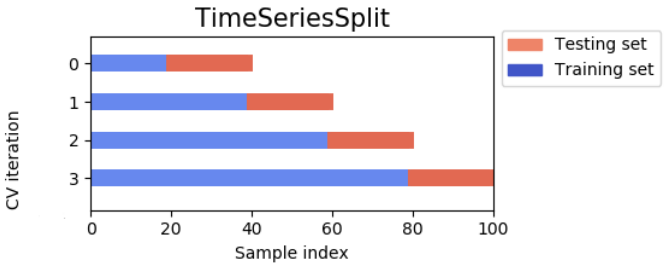


Figure 2.12: Time Series Splitter

- **__init__**: Initializes the TimeSeriesSplitter object.
- **_iter_test_indices**: Generates integer indices corresponding to test sets.
- **split**: Generate indices to split data into training and test set.
- **get_split_data**: Splits the data to test data or train data.
- **get_next_df_date**: Gets the next date from the given dataset.

On the other hand, the BlockingTimeSeries class provides an alternative cross-validation split by returning the data in non-intersecting blocks that prevents data leakage. One can utilize BlockingTimeSeriesSplitter class as shown in Figure 2.13. Methods of the BlockingTimeSeriesSplitter class can be seen below.

- **__init__**: Initializes the BlockingTimeSeriesSplitter object.
- **_iter_test_indices**: Generates integer indices corresponding to test sets.
- **split**: Generate indices to split data into training and test set.
- **get_n_splits**: Gets the number of splitting iterations in the cross-validator (i.e number of train-test pairs).

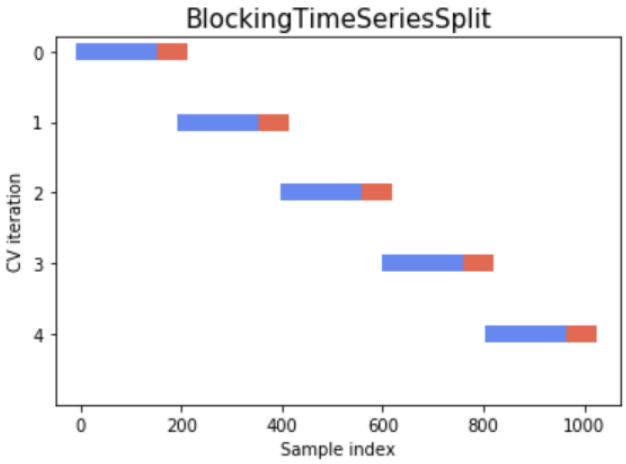


Figure 2.13: Blocking Time Series Splitter

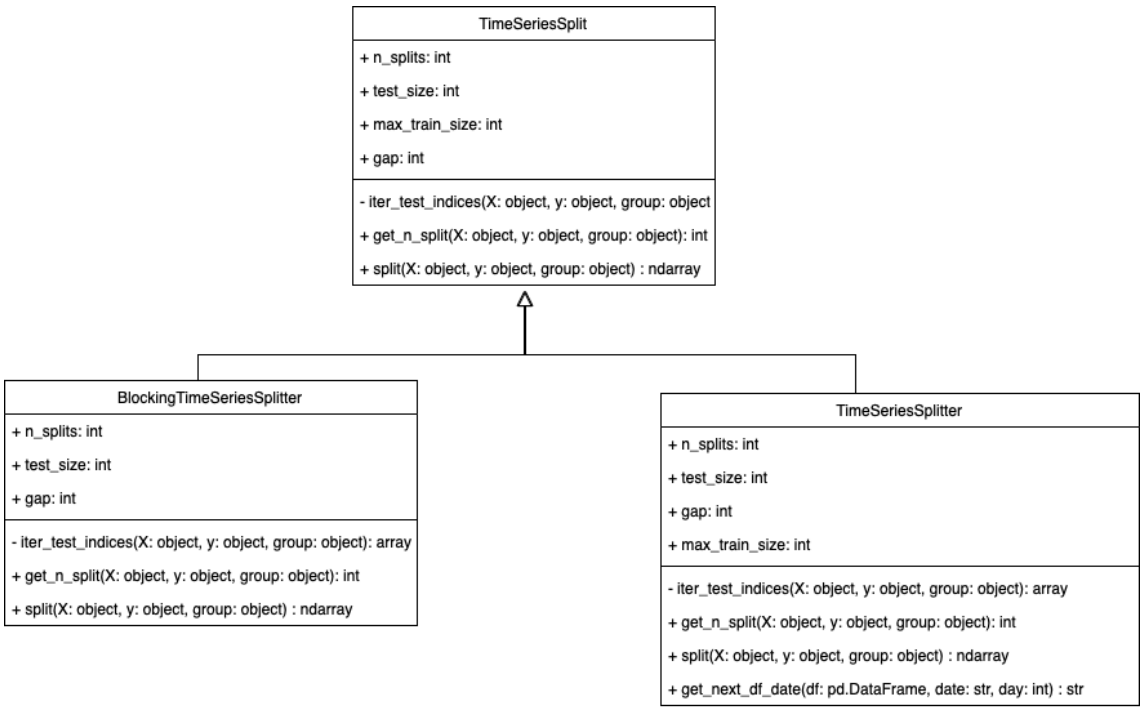


Figure 2.14: Data Splitter Class Diagram

Environment

Portfolio Environment Markov Decision Process (MDP): Markov Decision Process is a mathematical framework to describe environment in reinforcement learning. This is a crucial part for an rl agent and describes the observations and the interaction between the agent and the environment in a sense. The MDP defines the state space, action space, and the reward function.The

agent learns the policy that maximizes the expected return.

- **State space**:The state space describes an agent's perception of the market. State s represents the features of each ticker including correlations between them. These features include several technical indicators e.g MACD, Bollinger up, Bollinger down, RSI, close etc.
- **Action space**: The action space describes the allowed portfolio weights that the agent interacts with the environment. Each element in the portfolio weights is in range [0, 1]. Sum of the actions (portfolio weights) in state s is equal to 1, i.e keeping cash in any state is not allowed. All of the money has to be allocated to the tickers included in environment. At each step the action of the agent is to change the allocation of weights. Looking at the decrease or increase in the weight one can observe that whether the actions was a buy or sell order.
- **Reward Function**: r(s,a,s') is the incentive mechanism for an agent to learn a better action. r(s, a, s) = new portfolio value - transaction cost (from state s to s', for both buying and selling)

Starting with an initial state, an agent decides its action which is currently available in that state. After that, the agent observes the new reward and new state of the world as a response from environment. This cycle repeats until the last day of the world is reached.
Two distinct portfolio environments namely training environment and trading environment is necessary in order to be used in training and trading phase of RL agents.In order to construct the training and trading environment, the user should feed the training data and trading data.
Training data represents the observations that the agent is able to interact enough to learn the market behaviour. On the other hand, trading data is unobserved and the agent uses the information what is learnt in training phase in order to gain as much reward as possible in trading environment.

Environment module contains two classes which are named as Environment and PortfolioEnv. Prior one is an abstract base class which forces an environment to be used by reinforcement learning agents to implement the following methods:

- **reset**: Resets the environment.
- **step**: Steps the environment with the given action.
- **render**: Renders the environment.
- **get_env**: Creates and returns the vectorized environment.

Environment class inherits this structure from OpenAI Gym. Besides these methods, it also contains a static method called softmax normalization in order to normalize actions with softmax, as seen in Figure 2.15.

In order to ease the situation for researchers, there is an implementation of this Environment which is in PortfolioEnv class. This class allows user to define the following parameters to initiate their own environment by using the default implementation of methods in Environment class.

- **df**: input data
- **stock_dim**: number of unique securities in the investment universe
- **initial_amount**: initial cash value
- **transaction_cost_pct**: transaction cost percentage per trade
- **state_space**: state space

Figure 2.15: Environment Class Diagram

- **action_space**: action space
- **feature_list**: a list of features to be used for the state observations
- **lookback**: number of previous trading days to take into account
- **day**: an increment number to control the indexing of the date

Following are the methods implemented by PortfolioEnv class.

- **__init__**: Utilizing the parameters above, action space is defined such that minimum and maximum values are defined to be in range 0 and 1 indicating the minimum and maximum proportion for a ticker that can be allocated in the portfolio. Data related to current trading day is stored and used to extract correlations between tickers from that day. State variable is defined to store features of tickers from the current trading day. A flag variable called terminal is initialized to indicate whether it is the last trading day or not. Portfolio value is set to initial amount of money. Finally memory variables for portfolio values, actions and dates are initialized.

- **reset**: Resets the environment by setting the class parameters to default version and returns the initial states.
- **step**: Steps the environment with the given action and provides the new state, new reward function which is portfolio value and a flag indicating whether terminal value is reached or not. If terminal state is not reached, uses softmax normalization to normalize weights and stores the actions. Taking into account the previous and recent version of weight allocations, calculates the total change of weight distribution and multiplies them with transaction cost percentage and last value of portfolio in order to subtract total transaction cost from the reward function. After that, updates the day, current data, states and portfolio value. Stores the relevant information into memory variables. If the terminal state is reached, initial portfolio value and final portfolio value is reported to the user with the sharpe ratio.
- **render**: Renders the environment and returns the state.
- **get_env**: Creates and returns the vectorized environment.

Reinforcement Learning Agent

Objective of reinforcement learning agent is to train the agent in the training environment to get familiar with the market behaviour and use that experience to further trade in trading environment in order to find the best portfolio allocation. Compared to Conventional Agents, reinforcement learning based agents utilizes the underlying Markov Decision Process (MDP) in the environment to directly learn the policy that generates the portfolio weights. Hence the idea is rather implicit and linked to MDP.

To begin with, we have utilized Stable Baselines3 in order to construct the RL agents. To be able to train, we feed the training environment to agent which utilizes user defined parameters in order to train via the learn method from Stable Baselines3. On the other side, in the prediction phase, a similar approach is taken. Trading environment is fed to the trained agent. For each trading day in trading environment, based on a given state, an action is taken using the predict method of "stablebaselines3" and then environment leads us to the next state of the world based on our agents decision. This continues until the end of trading day.

There is an abstract base class called RLAgent which inherits the abstract base class Agent and forces the agent classes A2C (Advantage Actor Critic), DDPG (Deep Deterministic Policy Gradient), PPO (Proximal Policy Optimization) and TD3 (Twin Delayed DDPG) to implement the methods *"train_model()"*, *predict()*, *"save_model()"*, *"load_model()"*. One can initiate the model to be used, train the model utilizing the training environment and apply allocation for unseen trading days using the trading environment. Structure of the RL Agents can be seen in Figure 2.16. Following are the methods with their description which can be generalized for all of the RL agents.

- **__init__**: Initialize the RL agent by creating the corresponding agent from "stable_baselines3's". In addition that, store the training environment for further use in training.
- **train_model**: Train the RL agent based on corresponding training parameters from "stable_baselines3" and utilizing its *"learn"* method.
- **predict**: Gets the trading environment as parameter with corresponding testing parameters from "stable_baselines3". Applies portfolio allocation for each trading day.
- **save_model**: Saves the trained model.

- **load_model**: Loads the trained model.

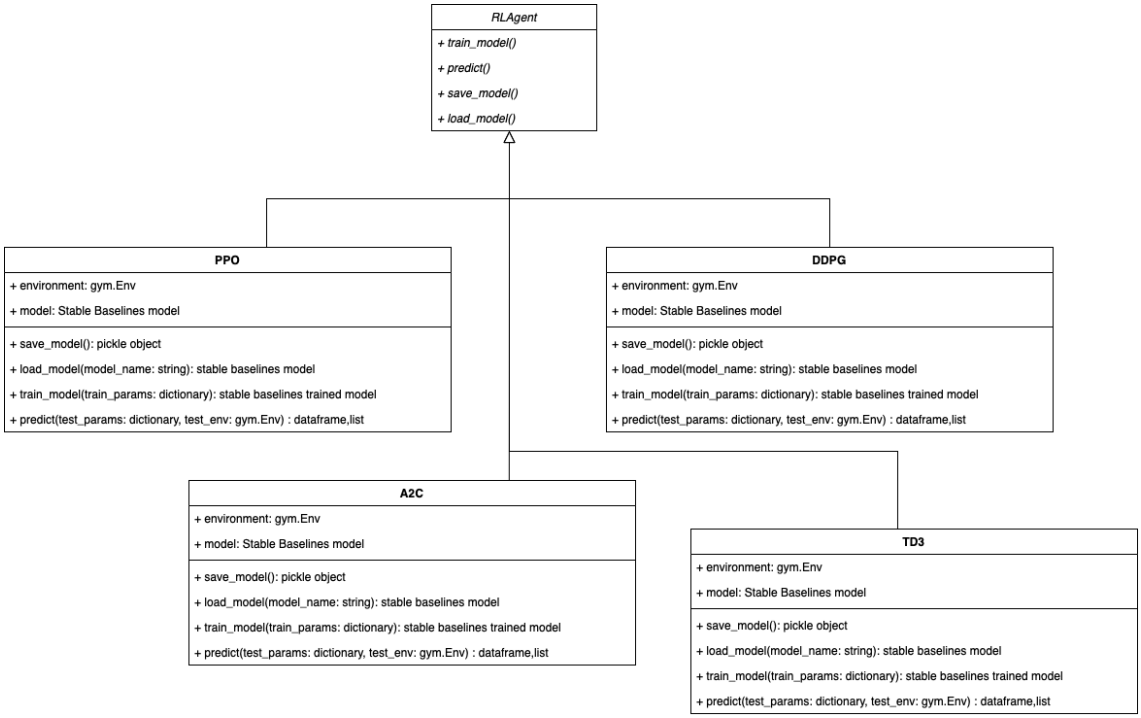


Figure 2.16: RL Agent Class Diagram

Conventional Agent

For conventional machine learning agents, the goal is to predict the return values for the next day and find the best portfolio allocation with the use of Markovitz Efficient Frontier via mean-variance optimization.

In the training phase as features we have the close price and technical indicators; and as targets we have the next day return values. With these data we train our conventional machine learning model, see Figure 2.17 . Here we use the Scikit learn library for training.

In the trading phase, a.k.a testing phase, we feed our trading data to the trained model and predict the return values for the next day. And this is fed to the weight optimization phase along with the covariance between return values. Finally with the help of the efficient frontier we get our optimal portfolio as shown in Figure 2.18.

- **Trading Phase 1: Prediction** Phase one is the prediction phase, and each trading day is taken into account individually. We predict the next days returns given the feature of the tickers of the current day, and this creates our expected return for the upcoming day. Next to the expected return we also need the risk, and the risk is the covariance between the return values of the tickers. This is covariance matrix aka risk is calculated with pyportfolio optimization library. Both of these information are passed to the next phase for weight optimization.

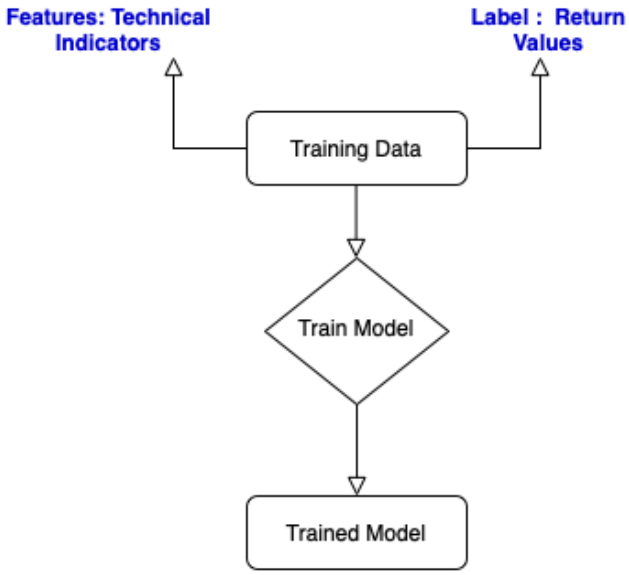


Figure 2.17: Training Phase for Conventional Agents

- **Trading Phase 2: Weight Optimization** Using the expected return and risk that comes from the phase 1, we construct the efficient frontier. The main objective of Efficient Frontier is to maximize the sharpe ratio while considering transaction cost as an additional objective. For an example visualization of Efficient Frontier see Figure 2.19. The Sharpe ratio is a risk ratio that calculates an investment's average returns compared to its potential risks, and it is our main objective. We want a portfolio where the sharpe ratio is maximized. The point where capital allocation line crosses the efficient frontier represents the portfolio with the maximum sharpe ratio, and it is called the tangency portfolio. Essentially, tangency portfolio gives us the portfolio weights. Thus, we have the final output.

There is an abstract base class called Conventional which inherits the abstract base class Agent and forces the agent classes DTAgent (Decision Tree Agent), RFAgent (Random Forest Agent), HRAgent (Huber Regression Agent), LRAgent (Linear Regression Agent) and SVRAgent (Support Vector Regression Agent) to implement the methods *"train_model()"*, *predict()*, *_return_predict()*, *_weight_optimization()*, *"save_model()"*, *"load_model()"*. These agents allow users to use distinct conventional methods in order to use for portfolio allocation task. One can initiate the model to be used, train the model with the training data and further apply allocation based on trading data. Following are the methods with their description which can be generalized for all of the Conventional Agents.

- **__init__**: Initialize the Conventional agent by creating the corresponding model from "sklearn" library.
- **train_model**: Train the model such that input data consists of features for tickers at each training day. The target is the return values of these tickers at those days. Utilize the fit method from "sklearn" library.
- **predict**: Predicts the return values of tickers and finds optimal weight allocation utilizing two helper methods *return_predict*, *weight_optimization*. Initially defines a variable to store weight allocations in each trading day. All of the tickers have equal allocation in portfolio

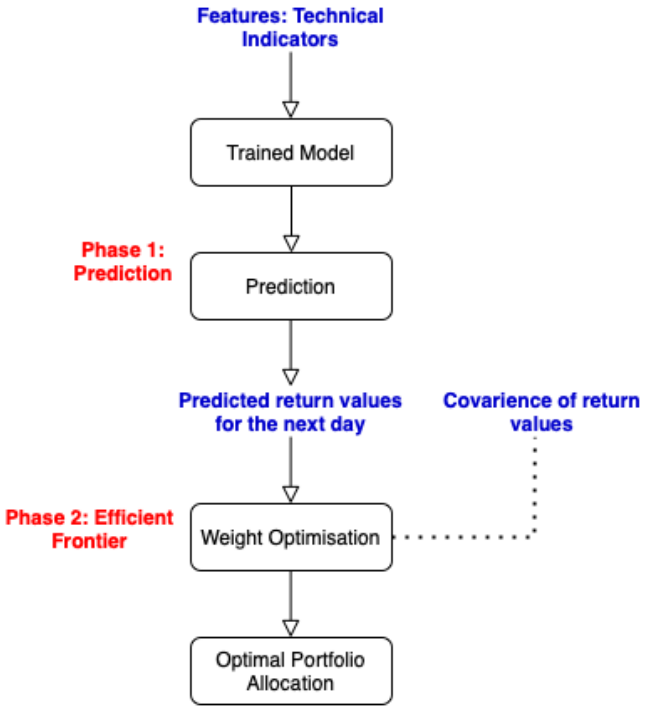


Figure 2.18: Trading Phase Workflow for Conventional Agents

in the first stage. Portfolio value is set to inital amount of money. For each trading day, *return_predict* and *weight_optimization* methods are called respectively.

- **return_predict**: Predict the return values of tickers at trading days based on features of those tickers. First of all current trading days and next trading days data is stored. Features of the tickers at those days are extracted. Current trading days features are used to predict the return of these tickers. Utilizing pypoft libraries risk_model method, sample covariance is obtained whereas expected returns are predicted values.

- **weight_optimization**: Utilizing the expected returns and predicted values, creates an object of "EfficientFrontier" in order to obtain the optimal portfolio allocation by solving non convex optimization which maximizes the Sharpe ratio while taking into account transaction costs as an additional objective. An instance of efficient frontier visualization can be seen in figure 2.19. In order to account for the transaction cost, previous weight allocation of the portfolio is supplied. Constraints of the optimization are weights being greater than minimum weight which is 0 and less than maximum weight which is 1. Afterwards, current amount of cash allocated for each ticker is calculated and used to calculate current shares. Utilizing the price of next state, new portfolio value is stored by multiplying each ticker with corresponding share proportion. Returns the portfolio value and weight allocation.

- **save_model**: Saves the trained model.

- **load_model**: Loads the trained model.

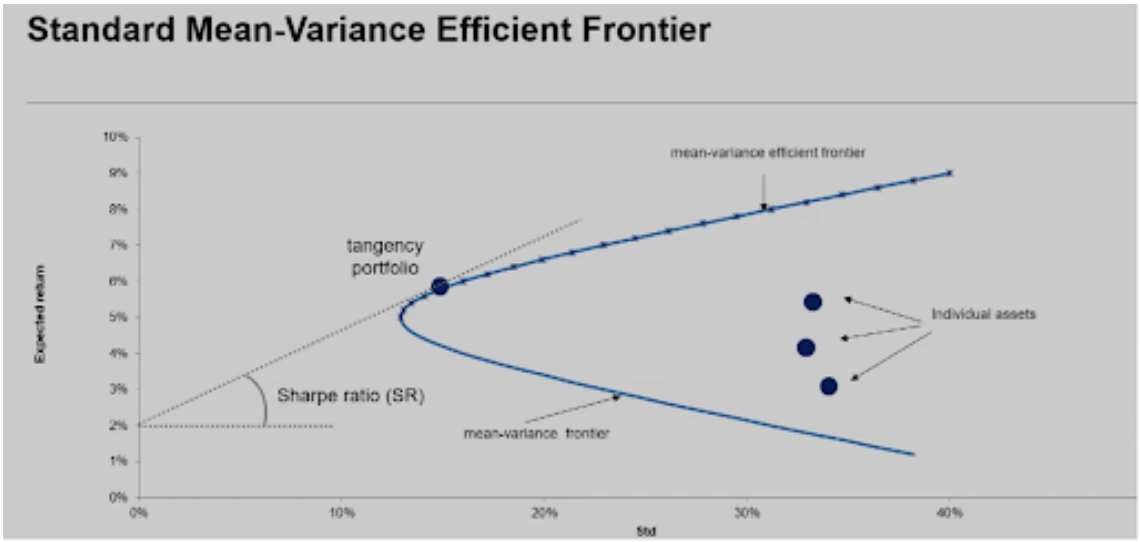


Figure 2.19: Efficient frontier example visualization

Metrics

This section is created to facilitate the researchers for performance evaluations. Initiated conventional methods have prediction method which includes the methods return prediction and weight optimization. As explained before, return prediction utilizes fitted agent to predict the future returns of tickers. This prediction can be assessed by several performance metrics. Following ones are suggested to the researchers:

- **max_error**
- **mean_absolute_error**
- **mean_squared_error**
- **mean_squared_log_error**
- **median_absolute_error**
- **r2_score**
- **explained_variance_score**
- **mean_tweedie_deviance**
- **mean_poisson_deviance**
- **mean_gamma_deviance**

These metrics are implemented as methods and inherited from "sklearn" library. True target values are the known return values where predictions are the predicted return values by trained agent.

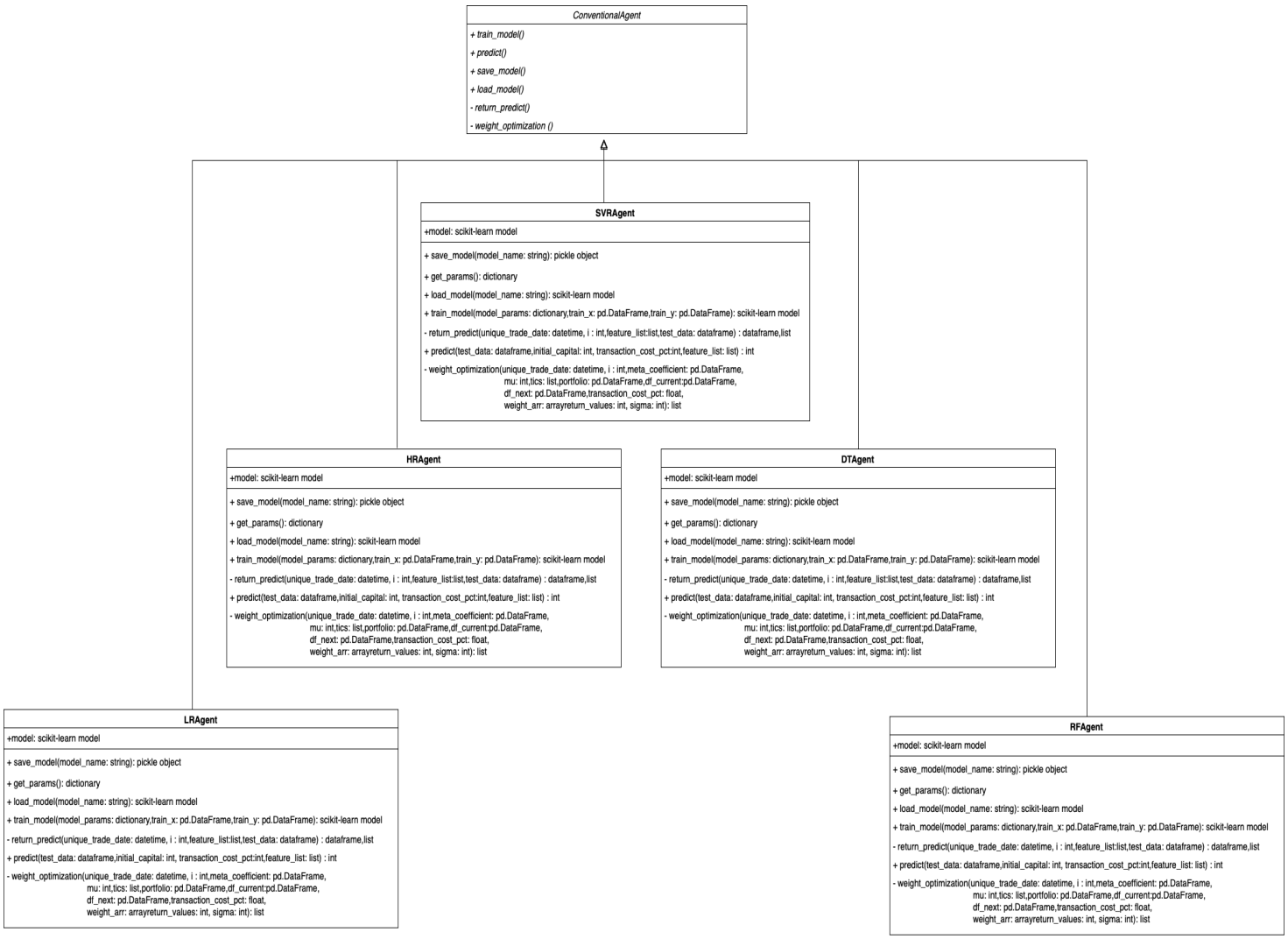


Figure 2.20: Conventional Agent Class Diagram

2.3.3 Evaluation Layer

The evaluation layer provides the utility to backtest and compare multiple agents with respect to each other and against a benchmark in a unified manner. The evaluator layer allows the user to produce several comparative financial statistics and visualizations through backtest_stats() and backtest_plot() methods respectively.

The implemented financial backtest statistics include:

- **Annual return**: The mean annual growth rate of returns, equivalent to the compound annual growth rate.
- **Cumulative returns**: The total change in the investment price over the trading period.
- **Annual volatility**: Annualized variance of returns.

- **Sharpe ratio**: A measure of risk adjusted returns. The Sharpe ratio adjusts a portfolio's expected future performance for the excess risk that was taken by the investor, and is calculated as:

$$\frac{\text{Return of the portfolio - Risk-free rate}}{\text{Standard deviation of the portfolio's excess return}}$$

- **Calmar ratio**: a measure of risk-adjusted returns calculated as

$$\frac{\text{Average annual rate of return}}{\text{Maximum Drawdown}}$$

- **Stability**: R-squared of an ordinary least squares linear fit and returns to the cumulative log returns.
- **Maximum drawdown**: An indicator of downside risk over a specified time period defined as the maximum observed loss from a peak to a trough of a portfolio, before a new peak is attained.
- **Omega ratio**: A weighted risk-return ratio for a given level of expected return that helps identifying the chances of winning in comparison to losing.
- **Sortino ratio**: A variation of the Sharpe ratio that utilizes the asset's downside deviation instead of the total standard deviation of portfolio returns, and is calculated as:

$$\frac{\text{Return of the portfolio - Risk-free rate}}{\text{Standard deviation of the downside}}$$

- **Skewness**: The sample skewness is computed as the Fisher-Pearson coefficient of skewness.
- **Kurtosis**: The fourth central moment divided by the square of the sample variance as calculated by the Fisher's definition.
- **Tail ratio**: The ratio between the right (95%) and left tail (5%).
- **Value at risk**: A statistic that quantifies the extent of possible financial losses within a portfolio.
- **Probabilistic Sharpe Ratio**: A statistic developed by Marcos López de Prado et al, see Bailey and Lopez de Prado (2012), that provides the confidence level associated with a particular Sharpe Ratio estimation by controlling for skewness and kurtosis.

In particular, there are two implementations of the Evaluator abstract base class, both of which can be extended further via the uniFi framework.

- **PortfolioEvaluator** provides the functionality to compare multiple agents against an exogenous index.
- **ExtendedPortfolioEvaluator** extends the functionality of the PortfolioEvaluator. It accepts a portfolio universe and simulates a sector standard passive strategy called the Uniform Buy and Hold (UBAH). The Uniform Buy and Hold is a passive portfolio management approach that allocates the total funds equally into the pre-selected portfolio universe and holding until the end of the trading period. In addition, the ExtendedPortfolioEvaluator provides more information about the distribution of the returns for an extended analysis.

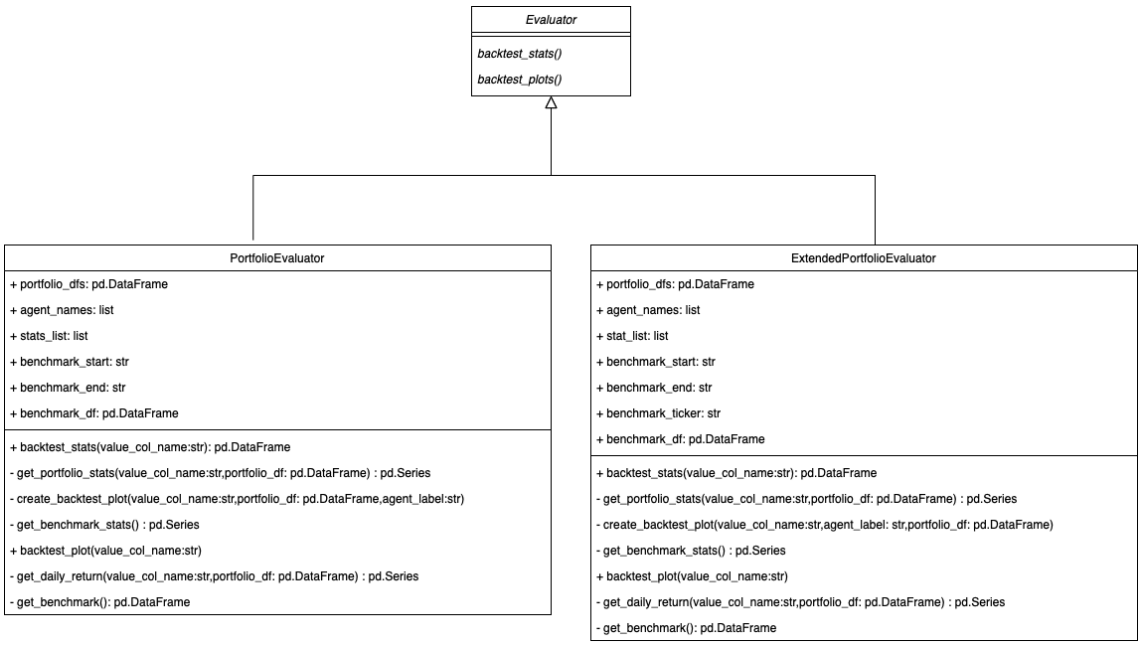


Figure 2.21: Evaluator Class Diagram

Both implementations provide backtest statistics and visualizations that allows the user to compare a set of trained agents with a benchmark in a unified manner. However, ExtendedPortfolioEvaluator improves the notion of a benchmark portfolio by not only taking a single exogenous ticker but giving the ability to compare against a passive strategy with the same portfolio universe. Therefore, it represents a better comparison against an "active" trained agent strategy. Moreover, due to the random walk nature of the price data, it might not be enough to analyze the returns "historically" since history is only a "single" occurrence of infinite possible realizations of the this Geometric Brownian motion. Therefore, it is more informative to look at the "distribution" of returns.

In addition, ExtendedPortfolioEvaluator implements the sector standard Probabilistic Sharpe Ratio (PSR). According to Marcos Lopez de Prado and David H. Bailey, Sharpe ratios are not comparable, unless controlled for the skewness and kurtosis of the returns Bailey and Lopez de Prado (2012). To solve this problem Prado et. al have developed the PSR, which takes those central moment characteristics into account and delivers a corrected, atemporal measure of performance expressed in terms of "probability of skill". Therefore, PSR indicates the probability of a given strategy to have a True SR greater than a given benchmark.

Comparing against zero skill, the evaluator sets the SR benchmark to 0 in order to compare against zero-investment skill. It allows the evaluator to give a confidence measure that the agent is a winner strategy (ie. it can generate "alpha").

Comparing against the benchmark UBAH strategy, the evaluator sets the SR benchmark to the historical Sharpe ratio of the benchmark strategy. The PSR then allows us to compare our trained agent directly against the benchmark strategy and gauge whether our trained strategy is significantly better than the benchmark strategy. Therefore the PSR against benchmark indicates the probability of the trained strategy to be the better strategy compared to the benchmark strategy.

Therefore, PSR is valuable also in difficult cases where we have to choose between multiple strategies with equally attractive Sharpe ratios, since it gives a confidence level around that num-

ber.

Chapter 3

Milestones and Deliverables

Assuming 30 hours of work per week and a total of 15 ECTS credits with 30 hours of workload per ECTS on average, this Master's project is expected to take 15 weeks. We are a group of 3 students with experience in working online and distributing workload appropriately in order to work in parallel.

- **Week 1-3:** Determining the scope of the project, literature search, finding similar financial frameworks, running a working minimum viable product using existing frameworks, researching readily available API components for utilities of the framework.
  - *Deliverables:*
    - Revised project proposal
    - API components document
    - Points of abstraction
    - Example code for a complete flow of portfolio management with RL and ML using existing frameworks
- **Week 4-5:** Further defining the scope of the utilities, flexibilities and improvements that the framework will offer, designing the interfaces to abstract away the framework components.
  - *Deliverables:*
    - First draft of the UML use case diagram
    - First draft of the UML class diagram
- **Week 6-7:** Improve and revise the existing use case and class diagrams according to feedback.
  - *Deliverables:*
    - Revised UML use case diagram
    - Revised UML class diagram
- **Milestone 1:** A clear API is defined in order to swiftly transition to the implementation of the portfolio management framework.
- **Week 8-11:** Implementation of the framework according to the class diagram.
  - *Deliverables:*
    - Implementation of the Financial Data Layer.

    * Implementation of the Agent Data Layer.
    * Implementation of the Evaluation Layer.

- **Milestone 2:** A fully functional portfolio management framework.
- **Week 12-13:** Testing, documenting, and demonstrating example use cases of the framework.
  - *Deliverables:*
    * Code documentation.
    * Example code for demonstrated use cases.
- **Milestone 3:** The framework is fully functional and well documented, with example use cases demonstrated.
- **Week 13-14:** Preparation of the presentation and the final report.
  - *Deliverables:*
    * Final report (Deliverables, difficulties, uml, implementation details, design).
    * Presentation (Motivation, what did we do, what can we do) 30 mins.
    * Extension example.
    * ReadMe.

3.1 Work Distribution

The work distribution has been done in a way that every student working on the project has equal amount of work as much as possible.

- **Barış Özakar :**
  - Designing the code architecture
  - UML Use case Diagram
  - API Components Document
  - Points of Abstraction Document
  - Implementation of Financial Data Layer (Feature Extractor and Data Downloader module), Evaluation Layer, and Agent Layer (Data Splitter Module)
  - Project Report
  - Example notebooks for demonstration
  - Deliverables Documentation
  - Presentation
- **Emine Didem Durukan :**
  - UML Class Diagrams
  - Framework Diagrams
  - Code Documentation
  - Implementation of Financial Data Layer (File Import Module) and Agent Layer (Conventional Agents and Metrics Modules)

  - Project Report
  - Presentation
  - README document

- **Doğan Parlak :**
  - UML Class Diagrams
  - Configuration of user parameters (.yaml file)
  - Implementation of Agent Layer (Deep Reinforcement Learning Agents, Conventional Agents and Environment Modules)
  - Project Report
  - Presentation
  - Example user Scenario (with Offline training)
  - README document

Chapter 4

Example User Scenario

Following is an example use case of the uniFi framework from the perspective of a researcher. The provided code snippet is divided into sections to demonstrate each part individually. One can run all of the lines together in a ".py" file.

4.1 Import user parameters

User parameters enables researchers to define the tickers with the relevant time interval, technical indicators and features to further consider, parameters to initialize, train and test the agents.

In figure 4.1, one can view an example structure of the parameters file.

```yaml
ENV_PARAMS:
  hmax: 100 # maximum number of shares to trade
  initial_amount: 1000000 # initial cash
  transaction_cost_pct: 0.001 # transaction cost perc
  state_space: 4 # number of unique stocks
  stock_dim: 4 # number of unique stocks
  feature_list: [
    "macd",
    "boll_ub",
    "boll_lb",
    "rsi_30",
    "cci_30",
    "dx_30",
    "close_30_sma",
    "close_60_sma",
    "close"
  ] # technical indicators
  action_space: 4 # number of stocks in training data
  reward_scaling: 0.1 # hyperparameter
```

Figure 4.1: Example structure of ".yaml" file

4.2 Financial Environment Layer

Researcher can choose between importing data from either using DataDownloader class which fetches the data from Yahoo API or importing data using CustomDatasetImporter. In each of these scenarios, utilizing the preferred time interval and tickers, data is stored and then processed by appending the feature engineering parameters to the dataset. This can be either achieved by DefaultFeatureEngineer class which appends the pre-defined techincal indicators or using CustomFeatureEngineer class which enables user to append any kind of feature to the data. This enables user a flexibility to amend the dataset.

Initially, 4.2 displays the initial form of data: After processing with DefaultFeatureEngineer

	date	open	high	low	close	volume	tic	day
0	2008-12-31	3.070357	3.133571	3.047857	2.606278	607541200	AAPL	2
1	2008-12-31	41.590000	43.049999	41.500000	32.005898	5443100	BA	2
2	2008-12-31	43.700001	45.099998	43.700001	30.628824	6277400	CAT	2
3	2008-12-31	72.900002	74.629997	72.900002	43.314434	9964300	CVX	2
4	2009-01-02	3.067143	3.251429	3.041429	2.771173	746015200	AAPL	4

Figure 4.2: Initial form of dataset

class 4.3 displays the final version of data:

	date	tic	open	high	low	close	volume	day	macd	boll_ub	boll_lb	rsi_30	cci_30	dx_30	close_30_sma	close_60_sma	cov_list	return_list
0	2010-12-31	AAPL	11.533929	11.552857	11.475357	9.849806	193508000.0	4.0	0.091907	9.973604	9.686130	58.973111	56.041751	4.076954	9.738818	9.566210	[[0.0002841359447857897, 0.000181404602563177...	tic AAPL BA CAT ...
0	2010-12-31	BA	64.900002	65.290001	64.620003	52.123985	2137400.0	4.0	-0.163485	53.158957	50.459630	49.173990	30.865840	6.670857	51.751254	53.392652	[[0.0002841359447857897, 0.000181404602563177...	tic AAPL BA CAT ...
0	2010-12-31	CAT	93.830002	93.900002	93.309998	68.765228	2542700.0	4.0	1.742875	70.643163	64.949738	68.265999	72.930050	45.749416	65.923380	62.360930	[[0.0002841359447857897, 0.000181404602563177...	tic AAPL BA CAT ...
0	2010-12-31	CVX	91.580002	91.800003	91.000000	57.546177	5152900.0	4.0	1.203227	58.549672	53.227386	65.105808	107.259721	53.406336	54.660562	53.596117	[[0.0002841359447857897, 0.000181404602563177...	tic AAPL BA CAT ...
1	2011-01-03	AAPL	11.630000	11.795000	11.601429	10.063869	445138400.0	0.0	0.101105	10.012176	9.684598	62.862118	142.931329	25.488753	9.760336	9.586746	[[0.0002849618209607939 5, 0.000180694510005855...	tic AAPL BA CAT ...

Figure 4.3: Processsed form of dataset

```
downloaded_df = DataDownloader(start_date=train_start,
                               end_date=trade_end,
                               ticker_list=tickers).download_from_yahoo()
data_processor = DefaultFeatureEngineer(**feature_eng_params)
# add technical indicators as features
df_processed = data_processor.extend_data(downloaded_df)
```

4.3 Agent Layer

In order to use data for training and testing, there are two options that user can implement. Either using TimeSeriesSplitter class to split the data according to given start and end times or use the

provided BlockingTimeSeries splitter to yield indices for validation.

The Agent layer provides the infrastructure for both offline and online training.

4.3.1 Offline Training

Training an agent offline only requires the users to provide the train and test set.

Following splits the data into train and test sets using TimeSeriesSplitter:

```
splitter = TimeSeriesSplitter()
train = splitter.get_split_data(df_processed, train_start, train_end)
trade = splitter.get_split_data(df_processed, trade_start, trade_end)
```

To be used by conventional agents, data has to be processed with given features and returns of the tickers has to be separated so that the training and return predictions stages can be completed.

Furthermore, there are several conventional agents the user can prefer. SVR is one of them. Use of other conventional agents are similar. The SVR agent is initialized, trained and tested as follows:

```
x_train, y_train = data_processor.prepare_ml_data(train)
svr = SVRAgent(**policy_params["SVR_PARAMS"])
svr.train_model(x_train, y_train, **train_params["SVR_PARAMS"])
SVR_portfolio_df, SVR_meta_coefficient = svr.predict(trade,
 ↪  **test_params["SVR_PARAMS"])
```

4.4 displays the portfolio value of SVR agent and 4.5 shows the allocation of tickers for each trading day:

	date	account_value
0	2020-07-01	1000000
1	2020-07-02	1002717.337037
2	2020-07-06	1042091.814716
3	2020-07-07	992014.183768
4	2020-07-08	998669.002421
...	...	...
290	2021-08-25	1365591.974686
291	2021-08-26	1358044.117188
292	2021-08-27	1367801.11403
293	2021-08-30	1409405.654901
294	2021-08-31	1397531.901162

295 rows × 2 columns

Figure 4.4: Portfolio value obtained by SVR agent for each trading day

On the other hand, one can choose a Reinforcement learning based agent to use for portfolio allocation. A2C is one of them. Use of other RL agents are all similar. In order to use a RL agent, one has to create an environment for it. This can be obtained either by inheriting the Environment abstract base class and writing your own environment class or utilizing the by default implemented PortfolioEnv class. A2C agent is initialized, trained and tested by using the environment from PortfolioEnv class.

```
# CREATE TRAIN ENV
env = PortfolioEnv(df=train, **env_kwargs)
```

date	AAPL	BA	CAT	CVX
2020-07-01	3.469447e-17	1.000000e+00	0.000000e+00	7.279765e-17
2020-07-02	8.180032e-17	1.000000e+00	0.000000e+00	0.000000e+00
2020-07-06	0.000000e+00	1.000000e+00	0.000000e+00	4.529124e-17
2020-07-07	1.415861e-17	1.000000e+00	1.110223e-16	4.935557e-17
2020-07-08	0.000000e+00	1.000000e+00	2.220446e-16	1.277815e-17
...	...	...	...	...
2021-08-24	1.000000e+00	0.000000e+00	1.006140e-15	0.000000e+00
2021-08-25	1.000000e+00	1.110223e-16	0.000000e+00	5.551115e-17
2021-08-26	1.000000e+00	0.000000e+00	1.812786e-16	0.000000e+00
2021-08-27	1.000000e+00	3.469447e-17	0.000000e+00	1.006140e-16
2021-08-30	1.000000e+00	3.677614e-16	1.387779e-17	0.000000e+00

294 rows × 4 columns

Figure 4.5: Allocation of tickers for each trading day by SVR agent

```
env_train, _ = env.get_env()
# CREATE TEST ENV
env_test = PortfolioEnv(df=trade, **env_kwargs)
# CREATE A2C AGENT
a2c = A2C(env=env_train, **policy_params["A2C_PARAMS"])
# TRAIN A2C AGENT
a2c.train_model(**train_params["A2C_PARAMS"])
# TEST A2C AGENT
A2C_portfolio_df, df_actions_a2c = a2c.predict(environment=env_test,
 ↪  **test_params["A2C_PARAMS"])
```

4.6 displays the portfolio value of A2C agent and 4.7 shows allocation of tickers for each trading day:

4.3.2 Online Training

Online training can be implemented by using the provided utility methods under TimeSeriesSplitter as such:

```
for i, train_day in enumerate(trade_period):
    if i==len(trade_period)-1:
        break
    else:
        trade = splitter.get_split_data(df_processed, train_day,
         ↪  splitter.get_next_df_date(df_processed, train_day))
        next_SVR_portfolio_df, next_SVR_meta_coefficient =
         ↪  svr.predict(trade, initial_capital=initial_capital,
         ↪  feature_list=feature_list)
        initial_capital=next_SVR_portfolio_df["account_value"][1]
```

A2C_portfolio_df

	date	account_value
0	2020-07-01	1.000000e+06
1	2020-07-02	1.004089e+06
2	2020-07-06	1.027394e+06
3	2020-07-07	1.008121e+06
4	2020-07-08	1.021519e+06
...	...	...
290	2021-08-25	1.525147e+06
291	2021-08-26	1.507172e+06
292	2021-08-27	1.525423e+06
293	2021-08-30	1.538191e+06
294	2021-08-31	1.530148e+06

295 rows × 2 columns

Figure 4.6: Portfolio value obtained by A2C agent for each trading day

```
        if i == 0:
            SVR_portfolio_df = pd.concat([SVR_portfolio_df.copy(),
            ↪ next_SVR_portfolio_df])
        else:
            SVR_portfolio_df = pd.concat([SVR_portfolio_df.copy(),
            ↪ next_SVR_portfolio_df.iloc[1:]])
        train = splitter.get_split_data(df_processed, train_start,
        ↪ splitter.get_next_df_date(df_processed, train_day))
        x_train, y_train = data_processor.prepare_ml_data(train,
        ↪ feature_list)
        sample_weight = [0.99**i for i in
        ↪ range(len(train.index.unique())-1)]
        sample_weight = sorted(sample_weight*len(tickers))
        svr.train_model(x_train, y_train, sample_weight=sample_weight)
```

During online training, the training set grows each day. Therefore, the user may wish to utilize a **sample weighting** scheme (linear or exponential decay) for the training samples such that recent trading days are weighed more heavily in training. The agent layer allows sample weights as shown above with an example of linear decay with constant 0.99.

4.4 Evaluation Layer

The Evaluation Layer provides the utility to produce the back-testing statistics and plots for multiple agents, allowing comparisons with respect to a benchmark. The evaluator can be initialized with the obtained predicted portfolio data frames. Backtest statistics and plots can be produced with the following code:

date	AAPL	BA	CAT	CVX
2020-07-01	0.250000	0.250000	0.250000	0.250000
2020-07-02	0.453611	0.212641	0.166874	0.166874
2020-07-06	0.453611	0.212641	0.166874	0.166874
2020-07-07	0.453611	0.212641	0.166874	0.166874
2020-07-08	0.453722	0.212449	0.166915	0.166915
...	...	...	...	...
2021-08-25	0.453611	0.212641	0.166874	0.166874
2021-08-26	0.453611	0.212641	0.166874	0.166874
2021-08-27	0.453611	0.212641	0.166874	0.166874
2021-08-30	0.453611	0.212641	0.166874	0.166874
2021-08-31	0.453611	0.212641	0.166874	0.166874

295 rows × 4 columns

Figure 4.7: Allocation of tickers for each trading day by A2C agent

```
extended_evaluator = ExtendedPortfolioEvaluator(SVR_portfolio_df,
    A2C_portfolio_df, agent_names=["SVR", "A2C"],
    benchmark_tickers=tickers)
extended_evaluator.backtest_stats()
extended_evaluator.backtest_plot()
```

Example backtest statistics can be seen in Fig. 4.8.

	Annual return	Cumulative returns	Annual volatility	Sharpe ratio	Calmar ratio	Stability	Max drawdown	Omega ratio	Sortino ratio	Skew	Kurtosis	Tail ratio	Daily value at risk	Prob. Sharpe (vs benchmark)	Prob. Sharpe (vs zero skill)
SVR	0.411331	0.134035	0.322191	1.229079	3.370066	0.873676	-0.122054	1.222906	1.890269	0.079819	0.002891	1.156142	-0.039021	0.773939	0.770278
A2C	0.027915	0.010102	0.247662	0.233741	0.243164	0.707341	-0.114798	1.038325	0.336161	-0.059509	-0.146259	1.094020	-0.030973	0.560598	0.555825
Uniform Buy and Hold	-0.033936	-0.012661	0.244376	-0.020136	-0.247399	0.649982	-0.137173	0.996715	-0.028208	-0.179254	0.163719	1.076835	-0.030808	0.500000	0.495146

Figure 4.8: Example Backtest Statistics

The evaluation layer produces several visualizations including but not limited to cumulative return, drawdown, and return distribution plots. These can be seen in Figures 4.9, 4.10, 4.11, 4.12, 4.13.

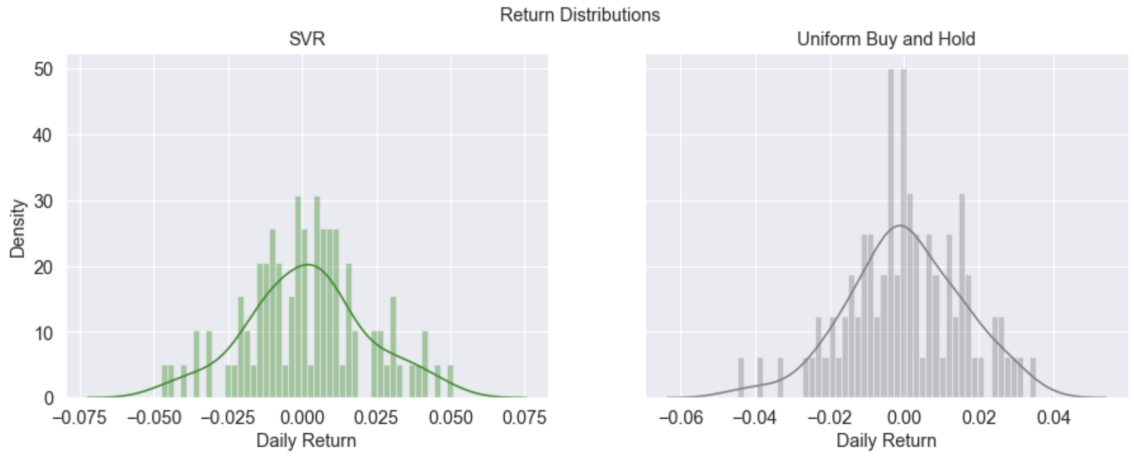


Figure 4.9: Example Backtest Return Histogram Plots

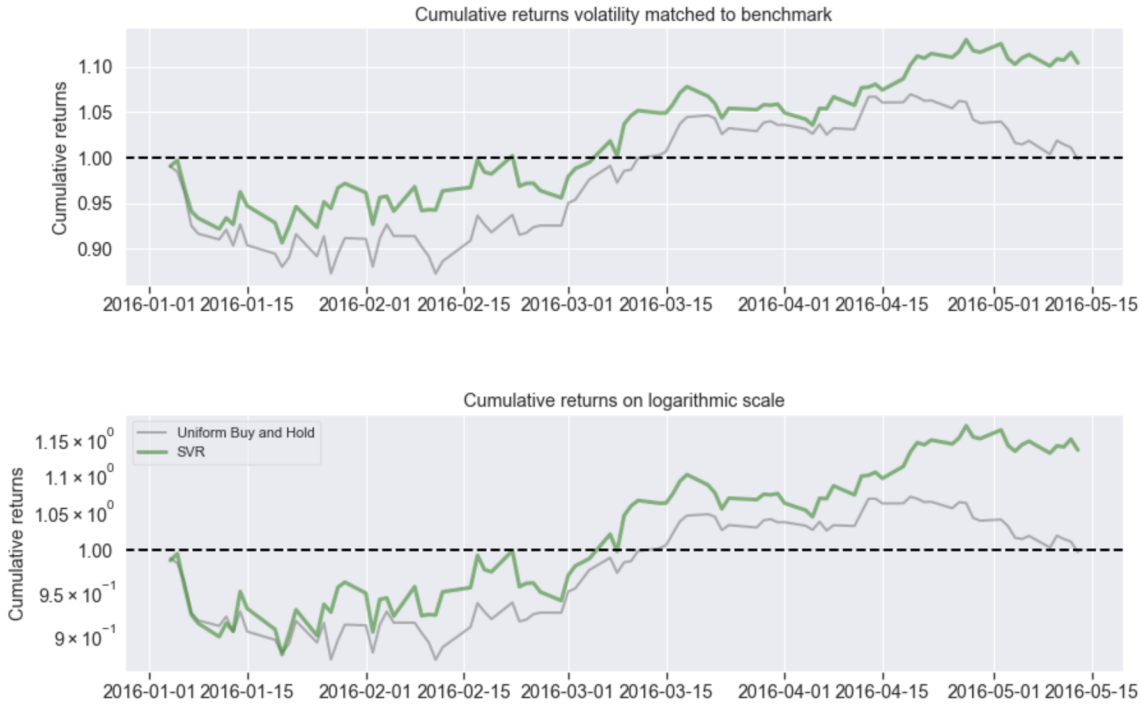


Figure 4.11: Example Backtest Cumulative Return Plots

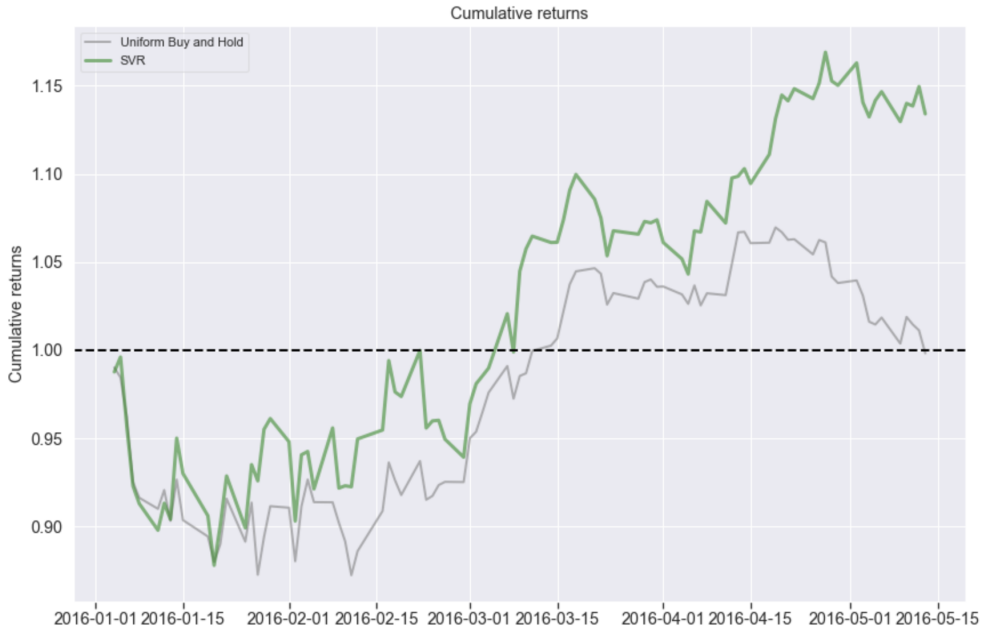


Figure 4.10: Example Backtest Cumulative Return Plot

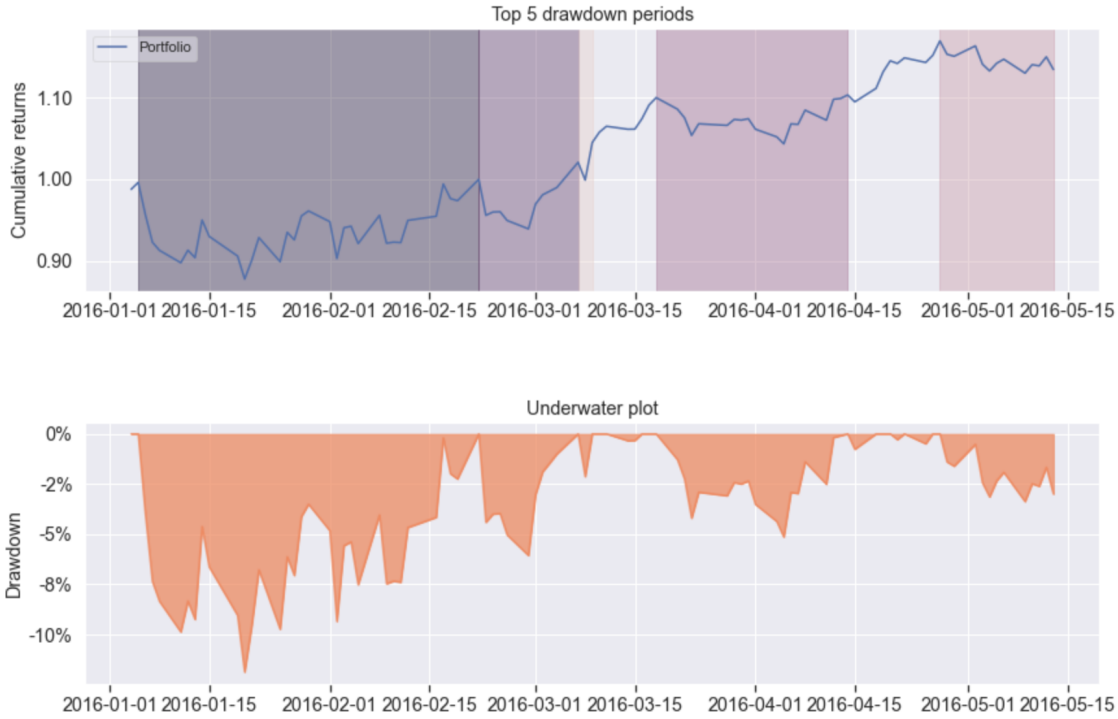


Figure 4.12: Example Backtest Drawdown Plots

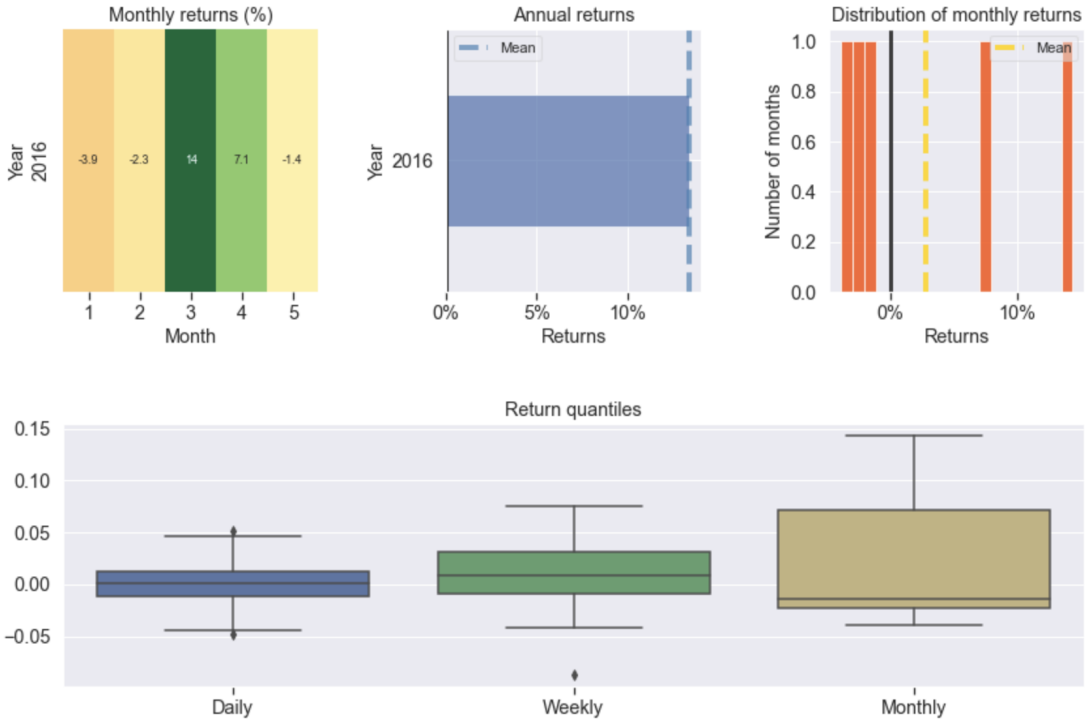


Figure 4.13: Example Backtest Return Distribution Plots

Chapter 5

Conclusion & Future Work

Given that there have been different approaches to tackle the portfolio management problem, being able to compare them is a basic need. Our research showed that there is no framework that creates an environment to compare these different methodologies in a unified way. With this motivation we introduced **"UniFi: A Unified Framework for Portfolio Management"**.

UniFi is a framework where researchers, scientists can compare their portfolio management strategies and methodologies in a unified manner, even though these approaches have tackled the problem in different ways. UniFi comes with some default implementations and practices to support users. Yet, due to its flexibility and its modular and sequential-layer-based structure, it can be tailored depending on the needs of the user. The users can integrate their own data collection methodology, implement their agents, and evaluation strategies as long as they follow the structure of the proposed framework.

With UniFi, users will be able to compare their portfolio management agents against sector standard benchmarks and/or against other strategies; and they can have a custom-made framework for each of their research projects.

The current state of the uniFi framework is suitable for it to be extended further. We encourage users to extend the utilities of the framework in multiple ways, including but not limited to:

- **Implementation of state of the art agents:** The UniFi framework is ready to accept researchers to develop state of the art portfolio management agents by implementing custom agent architectures and hyper-parameter tuning.
- **Increased variation of data downloading and processing functionalities:** The users may wish to implement other ways from which data can be fetched online instead of being limited to the Yahoo API. In addition, the variety of data splitting methods can be increased in order to provide multiple options for the user during validation and testing processes.
- **Implementation of higher frequency data download and processing capabilities:** The users may wish to work with higher frequency data instead of being limited to daily frequencies by modifying the data downloading and feature engineering functionalities.
- **Increased variation in conventional machine learning agents:** The users may wish to extend the available conventional agent variation (Support Vector Regressor, Linear Regressor, Huber Loss Regressor, Random Forest Regressor, Decision Tree Regressor)
- **Increased variation in deep reinforcement learning agents:** The users may wish to extend the available reinforcement learning agent variation (A2C, PPO, DDPG, TD3)
- **Increased variation in optimization objectives:** Conventional agent portfolio weight allocation optimization utilizes non-convex objectives including Sharpe ratio and transaction

costs. The users may wish to extend this by employing other objectives. Reinforcement learning agents focuses on optimizing the portfolio value while taking transaction costs into account through the reward function of the portfolio environment. In the Portfolio environment class, reward function can be taken as input rather than keeping it solely by portfolio value (while minimizing for transaction cost).

- **Implementation of other agent actions:** The allowed actions of the agents can be extended such that rather than allocating the entirety of the funds in the market, the option of cash allocation can be included. Moreover, shorting and leveraged trading could also be implemented.
- **Improving the defaults:** Default parameters of agents could be improved by an intricate validation workflow.

Appendix A

Technology Stack

Implementation of the UniFi framework is done with the Python programming language. Visual studio code is used as the IDE where Github is utilized to synchronize the work done and version control of the code. A .yaml file is used to store all configuration parameters. For each layer, the libraries used are listed below.

A.1 Financial Data Layer

- yfinance (yahoo finance) used to fetch raw data from API.

A.2 Agent Layer

- pypfopt, EfficientFrontier used to find the optimal weights given expected returns and covariances.
- pypfopt, risk_models used to calculate covariance of returns in return prediction process of Conventional Agents predict method.
- pypfopt, objective_functions used to add objectives in weight optimization step of conventional agents prediction method.
- sklearn.model_selection, TimeSeriesSplit used to implement data splitting in Data Splitter section.
- sklearn.svm, SVR used to implement the SVR agent.
- sklearn.tree, DecisionTreeRegressor used to implement the Decision Tree Regressor agent.
- sklearn.linear_model, HuberRegressor used to implement the Huber Regressor agent.
- sklearn.linear_model, LinearRegression used to implement the Linear Regression agent.
- sklearn.ensemble , RandomForestRegressor used to implement the Random Forest Regressor agent.
- stable_baselines3.common.vec_env, DummyVecEnv used to vectorize the environment.
- gym, spaces used to create the observation space in portfolio environment.

- stable_baselines3, A2C used to implement the A2C agent.
- stable_baselines3, DDPG used to implement the DDPG agent.
- stable_baselines3, PPO used to implement the PPO agent.
- stable_baselines3, TD3 used to implement the TD3 agent.

A.3 Evaluation Layer

- pyfolio used to create backtest plots.
- pyfolio, timeseries used to obtain benchmark statistics.
- seaborn used to create backtest plots in Custom Portfolio Evaluater.

A.4 Parameter File (.yaml) Structure

- TICKERS
- TRAIN_START_DATE
- TRAIN_END_DATE
- TRADE_START_DATE
- TRADE_END_DATE

A.4.1 FEATURE_ENG_PARAMS

- use_default
- use_vix
- use_return
- use_turbulence
- use_covar
- tech_indicator_list

A.4.2 ENV_PARAMS

- hmax
- initial_amount
- transaction_cost_pct
- state_space
- stock_dim

- feature_list
- action_space
- reward_scaling

A.4.3 TRAIN_PARAMS

SVR_PARAMS

- sample_weight

LR_PARAMS

- sample_weight

DT_PARAMS

- sample_weight
- check_input

HR_PARAMS

- sample_weight

RF_PARAMS

- sample_weight

A2C_PARAMS

- total_timesteps
- callback
- log_interval
- eval_env
- eval_freq
- n_eval_episodes
- tb_log_name
- eval_log_path
- reset_num_timesteps

PPO_PARAMS

- total_timesteps
- callback
- log_interval
- eval_env
- eval_freq
- n_eval_episodes
- tb_log_name
- eval_log_path
- reset_num_timesteps

DDPG_PARAMS

- total_timesteps
- callback
- log_interval
- eval_env
- eval_freq
- n_eval_episodes
- tb_log_name
- eval_log_path
- reset_num_timesteps

TD3_PARAMS

- total_timesteps
- callback
- log_interval
- eval_env
- eval_freq
- n_eval_episodes
- tb_log_name
- eval_log_path
- reset_num_timesteps

A.4.4 TEST_PARAMS

SVR_PARAMS

- initial_capital
- transaction_cost_pct
- feature_list

LR_PARAMS

- initial_capital
- transaction_cost_pct
- feature_list

DT_PARAMS

- initial_capital
- transaction_cost_pct
- feature_list

HR_PARAMS

- initial_capital
- transaction_cost_pct
- feature_list

RF_PARAMS

- initial_capital
- transaction_cost_pct
- feature_list

A2C_PARAMS

- state
- episode_start
- deterministic

PPO_PARAMS

- state
- episode_start
- deterministic

DDPG_PARAMS

- state
- episode_start
- deterministic

TD3_PARAMS

- state
- episode_start
- deterministic

A.4.5 POLICY_PARAMS

SVR_PARAMS

- kernel
- degree
- gamma
- coef0
- tol
- C
- epsilon
- shrinking
- cache_size
- verbose
- max_iter

LR_PARAMS

- fit_intercept
- copy_X
- positive

DT_PARAMS

- criterion
- splitter
- max_depth
- min_samples_split
- min_samples_leaf
- min_weight_fraction_leaf
- max_features
- random_state
- max_leaf_nodes
- min_impurity_decrease
- ccp_alpha

HR_PARAMS

- epsilon
- max_iter
- alpha
- warm_start
- fit_intercept
- tol

RF_PARAMS

- n_estimators
- criterion
- max_depth
- min_samples_split
- min_samples_leaf
- min_weight_fraction_leaf
- max_features
- max_leaf_nodes
- min_impurity_decrease
- bootstrap
- oob_score
- n_jobs
- random_state
- verbose
- warm_start
- ccp_alpha
- max_samples

A2C_PARAMS

- policy
- learning_rate
- n_steps
- gamma
- gae_lambda
- ent_coef
- vf_coef
- max_grad_norm
- rms_prop_eps
- use_rms_prop
- use_sde

- sde_sample_freq
- normalize_advantage
- tensorboard_log
- create_eval_env
- policy_kwargs
- verbose
- seed
- device
- _init_setup_model

PPO_PARAMS

- policy
- learning_rate
- n_steps
- batch_size
- n_epochs
- gamma
- gae_lambda
- clip_range
- clip_range_vf
- normalize_advantage
- ent_coef
- vf_coef
- max_grad_norm
- use_sde
- sde_sample_freq
- target_kl
- tensorboard_log
- create_eval_env
- policy_kwargs
- verbose
- seed
- device
- _init_setup_model

DDPG_PARAMS

- policy
- learning_rate
- buffer_size
- learning_starts
- batch_size
- tau
- gamma
- train_freq
- gradient_steps
- action_noise
- replay_buffer_class
- replay_buffer_kwargs
- optimize_memory_usage
- tensorboard_log
- create_eval_env
- policy_kwargs
- verbose
- seed
- device
- _init_setup_model

TD3_PARAMS

- policy
- learning_rate
- buffer_size
- learning_starts
- batch_size
- tau
- gamma
- train_freq

- gradient_steps
- action_noise
- replay_buffer_class
- replay_buffer_kwargs
- optimize_memory_usage
- tensorboard_log
- create_eval_env
- policy_kwargs
- verbose
- seed
- device
- _init_setup_model

List of Figures

Bibliography

(2022). Lookback period in trading (what is it? – optimal, best?).

Ayyadevara, V. K. (2018). Decision tree. In *Pro Machine Learning Algorithms*, pages 71–103. Springer.

Bailey, D. and Lopez de Prado, M. (2012). The sharpe ratio efficient frontier. *The Journal of Risk*, 15:3–44.

Dempster, M. A. and Leemans, V. (2006). An automated fx trading system using adaptive reinforcement learning. *Expert Systems with Applications*, 30(3):543–552.

Genuer, R., Poggi, J.-M., and Tuleau, C. (2008). Random forests: some methodological insights. *arXiv preprint arXiv:0811.3619*.

Haugen, R. A. (1986). Modern investment theory.

Heaton, J. B., Polson, N. G., and Witte, J. H. (2017). Deep learning for finance: deep portfolios. *Applied Stochastic Models in Business and Industry*, 33(1):3–12.

Huber, P. J. (1973). Robust regression: asymptotics, conjectures and monte carlo. *The annals of statistics*, pages 799–821.

Jiang, Z., Xu, D., and Liang, J. (2017). A deep reinforcement learning framework for the financial portfolio management problem. *arXiv preprint arXiv:1706.10059*.

Lillicrap, T. P., Hunt, J. J., Pritzel, A., Heess, N., Erez, T., Tassa, Y., Silver, D., and Wierstra, D. (2015). Continuous control with deep reinforcement learning. *arXiv preprint arXiv:1509.02971*.

Moody, J. and Saffell, M. (2001). Learning to trade via direct reinforcement. *IEEE transactions on neural Networks*, 12(4):875–889.

Schulman, J., Wolski, F., Dhariwal, P., Radford, A., and Klimov, O. (2017). Proximal policy optimization algorithms. *arXiv preprint arXiv:1707.06347*.

Smola, A. J. and Schölkopf, B. (2004). A tutorial on support vector regression. *Statistics and computing*, 14(3):199–222.

V. Mnih, K. Kavukcuoglu, D. S. (2015). Human- level control through deep reinforcement learning. *Nature*, 518.

Weisberg, S. (2005). *Applied linear regression*, volume 528. John Wiley & Sons.

Whaley, R. E. (2009). Understanding the vix. *The Journal of Portfolio Management*, 35(3):98–105.

Yang, B. (2022). Finrl for quantitative finance: Tutorial for single stock trading.

Zhang, T. (2020). Measuring financial turbulence and systemic risk.