**Stock Market Trader Bot**

Software Operation Manual

by Daniel Leal

Icon

Description automatically generated with low confidence

# IbTrading.py

## Connecting Code to TWS

Connecting to TWS begins by using its API ‘socket’ component that allows for external applications, such as a programming IDE, to trade through the TWS API. Within a given python file, the ibapi library is imported to utilize its EWrapper and EClient classes to initialize an IB instance and connect our code to the TWS API using the socket port and ensuring that the ‘Enable ActiveX and Socket Clients’ box is checked. The socket port and previously mentioned box can be found under:

File > Global Configuration > API > Settings

In order for code to be implemented to trading, the TWS application must be open and running while the code is simultaneously being compiled. In summary:

Step One: Open TWS API and log into account.

Step Two: Go to *File > Global Configuration > API > Settings*

Step Three: Ensure ‘**Enable ActiveX and Socket Clients’** box is checked.

## Executing Orders

When performing trades via python programming language, specific components are necessary such as: an *order* and a *contract*. An *order* defines the type of order (in our case it is a basic market order), the action (buy/sell), and the quantity of a given stock to perform that action on. A *contract* defines a stock, the exchange it is present in, and the currency to utilize for trading. Once these components are defined and the code is appropriately connected to TWS as mentioned in the previous section, the ibapi.order library has a placeOrder function that executes our buy/sell order. This placeOrder function requires parameter ‘orderId’ which is basically a unique ID that is reset every day on the IBKR’s end. This unique ID will simply be the current time that the functon is called. As implied, the only time orders will go through is during trading hours. In summary:

Step One: Follow Steps One through Three from ‘Connecting Code to TWS’ section.

Step Two: Call our IbFunctions class using something like ‘ibSession = IbFunctions()’ to connect TWS API to IBKR account.

Step Three: Call respective class functions using something like ‘ibSession.BuyOrder(‘GLD’, 10)’.

## Class IbTwsApi(EWrapper, EClient)

**Description**: This class aims to provide a connection to our TWS API by utilizing the ibapi library EWrapper and EClient classes to instantantiate a client-to-TWS connection.

### \_\_init\_\_(verbose=False)

**Description**: This function calls the constructor for the ibapi.EClient class that automates the connection to a TWS application.

**Returns**: Client instance.

## Class IbFunctions(object)

**Description**: This class aims to provide a connection to a IBKRPRO account by establishing a bridge from the TWS API to a IBKRPRO account. This class also includes functions to perform orders and a test example.

### \_\_init\_\_(verbose=False)

**Description**: This function serves as the constructor for the IbFunctions class that creates a connection to TWS API using the IbTwsApi class, then connects the TWS API to an IBKR account using the local IP, socket port, and client id, all while the TWS application is open. Finally, it will start a Thread for the ib account.

**Returns**: Connection to IBKR account from TWS API.

### BuyOrder(ticker, quantity)

**Description**: This function serves to execute a given number of buy orders (quantity) of a given ticker stock using contracts and order instances.

**Params**:

ticker (str): Stock acronym.

quantity (int): Number of stocks to buy.

**Returns**: Executed buy orders.

### SellOrder(ticker, quantity)

**Description**: This function serves to execute a given number of sell orders (quantity) of a given ticker stock using contracts and order instances.

**Params**:

ticker (str): Stock acronym.

quantity (int): Number of stocks to sell.

**Returns**: Executed sell orders.

### TestRun()

**Description**: This function serves test the IbFunctions class BuyOrder and SellOrder functionality by buying 54 apple stocks, selling 34 of them, buying 5 google stocks, and then selling 1 of them.

# Learners.py

## Class BollingerBandsLearner(object)

### BollingerBandsDf(ticker, startDate, endDate, window=8, stdVal=1.3)

**Description**: Preprocesses dataframe for trading.

**Params**:

ticker (str) – Acronym of stock.

startDate (str) – Start date of historical data in format YYYY-MM-DD.

endDate (str) – endDate of historical data YYYY-MM-DD.

window (int) – How often BB and SMA will be computed.

stdVal (float) – Size of standard deviation for bollinger bandwidth.

**Returns**: Single dataframe.

### BbSellPricesAndDates(checkDf)

**Description**: Gets the selling prices and dates based on Bollinger bands values.

**Params**:

checkDf (pd.DataFrame): Dataframe to calculate selling prices and dates.

**Returns**: Two arrays containing sell prices and sell dates.

### BbBuyPricesAndDates(checkDf)

**Description**: Gets the buying prices and dates based on Bollinger bands values.

**Params**:

checkDf (pd.DataFrame): Dataframe to calculate buying prices and dates.

**Returns**: Two arrays containing buy prices and buy dates.

### FormatTime(time)

**Description**: Formats time for visualization and results of Bollinger Bands Algorithm.

**Params**:

time (str) – Time to be formatted.

**Returns**: String of formatted time.

### BbTraderResults(tickerDf, ticker)

**Description**: Calculates results of Bollinger bands algorithm based on the buy and sell prices and dates.

**Params**:

tickerDf (pd.DataFrame) – Historical dataframe to calculate results from.

ticker (str) – Stock acronym.

**Returns**: All information of profits and total shares.

### BbVisualizeTrades(tickerDf, ticker)

**Description**: Plots the buys and sells on a candlestick graph.

**Params**:

tickerDf (pd.DataFrame) – Historical dataframe to calculate results from.

ticker (str) – Stock acronym.

**Returns**: Plot of candlestick along with buy and sell data points.

### StockTradeBb(ticker=‘GOOGL’, startDate=‘2022-01-01’, endDate=‘Today’, window=8, stdVal=1.3)

**Description**: Given a stock and time range, this function provides best purchase times based on Bollinger bands.

**Params**:

ticker (str) – Stock acronym.

startDate (str) – Start date of historical data in format YYYY-MM-DD.

endDate (str) – endDate of historical data YYYY-MM-DD.

window (int) – Window to be used for BB width computing.

stdVal (float) – Standard deviation value to be used for BB width.

**Returns**: Plot and information on profits and trade history.

## Class LinRegLearner(object)

**Description**: This class aims to provide a LinRegLearner instance where we create a linear regression algorithm of a given dataset.

### \_\_init\_\_(verbose=False)

**Description**: This function does nothin and serves as an empty constructor.

### AddEvidence(X, Y)

**Description:** This function serves to simply train an algorithm to use linear regression and calculate polynomial variable values of a best fit line given two datasets.

**Params:**

X (pd.DataFrame): Dataset X.

Y (pd.DataFrame): Dataset Y.

**Returns:** Model coefficients for a best fit polynomial or binomial line.

### Query(points)

**Description:** This function serves to predict a linear regression line given a set of datapoints.

**Params:**

points (pd.DataFrame): Dataframe containing predictive points.

**Returns:** Predicted model coefficients.

### GetLearnerInfo()

**Description**: This function serves to simply print out data from the learner.

**Returns:** Information on the learning algorithm.

## Class KNearestNeighborLearner(object)

**Description**: This class aims to provide a LinRegLearner instance where we create a linear regression algorithm of a given dataset.

### \_\_init\_\_(K=4, verbose=False)

**Description**: This is the constructor for the KNearestNeighborLearner class that simply initializes a k nearest neighbor Learner.

**Params**:

K (int): K nearest neighbor’s value.

verbose (bool): Print process or not.

**Returns**: Initializes variables.

### AddEvidence(X, Y)

**Description:** This function trains a knn learner when given training dataframes X and Y.

**Params:**

X (pd.DataFrame): Dataset X.

Y (pd.DataFrame): Dataset Y.

**Returns:** Variables designated to their respective class variables.

### Query(points)

**Description:** This function serves to predict a linear regression line given a set of datapoints.

**Params:**

points (pd.DataFrame): Dataframe containing predictive points.

**Returns:** Predicted model coefficients.

## Class DecisionTreeLearner(object)

**Description**: This class aims to provide a Decision Tree Learner instance where we create a decision tree algorithm of a given dataset.

### \_\_init\_\_(leafSize=1, verbose=False, tree=None)

**Description:** This function serves to initialize a Decision Tree Learner and all its respective variables.

**Params:**

leafSize (int): Maximum number of samples to be aggregated to a leaf.

verbose (bool): Print process or not.

tree (np.Array): Number of trees.

**Returns**: Initialized variables.

### BuildTree(X, Y)

**Description:** This function builds a decision tree using recursion by choosing the best column feature to split along with best value to split. Usually, the best feature has the highest correlation with data Y. If they are all the same however, then select the first feature. Typically, the best value to split is based on the median of the data according to its best determined feature.

**Params:**

X (np.Array): X values at every decision tree node.

Y (np.1DArray): Y values at every decision tree node.

**Returns:** A numpy NDArray that represents a tree.

### AddEvidence(X, Y)

**Description**: This function serves to add training data to the decision tree learner.

**Params**:

X (np.NDArray): X values of data to add.

Y (np.1DArray): Y training values.

**Returns**: Updated tree matrix for Decision Tree Learner.

### Query(points)

**Description:** This function serves to estimate a set of test points given a model we created. Basically, this is a test function for our model.

**Params:**

points (np.NDArray): Test queries.

**Returns:** Predictions in a numpy 1D array of estimated values.

## Class RandomTreeLearner(object)

**Description**: This class aims to provide a RandomTreeLearner instance where we create a random decision tree regression algorithm of a given dataset.

### \_\_init\_\_(leafSize=1, verbose=False, tree=None)

**Description:** This function serves to initialize a Random Tree Learner and all its respective variables.

**Params:**

leafSize (int): Maximum number of samples to be aggregated to a leaf.

verbose (bool): Print process or not.

tree (np.Array): Number of trees.

**Returns:** Initialized variables.

### BuildTree(data)

**Description:** This function builds a decision tree using recursion by choosing a

random column feature to split along with best value to split. Usually, the random

feature has does not perform as well as the DTL. If they are all the same however,

then we select the first feature. Typically, the best value to split is based on the median

of the data according to its best determined feature..

**Params:**

data(np.Array): X and Y values at every decision tree node.

**Returns:** A numpy NDArray that represents a tree.

### AddEvidence(X, Y)

**Description:** This function serves to add training data to the random tree learner.

**Params:**

X (np.NDArray): X values of data to add.

Y (np.1DArray): Y training values.

**Returns:** Updated tree matrix for Random Tree Learner.

### Query(points)

**Description:** This function serves to estimate a set of test points given a model we created. Basically, this is a test function for our model.

**Params:**

points (np.NDArray): Test queries.

**Returns:** Predictions in a numpy 1D array of estimated values.

### GetLearnerInfo()

**Description**: This function serves to simply print out data from the learner.

**Returns:** Information on the learning algorithm.

## Class BootstrapAggregatingLearner(object)

**Description**: This class aims to provide a Bootstrap Aggregating Learner instance where we create an algorithm that combines multiple algorithms of a given dataset!

### \_\_init\_\_(learner, bags=20, verbose=False, \*\*kwargs)

**Description:** This function serves to initialize a Boostrap Aggregating Learner and all its respective variables.

**Params:**

learner (object): LRL, DTL, or RTL.

bags (int): Quantity of learners to be trained.

boost (bool): Applies boosting.

verbose (bool): Print process or not.

\*\*kwargs: Additional arguments for the learner.

**Returns**: Initialized variables.

### AddEvidence(X, Y)

**Description**: This function serves to add training data to the bootstrap aggregating learner.

**Params**:

X (np.NDArray): X values of data to add.

Y (np.1DArray): Y training values.

**Returns**: Updated training data for BagLearner.

### Query(points)

**Description:** This function serves to estimate a set of test points given a model we created. Basically, this is a test function for our model.

**Params:**

points (np.NDArray): Test queries.

**Returns:** Predictions in a numpy 1D array of estimated values.

### GetLearnerInfo()

**Description:** This function serves to print out the data for the BagLearner.

**Returns:** Information on the learning algorithm.

## Class InsaneLearner(object)

**Description**: This class aims to provide an InsaneLearner instance where we create a learner that multiplies a bootstrap aggregating learner combined with any of the machine learning algorithms we created above by a defined number (20 by default) .

### \_\_init\_\_(learner=LinRegLearner, iterations=20, verbose=False, \*\*kwargs)

**Description:** This function serves to initialize an Insane Learner and all its respective variables.

**Params:**

learner (object): KNNL, LRL, DTL, or RTL.

verbose (bool): Print process or not.

\*\*kwargs: Additional arguments.

**Returns:** Initialized variables.

### AddEvidence(X, Y)

**Description:** This function serves to add training data to the insane learner.

**Params:**

X (np.NDArray): X values of data to add.

Y (np.1DArray): Y training values.

**Returns:** Updated tree matrix for Random Tree Learner.

### Query(points)

**Description:** This function serves to estimate a set of test points given a model we created. Basically, this is a test function for our model.

**Params:**

points (np.NDArray): Test queries.

**Returns:** Predictions in a numpy 1D array of estimated values.

## Class QLearner(object)

**Description**: This class aims to provide a Dyna-QLearner instance where we create a learner that is basically a conceptual algorithm that illustrates how real and simulated experience can be combined in building a policy.

### \_\_init\_\_(numOfStates=100, numOfActions=4, alpha=0.2, gamma=0.9, rar=0.5, radr=0.99, dyna=0, verbose=False)

**Description:** This function serves as the constructor for a Dyna Q Learner instance.

**Params:**

numOfStates (int): Number of states within a Q Table.

numOfActions (int): Number of actions within a Q Table.

alpha (float): Value for learning rate.

gamma (float): Value of future reward.

rar (float): Random action rate (Probability of selection a random action at each step).

radr (float): Random action decay rate (After each update, rar = rar \* radr).

dyna (int): Number of dyna updates.

verbose (bool): Display info or not.

**Returns:** Initialized variables.

### RememberQValues(state, action, reward, nextState, done)

**Description:** Allows for remember the Q values and appends to deque data structure.

**Params:**

state (int): State of Q table.

action (int): Action to perform for respective state.

reward (float): Reward for specific aciton.

nextState (int): Subsequent state of Q table.

done (bool): If q value acquisition is complete.

### Act(state, reward, done=False, update=True)

**Description:** Peforms a query operation depending on current status of Q table.

**Params:**

state (int): Current state to perform query on.

reward (float): Immediate reward from previous action.

done (bool): If acting has been performed.

update (bool): Update Q table based on values.

**Returns:** Query.

### QueryState(state)

**Description:** ind the next action to take in state s. Update the latest state and action without updating the Q table

**Params:**

state (int): New state.

**Returns:** Selected action take in state.

### Query(statePrime, reward, done=False)

**Description:** This function serves to find the next action to take in state statePrime amd update the latest state and action and the Q table. The update rule is defined as follows:

Q'[s, a] = (1 - α) · Q[s, a] + α · (r + γ · Q[s', argmax a'(Q[s', a'])]).

**Params:**

statePrime (int): New state.

reward (float): Immediate reward for taking the previous action.

**Returns:** Selected action take in statePrime.

## Class DeepQNetwork(object)

**Description**: This class aims to provide a Deep-Q Neural Network Learner instance where we create a learner that is basically a conceptual algorithm that illustrates how real and simulated experience can be combined in building a policy using daily return as its reward system.

### \_\_init\_\_( stateSize=20, actionSize=20, alpha=0.001, gamma=0.95, epsilon=1.0, minEpsilon=0.9, epsilonDecay=0.9, verbose=False)

**Description:** Constructor for deep nueral network q learner.

**Params:**

stateSize(int): Number of states.

actionSize (int): Number of actions.

alpha (float): Learning rate.

gamma (float): Value of future reward.

epsilon (float): Exploration rate.

minEpsilon (float): Minimum exploration rate.

epsilonDecay (float): Decay rate for exploration.

verbose (bool): Print info out.

**Returns:** Initialized variables.

### BuildModel()

**Description:** Builds the neural networks for our deep-q learning model

### Remember(self, state, action, reward, nextState, done)

**Description:** Appends information to memory to remember performance and values

**Params:**

state (int): Current state.

action (int): Current action.

reward (int): Current reward.

nextState (int): Next state.

done (bool): Completed or not

### Act(state)

**Description:** Makes decision on best action to perform based on state parameter.

**Params:**

state (int): Current state.

**Returns:** Action to perform

### RewardTarget(memory)

**Description:** Makes model understand what the best outcome is and reward algorithm accordingly.

**Params:**

Memory (deque): Model memory.

**Returns:** Target reward/action.

### Query(df)

**Description:** Fits model.

**Params:**

Df (pd.DataFrame): Data to be tested.

**Returns:** Trained model.

### AddEvidence(df, endDate, memory=[])

**Description:** Creates memory for training purposes.

**Params:**

Df (pd.DataFrame): Data to be tested.

endDate (string): End data of dataframe.

memory (deque): Memory.

**Returns:** Created memory.

### TransformDf(df, windowSize=2)

**Description:** Transforms dataframe into a tradeable one using daily returns for the DQN to use as reward system.

**Params:**

Df (pd.DataFrame): Data to be transformed.

windowSize (int): Window size for return computing.

**Returns:** Transformed dataframe.

### CreateTradesDf(df, learner)

**Description:** Creates a dataframe that contains the trade orders it recommends by date as its index.

**Params:**

Df (pd.DataFrame): Data to be appended with the orders.

Learner (DQN): This is the trained Deep-Q Nueral Network learner.

**Returns:** Dataframe with orders.

### PlotDqnPerformacne(df, testTrades, symbol)

**Description:** Visualizes the performance of the DQN based on the trades it recommended.

**Params:**

testTrades (pd.DataFrame): Contains the regular performance of the stock and the performance of the DQN.

Symbol (string): Stock symbol.

**Returns:** Saved plot.

## Class StrategyLearner(object)

**Description**: This class aims to provide a Dyna-Q Learner instance where we make our trades based off specified technical indicators in order to accustom the algorithm to working in a trading environment.

### \_\_init\_\_(self, accuracyBool=False, numOfShares=1, epochs=100, numOfSteps=10, impact=0.0, commission=0.0, verbose=False, learner=QLearner(numOfStates=3000, numOfActions=3))

**Description:** This function serves to create a StrategyLearner that can learn a trading policy.

**Params:**

numOfShares (int): Number of shares that can be traded in one order.

epochs (int): The number of times to train the learner.

numOfSteps (int): Steps in getting discretization thresholds.

impact (float): Difference between learner and actual data.

commision (float): Amount charged per transaction.

verbose (bool): Print info or not.

learner (object): Learner to implement strategy on.

**Returns:** Initialized variables.

### GetTechnicalIndicators(df)

**Description:** This function implements the technical indicators and features of a position and feeds it into the Q Learner.

**Params:**

df (pd.DataFrame): Dataframe to compute tech indicators.

**Returns:** A pandas dataframe of the technical indicators.

### GetThresholds(dfTechnicalIndicators, numOfSteps)

**Description:** Computes thresholds to be used for discretization and returns a 2-d numpy array where the first dimension indicates the index of features in dfTechnicalIndicators and second dimension refers to the value of a feature at the particular threshold.

### Discretize(dfTechnicalIndicators, nonNegativePosition, thresholds)

**Description:** This function serves to discretize the upcoming values of the deep q network. In applied mathematics, discretization is the process of transferring continuous functions, models, variables, and equations into discrete counterparts. This process is usually carried out as a first step toward making them suitable for numerical evaluation and implementation on digital computers.

**Params:**

dfTechnicalIndicators (pd.DataFrame): Dataframe with technical indicators.

nonNegativePosition (int): Positions of DQN.

thresholds (float): Threshold computed from previous function.

**Returns:** State in Q Table from which we query the action.

### GetPosition(prevPosition, signal)

**Description:** This function serves to find a new position based on the previous position and the given signal. Signal is the action that results from querying a state which comes from discretize in the q table. Action is either 0, 1, 2 and is the second index of the q table.

### GetDailyReward(prevPrice, currPrice, position)

**Description:** This function serves to calculate the daily reward of the dataframe as a percentage change in prices.

### CheckConverged(cumReturns, patience=10)

**Description:** This function serves to check if the cumulative returns has converged. Patience is the number of epochs with no improvements in cumulative returns. This will return either true or false.

### CreateDfTrades(orders, numOfShares, hold=0, buy=1, sell=-1)

**Description:** This function serves to simply create a dataframe for orders executed to simulate trading.

### CreateBmDfTrades(symbol, startDate, endDate, numOfShares)

**Description:** This function serves to simply createa an empty df that can be used to test against the dfTrades dataframe and establishes a benchmark.

### SymbolValueFromTrading(dfOrders, symbol, startDate, endDate, startVal=1, commision=9.95, impact=0.05)

**Description:** This function serves to simulate trading a stock based on the orders performed and symbol given. This returns a column of the portfolio value given that one stock after every action performed.

### MarketSimulator(dfOrders, dfOrdersBm, symbol, startDate, endDate, title, startVal=1, commission=9.95, impact=0.005,saveFigure=False, figName="Plot.png", showPlot=False)

**Description:** This function serves to mimic the market simulator project from ML4T university course by Tucker Balch. In summary, this function intakes dfOrders dataframe that executes the trades and displays the portfolio value respectively.

### AddEvidence(symbol='GLD', startDate="2021-01-01", endDate="2022-01-01", startVal=1)

**Description:** This function serves to add training data to the Strategy learner.

**Params:** self-explanatory**.**

**Returns:** Updated training data for Strategy learner.

### Query(symbol='GLD', startDate="2021-01-01", endDate="datetime.today().strftime('%Y-%m-%d')”)

**Description:** This function serves to test the existing policy on a new data set.

## Class DefeatLearners(object)

**Description**: This class aims to provide a DefeatLearners instance where we test our linear regression and decision tree learners on a predefined dataset from GenerateData class.

### \_\_init\_\_(verbose=False)

**Description**: This function does nothin and serves as an empty constructor.

### BestForLinRegLearner(seed=1489683273)

**Description**: This function serves to create a perfectly linear dataset in order to allow for our LinRegLearner to perform better, more accurate predictions than a decision tree learner.

**Params**:

Seed (int): Input seed value to repeat random number.

**Returns**: Optimized data for a linear regression learner.

### BestForDecisionTreeLearner(seed=1489683273)

**Description**: This function serves to create a dataset that is the complete opposite of linear in order to allow for our DecisionTreeLearner to perform better, more accurate predictions than a linear regression learner.

**Params**:

Seed (int): Input seed value to repeat random number.

**Returns**: Optimized data for a linear regression learner.

### CompareRmses(learnerOne, learnerTwo, X, Y)

**Description**: This function serves to simply compare the two learners..

**Params**:

learnerOne/learnerTwo (object): LRL, KNN, DTL, RTL, etc.

X (array): X axis values.

Y (array): Y axis values.

**Returns**: Rmses for both learners.

### TestDefeatLearners()

**Description**: This function serves to call our class methods to perform a prediction using the linear regression and decision tree algorithms and see which performs better on each of the datasets.

**Returns**: Printed description of the performance of each algorithm based on the fed dataset.

# Output.py

## Deep-Q Algorithm

### DqnAccuracyOutput(dfTrades)

**Description**: This function will interpret the dataframe it receives, check the last 30 predictions it made, and check if its predictions where accurate or not for the deep q neural network learner.

**Params**:

dfTrades (pd.DataFrame) – Dataframe of the trades.

**Returns**: A tuple with an accuracy dictionary with dates as keys and accurate/inaccurate as values and an accuracy list containing -1, -4, -20, etc.

### DqnOutput(symbol, verbose=False)

**Description**: This function will basically provide an order recommendation of hold/sell/buy based on a trained Deep-Q Neural Network Learner. It will output what the order recommendation is based on what specific it believes is best..

**Params**:

symbol (string): Symbol to provide a recommendation on.

**Returns**: Tuple of trading decision, testTrades dataframe, accuracy dictionary, and last date of the dataframe.

## Dyna-Q Algorithm

### StlAccuracyOutput(dfTrades)

**Description**: This function will interpret the dataframe it receives, check the last 30 predictions it made, and check if its predictions where accurate or not for the deep q neural network learner.

**Params**:

dfTrades (pd.DataFrame) – Dataframe of the trades.

**Returns**: A tuple with an accuracy dictionary with dates as keys and accurate/inaccurate as values and an accuracy list containing -1, -4, -20, etc.

### DynaQOutput(symbol, verbose=False)

**Description**: This function will basically provide an order recommendation of hold/sell/buy based on a trained Dyna-Q Learner those trades based on momentum, standard moving average, and bollinder bandwidth. It will output what the order recommendation is based on what specific computation.

**Params**:

symbol (string): Symbol to provide a recommendation on.

**Returns**: Tuple of trading decision, testTrades dataframe, accuracy dictionary, and last date of the dataframe.

# UserInterface.py

## Class SelectionInterface(customtkinter.CTkToplevel)

### \_\_init\_\_(self, symbol, stlDecision, stlLastDate, stlTrades, stlAccuracyDict, dqnDecision, dqnLastDate, dqnTrades, dqnAccuracyDict)

**Description**: This is the second pop-up window that allows you to select the learner.

**Params**:

Symbol (string) – Stock symbol.

stlDecision (string) – Decision for QLearnerInterface class.

stlLastDate (string) – Last date of the trading dataframe for the QLearnerInterface class.

stlTrades (pd.Dataframe) – Trading dataframe passed on to the QLearnerInterface class.

stlAccuracyDict (Dictionary) – Accuracy dictionary for the QLearnerInterface class.

dqnDecision (string) – Trading decision for the DqnLearnerInterface class.

dqnLastDate (string) – Last date of the trading dataframe for DqnLearnerInterface class.

dqnTrades (pd.DataFrame) – Trading dataframe passed on to DqnLearnerInterface class.

dqnAccuracyDict (Dictionary) – Accuracy dictionary for the DqnLearnerInterface class.

### AppearanceMode(mode)

**Description**: Sets the appearance mode for the user interface.

### CloseWindow(mode)

**Description**: Closes the window.

### OpenStlLearner(mode)

**Description**: Opens the QLearnerInterface window.

### OpenDqnLearner(mode)

**Description**: Opens the DqnLearnerInterface window.

## Class QLearnerInterface(customtkinter.CTkToplevel)

### \_\_init\_\_(self, symbol, decision, lastDate, dfTrades, accuracyDict)

**Description**: This is the window that pops up when you select the Dyna-Q algorithm from the second pop up.

**Params**:

Symbol (string) – Stock symbol.

decision (string) – Decision for QLearnerInterface class.

lastDate (string) – Last date of the trading dataframe for the QLearnerInterface class.

dfTrades (pd.Dataframe) – Trading dataframe passed on to the QLearnerInterface class.

accuracyDict (Dictionary) – Accuracy dictionary for the QLearnerInterface class.

### StlPredictionsButton()

**Description**: Displays the plot of the 30 recent predictions.

### StlBuyStock (symbol, quantity)

**Description**: Button action to buy the stock that is currently displaying its prediction for.

### StlSellStock (symbol, quantity)

**Description**: Button action to sell the stock that is currently displaying its prediction for.

### StlAppearanceLearner(mode)

**Description**: Modifies the windows appearance.

### StlCloseWindow()

**Description**: Closes the window.

## Class DqnLearnerInterface(customtkinter.CTkToplevel)

### \_\_init\_\_(self, symbol, decision, lastDate, dfTrades, accuracyDict)

**Description**: This is the window that pops up when you select the Deep-Q algorithm from the second pop up.

**Params**:

Symbol (string) – Stock symbol.

decision (string) – Decision for QLearnerInterface class.

lastDate (string) – Last date of the trading dataframe for the DqnLearnerInterface class.

dfTrades (pd.Dataframe) – Trading dataframe for the DqnLearnerInterface class.

accuracyDict (Dictionary) – Accuracy dictionary for the DqnLearnerInterface class.

### DqnPredictionsButton()

**Description**: Displays the plot of the 30 recent predictions.

### DqnBuyStock (symbol, quantity)

**Description**: Button action to buy the stock that is currently displaying its prediction for.

### DqnSellStock (symbol, quantity)

**Description**: Button action to sell the stock that is currently displaying its prediction for.

### DqnAppearanceLearner(mode)

**Description**: Modifies the windows appearance.

### DqnCloseWindow()

**Description**: Closes the window.

## Other Functions

### CreateSymbolSelectionInterface()

**Description**: This is the first pop-up window that allows you to input the stock acronym.

### RunMe()

**Description**: This is the function that combines the sequencing of pop up windows and connects the entire project into one function.

# Utilities.py

## Dataframes Functions

### IndividualHistoricalData(symbol, startDate, endDate=Today, keepAllColumns=“Yes”)

**Description**: Gets historical data from yahoo finance of one stock.

**Params**:

symbol (str) – Acronym of stock.

startDate (str) – Start date of historical data in format YYYY-MM-DD.

endDate (str) – endDate of historical data YYYY-MM-DD.

keepAllColumns (str) – ‘Yes’ or ‘No.’

**Returns**: Single dataframe.

### HistoricalData(symbolsArray, startDate, endDate=Today, keepAllColumns = “Yes”)

**Description**: Gets historical dataframes from yahoo finance given a time range.

**Params**:

symbolsArray (list) – Array of stock acronyms.

startDate (str) – Start date of historical data in format YYYY-MM-DD.

endDate (str) – End date of historical data YYYY-MM-DD.

keepAllColumns (str) – ‘Yes’ or ‘No.’

**Returns**: List of dataframes.

### NormalizeDfs(dfArray)

**Description**: Normalizes all dataframes inputted.

**Params**:

dfArray (pd.DataFrame or list) – single dataframe or array of dataframes.

**Returns**: List of dataframes.

### CombineDfs(dfArray, startDate, endDate=Today)

**Description**: Combines dataframes.

**Params**:

dfArray (list) – Array of dataframes to be combined.

startDate (str) – Start date of combined dataframe in format YYYY-MM-DD.

endDate (str) – End date of combined dataframe in format YYYY-MM-DD.

**Returns**: Dataframe.

### StockReturns(dfArray, dailyOrMonthly=‘daily’)

**Description**: Computes the daily, or monthly, return of all dataframes inputted.

**Params**:

dfArray (pd.DataFrame or list) – Single dataframe or array of dataframes.

dailyOrMonthly (str) – ‘daily’ or ‘monthly.’

**Returns**: List of dataframes.

### **Cum**ulativeReturns(dfArray, dailyOrMonthly=‘daily’)

**Description**: Computes the daily, or monthly, cumulative returns of all dataframes inputted.

**Params**:

dfArray (pd.DataFrame or list) – Single dataframe or array of dataframes.

dailyOrMonthly (str) – ‘daily’ or ‘monthly.’

**Returns**: List of dataframes.

## Statistical Functions

### GetMaxClose(dfArray)

**Description**: Gets the max closing prices of a stock or set of stocks.

**Params**:

dfArray (pd.DataFrame or list) – Single dataframe or array of dataframes.

**Returns**: Dict (e.g., symbol: MaxPrice).

## Slice Functions

### SliceRow(dfArray, startDate, endDate)

**Description**: Slices dataframes based on specific rows.

**Params**:

dfArray (pd.DataFrame or list) – Single dataframe or array of dataframes.

startDate (str) – Start date of sliced dataframe in format YYYY-MM-DD.

endDate (str) – End date of sliced dataframe in format YYYY-MM-DD.

**Returns**: List of dataframes.

### SliceColumn(dfArray, colArray)

**Description**: Slices dataframes based on specific columns.

**Params**:

dfArray (pd.DataFrame or list) – Single dataframe or array of dataframes.

colArray (str or list) – Single string or list of strings of column names.

**Returns**: List of dataframes.

## Validation Functions

### ValidateDates(startDate, endDate)

**Description**: Checks if startDate to endDate is a valid timeframe.

**Params**:

startDate (str) – Start date in format YYYY-MM-DD.

endDate (str) – End date in format YYYY-MM-DD.

**Returns**: Invalid or Valid output message.

## Plot Functions

### PlotData(dfArray, title=“X and Y.”, x=“X-Axis”, y =“Y-Axis”)

**Description**: Plots any inputted dataframes on one graph.

**Params**:

dfArray (pd.DataFrame or list) – Single dataframe or array of dataframes.

title (str) – Title for graph.

x (str) – Label for x-axis.

y (str) – Label for y-axis.

**Returns**: One graph.

### PlotRollingMean(dfArray, window=8)

**Description**: Plots rolling mean along with standard plot of all inputted dataframes. Will print out an individual graph for each dataframe.

**Params**:

dfArray (pd.DataFrame or list) – Single dataframe or array of dataframes

window (int) – Frequency of mean computation.

**Returns**: Individual graph for each dataframe inputted.

### PlotBollingerBands(dfArray, window=8, stdVal=1.3)

**Description**: Plots standard plot, Bollinger bands, and moving average of all dataframes inputted.

**Params**:

dfArray (pd.DataFrame or list) – Single dataframe or array of dataframes.

window (int) – Frequency of mean computation.

stdVal (float) – Decides bandwidth of bands.

**Returns**: Individual graph for each dataframe inputted.

### PlotHistogram(dfArray, plotStatisticsYesOrNo=‘No’, bin=10)

**Description**: Plots histograms of multiple dataframes at once. If less than two dataframes is inputted, statistics such as mean, standard deviation, and kurtosis can be printed.

**Params**:

dfArray (pd.DataFrame or list) – Single dataframe or array of dataframes.

plotStatisticsYesOrNo (str) – ‘yes’ or ‘no.’

bin (int) – Frequency in which bars are computed.

**Returns**: One graph with all dataframes in it.

### PlotScatter(dfArray, title=“Scatter Plot.”)

**Description**: Plots a scatter plot of two dataframes.

**Params**:

dfArray (pd.DataFrame) – Single, combined dataframe.

Title (str) – Title of plot.

**Returns**: Scatter plot with a dataframe symbol represented on each axis and beta/alpha values for best fit line.

### PlotCorrelationMatrix(dfArray)

**Description**: Plots a correlation matrix of all dataframes inputted.

**Params**:

dfArray (pd.DataFrame) – Single, combined dataframe.

**Returns**: One correlation matrix image.

### PlotAccurateData(accurateData, saveFigure=True, showPlot=False, figName="AccuracyPlot.png")

**Description**: Plots the accurate data used in the user interface of the project.

**Params**:

accurateData (list) – Containing -1 for yesterday, etc for accurate data.

saveFigure(bool) – Whether or not you want to save the figure.

showPlot(bool) – Whether or not you want to visualize the results of the plot.

figName (str) – Name of the saved figure..

### PlotDqnIteration(tempTrades, symbol, epoch)

**Description**: Plots the results of the DQN algorithm within the specific epoch.

**Params**:

tempTrades (pd.DataFrame) – Trades data frame.

Symbol (string) – Stock symbol.

Epoch (int) – Epoch/iteration count.

**Returns**: Visual of the performance based on the trades df.

## Portfolio Functions

### ComputePortfolioValue(startValue=1, startDate, endDate, symbols, allocations, statistics=True)

**Description**: Computes the value of a portfolio over a given time frame.

**Params**:

startValue (int or float) – Value in dollars of total invested in portfolio.

startDate (str) – Start date of portfolio investment in format YYYY-MM-DD.

endDate (str) – End date of portfolio investment in format YYYY-MM-DD.

symbols (list) – Array of symbols in portfolio.

allocations (list) – Array of allocated numbers of representing percentage of startValue

per respective symbol. When elements of array are added together, they must equal to 1.

**Returns**: Dataframe.

### ComputeSharpeRatio(historicalData, k=252)

**Description**: Computes either the annual, weekly, or daily sharpe ratio.

**Params**:

historicalData (pd.DataFrame or list) – Single historical dataframe or array of historical

dataframes.

startDate (str) – Start date in format YYYY-MM-DD.

endDate (str) – End date in format YYYY-MM-DD.

**Returns**: List of symbols and sharpe ratio.

### MoneCarloPortfolioOptimization(symbols, numOfSim, startDate, endDate)

**Description**: Computes the best set of allocations to effectively optimize portfolio profits using Monte Carlo Simulations.

**Params**:

Symbols (list) – List of strings of the stock symbol acronyms.

numOfSim (int) – Number of simulations.

startDate (str) – Start date in format YYYY-MM-DD.

endDate (str) – End date in format YYYY-MM-DD.

**Returns**: List of optimized allocations.

### ScipyOptimizePortfolio(symbols, startDate, endDate)

**Description**: Computes the best set of allocations to effectively optimize portfolio profits using Scipy.optimize.minimize function.

**Params**:

Symbols (list) – List of strings of the stock symbol acronyms.

startDate (str) – Start date in format YYYY-MM-DD.

endDate (str) – End date in format YYYY-MM-DD.

**Returns**: List of optimized allocations.

# UtilitiesTestsAndExamples.py

## Dataframes Functions Unit Tests

### IndividualHistoricalDataUnitTests()

**Description**: Tests the IndividualHistoricalData function of TraderBot.py.

**Test Case 1 (SUCCESS)**: Historical dataframe of Microsoft stock from 01-01-2020 to 09-09-2021.

**Test Case 2 (ERROR)**: Symbols variable error.

**Test Case 3 (ERROR)**: TimeRange variable error.

### HistoricalDataUnitTests()

**Description**: Tests the HistoricalData function of TraderBot.py.

**Test Case 1 (SUCCESS)**: List of historical dataframes for Apple, Microsoft, Twitter, IBM, and Amazon from 01-01-2020 to 09-09-2021.

**Test Case 2 (ERROR)**: TimeRange variable error.

**Test Case 3 (ERROR)**: SymbolsArray variable error.

### NormalizeDfsUnitTests()

**Description**: Tests the NormalizeDfs function of TraderBot.py.

**Test Case 1 (SUCCESS)**: List of normalized dataframes of the historical data of Apple, Microsoft, Twitter, IBM, and Amazon from 01-01-2020 to 09-09-2021.

**Test Case 2 (ERROR)**: DfArray variable error.

### CombineDfsUnitTests()

**Description**: Tests the CombineDfs function of TraderBot.py.

**Test Case 1 (SUCCESS)**: Single combined dataframe of the historical data of Apple, Microsoft, Twitter, IBM, and Amazon from 01-01-2020 to 09-09-2021.

**Test Case 2 (ERROR)**: TimeRange variable error.

### StockReturnsUnitTests()

**Description**: Tests the StockReturns function of TraderBot.py.

**Test Case 1 (SUCCESS)**: List of daily returns dataframes for respective normalized dataframes from the historical data of Apple, Microsoft, Twitter, IBM, and Amazon from 01-01-2020 to 09-09-2021.

**Test Case 2 (SUCCESS)**: List of monthly returns dataframes for respective normalized dataframes from the historical data of Apple, Microsoft, Twitter, IBM, and Amazon from 01-01-2020 to 09-09-2021.

**Test Case 3 (ERROR)**: DfArray variable error.

**Test Case 4 (ERROR)**: DailyOrMonthly variable error.

### CumulativeReturnsUnitTests()

**Description**: Tests the CumulativeReturns function of TraderBot.py.

**Test Case 1 (SUCCESS)**: List of daily cumulative returns dataframes for respective normalized dataframes from the historical data of Apple, Microsoft, Twitter, IBM, and Amazon from 01-01-2020 to 09-09-2021.

**Test Case 2 (SUCCESS)**: List of monthly cumulative returns dataframes for respective normalized dataframes from the historical data of Apple, Microsoft, Twitter, IBM, and Amazon from 01-01-2020 to 09-09-2021.

**Test Case 3 (ERROR)**: DailyOrMonthly variable error.

## Statistical Functions Unit Tests

### GetMaxCloseUnitTests()

**Description**: Tests the GetMaxClose function of TraderBot.py.

**Test Case 1 (SUCCESS)**: Dictionary of maximum closing price values for Apple, Microsoft, Twitter, IBM, and Amazon from 01-01-2020 to 09-09-2021.

**Test Case 2 (ERROR)**: DfArray variable error.

## Slice And Validate Functions Unit Tests

### SliceRowUnitTests()

**Description**: Tests the SliceRow function of TraderBot.py.

**Test Case 1 (SUCCESS)**: List of sliced dataframes of Apple, Microsoft, Twitter, IBM, and Amazon from 01-01-2020 to 01-31-2020 of original dataframes of Apple, Microsoft, Twitterm IBM, and Amazon from 01-01-2020 to 09-09-2021.

**Test Case 2 (ERROR)**: DfArray variable error.

**Test Case 3 (ERROR)**: TimeFrame variable error.

### SliceColumnUnitTests()

**Description**: Tests the SliceColumnAndRow function of TraderBot.py.

**Test Case 1 (SUCCESS)**: List of sliced dataframes for columns Apple and Microsoft of combined historical dataframes of Apple, Microsoft, Twitter, IBM, and Amazon from 01-01-2020 to 09-09-2021.

**Test Case 2 (ERROR)**: DfArray variable error.

**Test Case 3 (ERROR)**: ColArray variable error.

### ValidateDatesUnitTests()

**Description**: Tests the ValidateDates function of TraderBot.py.

**Test Case 1 (SUCCESS)**: Timeframe from 01-01-2021 to 12-31-2021..

**Test Case 2 (ERROR)**: Timeframe error from 01-22-2022 to 01-01-2000.

## Plot Functions Unit Tests

### PlotDataUnitTests()

**Description**: Tests the PlotData function of TraderBot.py.

**Test Case 1 (SUCCESS)**: Graph of normalized historical dataframes for Apple, Microsoft, Twitter, IBM, and Amazon from 01-01-2020 to 09-09-2021 with “Stock Prices” as graph title, “Date” as the x-axis, and “Price” as the y-axis.

**Test Case 2 (ERROR)**: Title variable error.

**Test Case 3 (ERROR)**: X-Label variable error.

**Test Case 4 (ERROR)**: Y-Label variable error.

### PlotRollingMeanUnitTests()

**Description**: Tests the PlotRollingMean function of TraderBot.py.

**Test Case 1 (SUCCESS)**: Individual graphs of the rolling mean for the normalized historical dataframes for Apple, Microsoft, Twitter, IBM, and Amazon from 01-01-2020 to 09-09-2021 with a window value of 10.

**Test Case 2 (ERROR)**: DfArray variable error.

**Test Case 3 (ERROR)**: Window variable error.

### PlotBollingerBandsUnitTests()

**Description**: Tests the PlotBollingerBands function of TraderBot.py.

**Test Case 1 (SUCCESS)**: Individual graphs of the Bollinger bands for the normalized historical dataframes of Apple, Microsoft, Twitter, IBM, and Amazon from 01-01-2020 to 09-09-2021 with a window value of 15.

**Test Case 2 (ERROR)**: DfArray variable error.

**Test Case 3 (ERROR)**: Window variable error.

### PlotHistogramUnitTests()

**Description**: Tests the PlotHistogram function of TraderBot.py.

**Test Case 1 (SUCCESS)**: Single histogram plot of the daily stock returns for Apple, Microsoft, Twitter, IBM, and Amazon from 01-01-2020 to 09-09-202 without the statistical lines displayed and a bin value of 10.

**Test Case 2 (SUCCESS)**: Single histogram plot of the daily stock returns for Apple and Twitter from 01-01-2020 to 09-09-202 with the statistical lines displayed and a bin value of 10.

**Test Case 3 (ERROR)**: DfArray variable error.

**Test Case 4 (ERROR)**: PlotStatisticsYesOrNo variable error.

**Test Case 5 (ERROR)**: Bin variable error.

### PlotScatterUnitTests()

**Description**: Tests the PlotScatter function of TraderBot.py.

**Test Case 1 (SUCCESS)**: Single scatterplot of the daily stock return historical dataframes for Apple on the x-axis and Twitter on y-axis from 01-01-2020 to 09-09-2021.

**Test Case 2 (ERROR)**: CombineDfs error.

**Test Case 3 (ERROR)**: DfArray variable error.

### PlotCorrelationMatrixUnitTests()

**Description**: Tests the PlotCorrelationMatrix function of TraderBot.py.

**Test Case 1 (SUCCESS)**: Graph of correlation matrix for the list of daily stock returns dataframes from the historical data for Apple, Microsoft, Twitter, IBM, and Amazon from 01-01-2020 to 09-09-2021.

**Test Case 2 (ERROR)**: CombineDfs error.

## Portfolio Functions Unit Tests

### ComputePortfolioValueUnitTests()

**Description**: Tests the ComputePortfolioValue function of TraderBot.py.

**Test Case 1 (SUCCESS)**: Single dataframe of the portfolio value with 100 total dollars of 5% Exxib Mobil, 10% General Electric, 5% JP Morgan, 10% BP, 15% Amazon, 25% Microsoft, 15% Apple, and 15% Google from 01-01-2020 to 09-09-2021.

**Test Case 2 (ERROR)**: StartVal variable error.

**Test Case 3 (ERROR)**: TimeFrame variable error.

**Test Case 4 (ERROR)**: Symbols variable error.

**Test Case 5 (ERROR)**: Allocations variable error.

### ComputeSharpeRatioUnitTests()

**Description**: Tests the ComputeSharpeRatio function of TraderBot.py.

**Test Case 1 (SUCCESS)**: List of respective symbols and daily sharpe ratios for Exxon Mobil, General Electric, JP Morgan, BP, Amazon, Microsoft, Apple, and Google from 01-01-2020 to 09-09-2021 where k value is 252 to represent daily computing.

**Test Case 2 (SUCCESS)**: List of respective symbols and weekly sharpe ratios for Exxon Mobil, General Electric, JP Morgan, BP, Amazon, Microsoft, Apple, and Google from 01-01-2020 to 09-09-2021 where k value is 52 to represent weekly computing.

**Test Case 3 (SUCCESS)**: List of respective symbols and monthly sharpe ratios for Exxon Mobil, General Electric, JP Morgan, BP, Amazon, Microsoft, Apple, and Google from 01-01-2020 to 09-09-2021 where k value is 12 to represent monthly computing.

**Test Case 4 (ERROR)**: DfArray variable error.

**Test Case 5 (ERROR)**: K variable error.

**Test Case 6 (ERROR)**: TimeFrame variable error.

### MonteCarloPortfolioOptimizationUnitTests()

**Description**: Tests the MonteCarloPortfolioOptimization function of TraderBot.py.

**Test Case 1 (SUCCESS)**: Graphs the original portfolio value of 5% Exxib Mobil, 10% General Electric, 5% JP Morgan, 10% BP, 15% Amazon, 25% Microsoft, 15% Apple, and 15% Google from 01-01-2020 to 09-09-2021 in red and the optimized portfolio allocations using Monte Carlo simulations in green.

### ScipyOptimizePortfolioUnitTests()

**Description**: Tests the ScipyOptimizePortfolio function of TraderBot.py.

**Test Case 1 (SUCCESS)**: Graphs the original portfolio value of 5% Exxib Mobil, 10% General Electric, 5% JP Morgan, 10% BP, 15% Amazon, 25% Microsoft, 15% Apple, and 15% Google from 01-01-2020 to 09-09-2021 in red and the optimized portfolio allocations using scipy.optimize.minimize function on the inverse sharp ratio function in green.

## Professor Requested Example

### ML4TProfessorExample()

**Description**: Utilizes TraderBot.py ComputePortfolioValue, MonteCarloPortfolioOptimization, and ScipyOptimizePortfolio functions to plot a two differently optimized portfolios along with the SPY graph for comparison to imitate the ML4T udacity course example.

**Video**: #2 “The difference optimization can make” of lesson ten.

**Timestamp**: 0:48

### ProfessorComparisonExample()

**Description**: Utilizes TraderBot.py ComputePortfolioValue, HistoricalData, CombineDfs, CumulativeReturns, PlotData, StockReturns, and ComputeSharpeRatio functions to mimic the professor’s validation jupyter code using our unique functions.

## Lesson Seven Examples

### SevenExampleOne()

**Description**: Utilizes TraderBot.py PlotData and HistoricalData functions to plot the SPY stock graph from 2009 to 2013 to imitate the ML4T udacity course example.

**Video**: #5 “How to plot a histogram.”

**Timestamp**: 0:51

### SevenExampleTwo()

**Description**: Utilizes TraderBot.py PlotData, StockReturns, and HistoricalData functions to plot the daily returns graph for SPY from 2009 to 2013 to imitate the ML4T udacity course example.

**Video**: #5 “How to plot a hisogram.”

**Timestamp**: 0:55

### SevenExampleThree()

**Description**: Utilizes TraderBot.py PlotHistogram, StockReturns, and HistoricalData functions to plot the daily returns histogram of SPY from 2009 to 2013 with bin value of 20 to imitate the ML4T udacity course.

**Video**: #5 “How to plot a hisogram.”

**Timestamp**: 1:14

### SevenExampleFour()

**Description**: Utilizes TraderBot.py PlotHistogram, StockReturns, and HistoricalData functions to plot the daily returns histogram for the SPY from 2009 to 2013 with its statistically computed lines using bin value 20 to imitate the ML4T udacity course.

**Video**: #6 “Computing histogram statistics.”

**Timestamp**: 2:00

### SevenExampleFive()

**Description**: Utilizes TraderBot.py PlotHistogram, StockReturns, and HistoricalData functions to plot the daily returns histogram for the SPY and Exxon Mobil from 2009 to 2013 using bin value 20 to imitate the ML4T udacity course.

**Video**: #8 “Plot two histograms together.”

**Timestamp**: 1:13

### SevenExampleSix()

**Description (First Plot)**: Utilizes TraderBot.py PlotScatter, CombineDfs, StockReturns, and HistoricalData functions to plot the daily returns histogram for the SPY and Exxon Mobil on each respective axis from 2009 to 2013 to imitate the ML4T udacity course.

**Video**: #13 “Scatterplots in python.”

**Timestamp**: 3:36

**Description (Second Plot)**: Utilizes TraderBot.py PlotScatter, CombineDfs, StockReturns, and HistoricalData functions to plot the daily returns histogram for the SPY and Gold on each respective axis from 2009 to 2013 to imitate the ML4T udacity course.

**Video**: #13 “Scatterplots in python.”

**Timestamp**: 4:25

### SevenExampleSeven()

**Description**: Utilizes TraderBot.py PlotScatter and HistoricalData functions to plot the historical data of SPY, Exxon Mobil, and Gold from 2009 to 2013 where “Stock Prices” is the title, “Date” is the x-axis, and “Price” is the y-axis to imitate the ML4T udacity course.

**Video**: #13 “Scatterplots in python.”

**Timestamp**: 3:41

### SevenExampleEight()

**Description**: Utilizes TraderBot.py PlotCorrelationMatrix, CombineDfs, StockReturns, and HistoricalData functions to plot the correlation amongst the daily returns of the historical data of SPY, Exxon Mobil, and Gold from 2009 to 2013 to imitate the ML4T udacity course.

**Video**: #13 “Scatterplots in python.”

**Timestamp**: 4:20

## Lesson Nine Examples

### NineExampleOne()

**Description**: Utilizes f(x) and scipy.optimize.minimize functions to calculate the minima of a given function to imitate the ML4T udacity course.

**Video**: #3 “Minimizer in Pyton.”

**Timestamp**: 2:13

### NineExampleTwo()

**Description**: Utilizes f(x) and scipy.optimize.minimize functions to calculate the objective minima function to imitate the ML4T udacity course.

**Video**: #3 “Minimizer in Pyton.”

**Timestamp**: 3:04

### NineExampleThree()

**Description**: Utilizes Error, FitLine, and scipy.optimize.minimize functions to fit a line to a given set of data points using optimization to imitate the ML4T udacity course.

**Video**: #9 “Fit a line to given data points.”

**Timestamp**: 5:17

### NineExampleFour()

**Description**: Utilizes ErrorPoly, FitPoly, and scipy.optimize.minimize functions to fit a line to a given set of polynomial function specific data points using optimization to imitate the ML4T udacity course.

**Video**: #10 “And it works for polynomials too!”

**Timestamp**: 0:00

## Lesson Ten Examples

### TenExampleOne()

**Description**: Utilizes TraderBot.py ComputePortfolioValue, NormalizeDfs, HistoricalData, and PlotData functions to plot the portfolio value of 25% Google, 25% Apple, 25% Gold, and 25% Exxon Mobil to imitate the ML4T udacity course.

**Video**: #2 “The difference optimization can make.”

**Timestamp**: 0:05

### TenExampleTwo()

**Description**: Utilizes TraderBot.py ComputePortfolioValue, NormalizeDfs, HistoricalData, and PlotData functions to plot the portfolio value of 0% Google, 40% Apple, 60% Gold, and 0% Exxon Mobil to imitate the ML4T udacity course.

**Video**: #2 “The difference optimization can make.”

**Timestamp**: 0:48

### TenExampleThree()

**Description**: Utilizes TraderBot.py NormalizeDfs, HistoricalData, ScipyOptimizePortfolio, and ComputePortfolioValue functions to plot the optimized portfolio value of the ML4T udacity course example using our own functions for comparison.

# TestLearners.ipynb

**Description:** Please see jupyter notebook file for more information.

## Utility Functions

### ProcessData(trainSize=0.6)

**Description:** This function serves to read a .csv file and prepare the data for training and testing by splitting it for us.

**Params:**

trainSize (float): Training size split.

**Returns:** Data for Train and Test for X and Y.

### TrainTestLearner(trainX, trainY, testX, testY, learnerArgs, iterations=1, maxLeafSize=None, maxBagSize=None, verbose=False, \*\*kwargs)

**Description:** This function serves to train and test a learner.

**Params:**

trainX (np.Array): Training data for X.

trainY (np.Array): Training data for Y.

testX (np.Array): Testing data for X.

testY (np.Array): Testing data for Y.

learnerArgs (Object): Learner being used (i.e., DTLearner)

iterations (int): Times data is trained and tested.

maxLeafSize (int): Max leaf size range for training tree learner.

maxBagSize (int): Max value for bag size range when training bag learner.

\*\*kwargs: Additional arguments for learner constructors.

**Returns:** RMSEs and Correlations for training and testing data.

### Visualize(trainData, testData, title=’Results’, xLabel=’X-Axis’, yLabel=’Y-Axis’, xAxisLen=1)

**Description:** This function serves to plot the results of training and testing a learner.

**Params:**

trainData (np.Array): Training data.

testData (np.array): Testing data.

title (str): Title of plot.

xLabel (str): X label of plot.

yLabel (str): Y label of plot.

xAxisLen (int): Length of X Axis.

**Returns:** A plot visualizing the results.

**\*For the rest of the functions, refer to the TestLearners.ipynb file directly\*.**