InferTrade

Release 0.0.35

InferStat

Jun 20, 2021

CONTENTS:

CHAPTER

ONE

INFERTRADE

`infertrade` is an open source trading and investment strategy library designed for accessibility and compatibility.

The `infertrade` package seeks to achieve three objectives:

- Simplicity: a simple `pandas` to `pandas` interface that those experienced in trading but new to Python can easily use.
- Gateway to data science: classes that allow rules created for the infertrade simple interface to be used with `scikit-learn` functionality for prediction and calibration. (fit, transform, predict, pipelines, gridsearch) and `scikit-learn` compatible libraries, like `feature-engine`.
- The best open source trading strategies: wrapping functionality to allow strategies from any open source Python libraries with compatible licences, such as `ta` to be used with the `infertrade` interface.

The project is licenced under the Apache 2.0 licence.

1.1 Connection to InferTrade.com

Many thanks for looking into the `infertrade` package!

I created InferTrade.com to provide cutting edge statistical analysis in an accessible free interface. The intention was to help individuals and small firms have access to the same quality of analysis as large institutions for systematic trading and to allow more time to be spent on creating good signals rather than backtesting and strategy verification. If someone has done the hard work of gaining insights into markets I wanted them to be able to compete in a landscape of increasingly automated statistically-driven market participants. A huge amount of effort has been made by the trading and AI/ML communities to create open source packages with powerful diagnostic functionality, which means you do not need to build a large and complex in-house analytics library to be able to support your investment decisions with solid statistical machine learning. However there remain educational and technical barriers to using this community-created wealth if you are not an experience programmer or do not have mathematical training. I want InferTrade.com to allow everyone trading in markets to have access without barriers - cost, training or time - to be competitive, with an easy to use interface that both provides direct analysis and education insights to support your trading.

The initial impetus for the creation of this open source package, `infertrade` was to ensure any of our users finding an attractive strategy on InferTrade.com could easily implement the rule in Python and have full access to the code to fully understand every aspect of how it works. By adding wrapper for existing libraries we hope to support further independent backtesting by users with their own preferred choice of trading libraries. We at InferStat heavily use open source in delivering InferTrade.com's functionality and we also wanted to give something back to the trading and data science community. The Apache 2.0 licence is a permissive licence, so that you can use or build upon `infertrade` for your personal, community or commercial projects.

The `infertrade` package and InferTrade.com will be adding functionality each week, and we are continually seeking to improve the experience and support the package and website provides for traders, portfolio managers and other users. Gaining feedback on new features is extremely helpful for us to improve our UX and design, as are any ideas for enhancements that would help you to trade better. If you would like to assist me in turning InferTrade into the leading open source trading platform we can offer participation in our Beta Testing programme (sign up link). You can also fork this repository and make direct improvements to the package.

Best, Tom Oliver

InferStat Founder and CEO

- https://github.com/ta-oliver
- https://www.linkedin.com/in/thomas-oliver-09487b9/

1.2 Contact Us

This was InferStat's first open source project and we welcome your thoughts for improvements to code structure, documentation or any changes that would support your use of the library.

If you would like assistance with using the `infertrade` you can email us at support@infertrade.com or book a video call

If you would like to contribute to the package, e.g. to add support for an additional package or library, please see our contributing information.

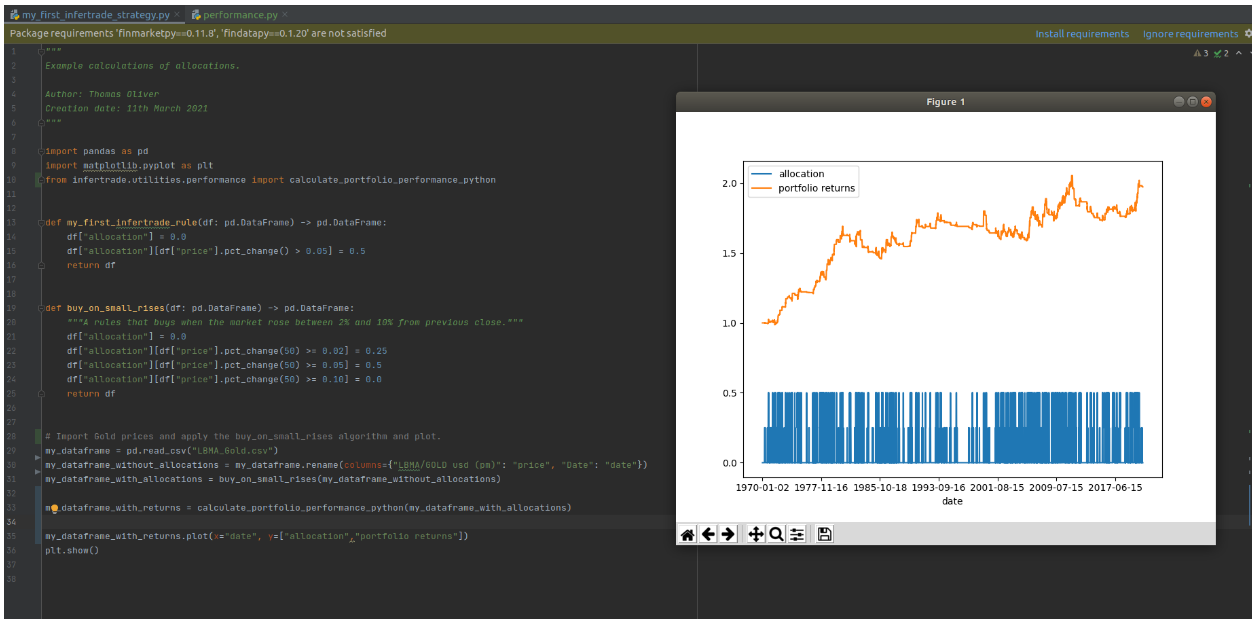
1.3 Quickstart

Please note the project requires Python 3.7 or higher due to dependent libraries used.

See Windows or Linux guides for installation details.

1.3.1 My First InferTrade Rule

```
import pandas as pd
import matplotlib.pyplot as plt

def my_first_infertrade_rule(df: pd.DataFrame) -> pd.DataFrame:
    df["allocation"] = 0.0
    df["allocation"][df.pct_change() > 0.02] = 0.5
    return df

my_dataframe = pd.read_csv("example_market_data.csv")
my_dataframe_with_allocations = my_first_infertrade_rule(my_dataframe)
my_dataframe_with_allocations.plot(["close"], ["allocation"])
plt.show()
```



Package requirements 'finmarketpy==0.11.8', 'findatapy==0.1.20' are not satisfied

```python
"""
Example calculations of allocations.

Author: Thomas Oliver
Creation date: 11th March 2021
"""

import pandas as pd
import matplotlib.pyplot as plt
from infertrade.utilities.performance import calculate_portfolio_performance_python


def my_first_infertrade_rule(df: pd.DataFrame) -> pd.DataFrame:
    df["allocation"] = 0.0
    df["allocation"][df["price"].pct_change() > 0.05] = 0.5
    return df


def buy_on_small_rises(df: pd.DataFrame) -> pd.DataFrame:
    """A rules that buys when the market rose between 2% and 10% from previous close."""
    df["allocation"] = 0.0
    df["allocation"][df["price"].pct_change(50) >= 0.02] = 0.25
    df["allocation"][df["price"].pct_change(50) >= 0.05] = 0.5
    df["allocation"][df["price"].pct_change(50) >= 0.10] = 0.0
    return df


# Import Gold prices and apply the buy_on_small_rises algorithm and plot.
my_dataframe = pd.read_csv("LBMA_Gold.csv")
my_dataframe_without_allocations = my_dataframe.rename(columns={"LBMA/GOLD usd (pm)": "price", "Date": "date"})
my_dataframe_with_allocations = buy_on_small_rises(my_dataframe_without_allocations)

my_dataframe_with_returns = calculate_portfolio_performance_python(my_dataframe_with_allocations)

my_dataframe_with_returns.plot(x="date", y=["allocation","portfolio returns"])
plt.show()
```

1.3.2 Basic usage with community functions

"Community" functions are those declared in this repository, not retrieved from an external package. They are all exposed at `infertrade.algos.community`.

```python
from infertrade.algos.community import normalised_close, scikit_signal_factory
from infertrade.data.simulate_data import simulated_market_data_4_years_gen
signal_transformer = scikit_signal_factory(normalised_close)
signal_transformer.fit_transform(simulated_market_data_4_years_gen())
```

1.3.3 Usage with TA

```python
from infertrade.algos.community import scikit_signal_factory
from infertrade.data.simulate_data import simulated_market_data_4_years_gen
from infertrade.algos import ta_adaptor
from ta.trend import AroonIndicator
adapted_aroon = ta_adaptor(AroonIndicator, "aroon_down", window=1)
signal_transformer = scikit_signal_factory(adapted_aroon)
signal_transformer.fit_transform(simulated_market_data_4_years_gen())
```

1.3.4 Calculate positions with simple position function

```python
from infertrade.algos.community.allocations import constant_allocation_size
from infertrade.utilities.operations import scikit_allocation_factory
from infertrade.data.simulate_data import simulated_market_data_4_years_gen

position_transformer = scikit_allocation_factory(constant_allocation_size)
position_transformer.fit_transform(simulated_market_data_4_years_gen())
# TODO add example with parameters
```

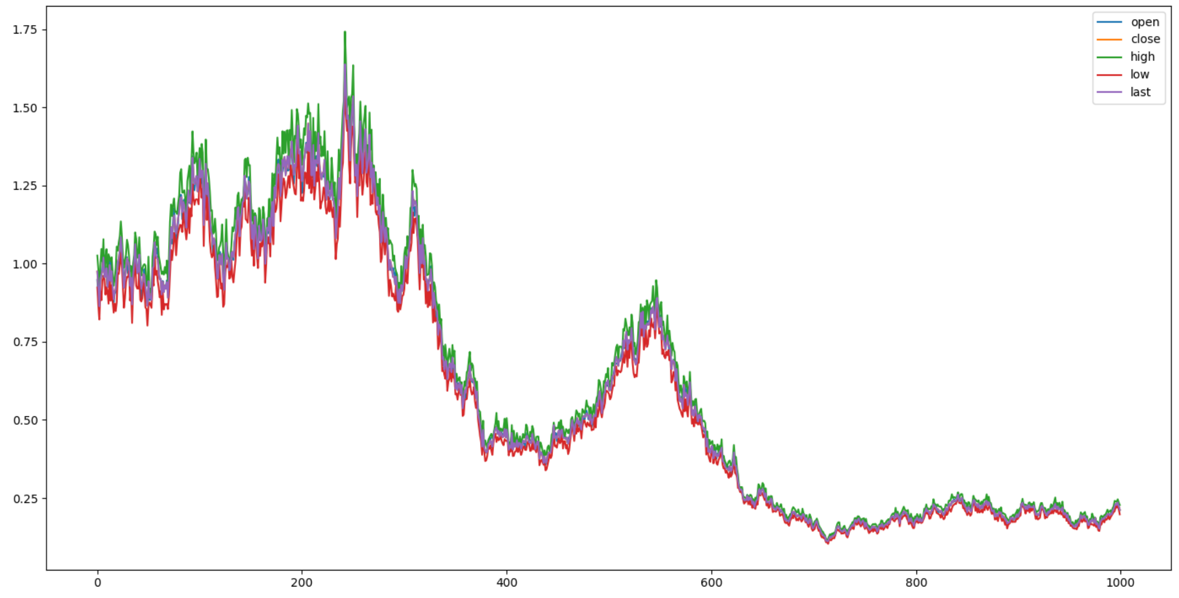
1.3.5 Example of position calculation via kelly just based on signal generation

```
from infertrade.algos.community import scikit_signal_factory
from infertrade.data.simulate_data import simulated_market_data_4_years_gen
from infertrade.utilities.operations import PositionsFromPricePrediction,␣
↪PricePredictionFromSignalRegression
from sklearn.pipeline import make_pipeline
from infertrade.algos import ta_adaptor
from ta.trend import AroonIndicator

adapted_aroon = ta_adaptor(AroonIndicator, "aroon_down", window=1)

pipeline = make_pipeline(scikit_signal_factory(adapted_aroon),
                         PricePredictionFromSignalRegression(),
                         PositionsFromPricePrediction()
                         )

pipeline.fit_transform(simulated_market_data_4_years_gen())
```

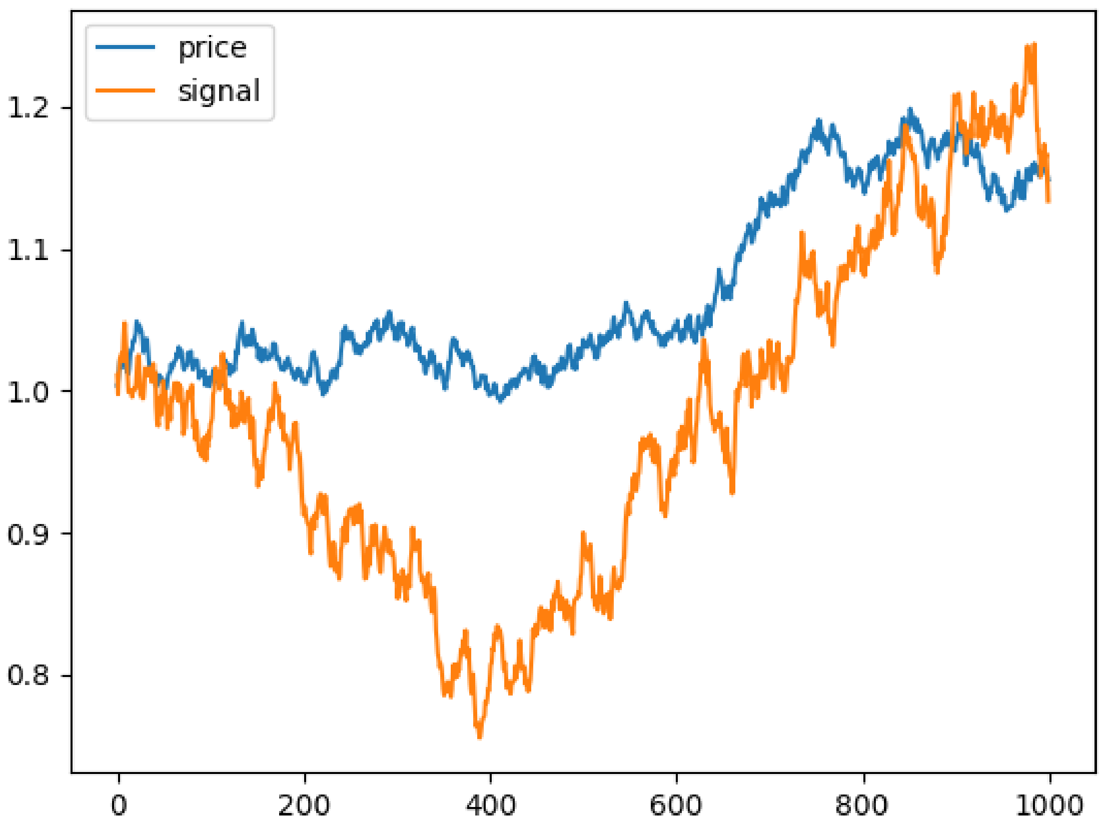
1.3.6 Creating simulated data for testing

For convenience, the `infertrade.data` module contains some basic functions for simulating market data.

```
import matplotlib.pyplot as plt
from infertrade.data.simulate_data import simulated_market_data_4_years_gen
simulated_market_data_4_years_gen().plot(y=["open", "close", "high", "low", "last"])
plt.show()
```



```
import matplotlib.pyplot as plt
from infertrade.data.simulate_data import simulated_correlated_equities_4_years_gen
simulated_correlated_equities_4_years_gen().plot(y=["price", "signal"])
plt.show()
```


price
signal
1.2
1.1
1.0
0.9
0.8
0
200
400
600
800
1000

CHAPTER

TWO

INFERTRADE PACKAGE

2.1 infertrade.PandasEnum module

PandasEnum for providing the special string names used with the InferTrade interface.

class infertrade.PandasEnum.**PandasEnum**(*value*)

Bases: enum.Enum

Provides the strings for special column names that InferTrade uses in identifying pandas dataframe contents.

These strings should not be used for other purposes.

Meanings: MID - this is the mid price used to calculate performance.

ALLOCATION - the fraction of the overall portfolio the strategy wants to invest in the market. May differ from the
amount invested where the strategy requires minimum deviations to trigger position adjustment.

VALUATION - the value of strategy, after running a hypothetical rule implementing the strategy. 1.0 = 100% means no
profit or loss. 0.9 = 90%, means a -10% cumulative loss. 1.1 = 110% means a 10% overall cumulative gain.

BID_OFFER_SPREAD - the fractional bid-offer spread - 2 * (ask - bid)/(ask + bid) - for that time period.

SIGNAL - an information time series that could be used for construction of an allocation series.

ADJUSTED_CLOSE = 'adjusted close'

ADJ_CLOSE = 'adj close'

ADJ_DOT_CLOSE = 'adj. close'

ALLOCATION = 'allocation'

BID_OFFER_SPREAD = 'bid_offer_spread'

CASH_INCREASE = 'cash_flow'

CLOSE = 'close'

FORECAST_PRICE_CHANGE = 'forecast_price_change'

MID = 'price'

PERCENTAGE_TO_BUY = 'trade_percentage'

PERIOD_END_ALLOCATION = 'end_of_period_allocation'

PERIOD_END_CASH = 'period_end_cash'

PERIOD_END_SECURITIES = 'period_end_securities'

PERIOD_START_ALLOCATION = 'start_of_period_allocation'

PERIOD_START_CASH = 'period_start_cash'

PERIOD_START_SECURITIES = 'period_start_securities'

PRICE_SYNONYMS = ['close', 'adjusted close', 'adj close', 'adj. close']

SECURITIES_BOUGHT = 'security_purchases'

SIGNAL = 'signal'

TRADING_SKIPPED = 'trading_skipped'

VALUATION = 'portfolio_return'

infertrade.PandasEnum.**create_price_column_from_synonym**(*df_potentially_missing_price_column: pandas.core.frame.DataFrame*)

If the price column is missing then we will look for the "close" instead and copy those values.

2.2 infertrade.api module

API facade that allows interaction with the library with strings and vanilla Python objects.

class infertrade.api.**Api**

Bases: object

All public methods should input/output json-serialisable dictionaries.

static algorithm_categories() → List[str]

Returns the list of algorithm types.

static available_algorithms(*filter_by_package: Optional[Union[str, List[str]]] = None, filter_by_category: Optional[Union[str, List[str]]] = None*) → List[str]

Returns a list of strings that are available strategies.

static available_packages() → List[str]

Returns the list of supported packages.

static calculate_allocations(*df: pandas.core.frame.DataFrame, name_of_strategy: str, name_of_price_series: str = 'price'*) → pandas.core.frame.DataFrame

Calculates the allocations using the supplied strategy.

static calculate_allocations_and_returns(*df: pandas.core.frame.DataFrame, name_of_strategy: str, name_of_price_series: str = 'price'*) → pandas.core.frame.DataFrame

Calculates the returns using the supplied strategy.

static calculate_returns(*df: pandas.core.frame.DataFrame*) → pandas.core.frame.DataFrame

Calculates the returns from supplied positions.

static calculate_signal(*df: pandas.core.frame.DataFrame, name_of_signal: str*) → pandas.core.frame.DataFrame

Calculates the allocations using the supplied strategy.

static determine_package_of_algorithm(*name_of_algorithm: str*) → str

Determines the original package of a strategy.

static get_algorithm_information() → dict

Provides information on algorithms (signals and positions) as flat list (not nested by category).

static get_allocation_information() → dict
: Provides information on algorithms that calculate positions.

static get_available_representations(*name_of_algorithm: str*) → List[str]
: Describes which representations exist for the algorithm.

static get_signal_information() → dict
: Provides information on algorithms that calculate signals.

static required_inputs_for_algorithm(*name_of_strategy: str*) → List[str]
: Describes the input columns needed for the strategy.

static required_parameters_for_algorithm(*name_of_strategy: str*) → List[str]
: Describes the input columns needed for the strategy.

static return_algorithm_category(*algorithm_name: str*) → str
: Returns the category of algorithm as a string.

static return_representations(*name_of_algorithm: str, representation_or_list_of_representations: Optional[Union[str, List[str]]] = None*) → dict
: Returns the representations (e.g. URLs of relevant documentation).

2.3 infertrade.base module

Base functionality used by other functions in the package.

infertrade.base.**get_portfolio_calc**(*func: callable*) → callable
: Given a position calculation generates the portfolio valuation index.

infertrade.base.**get_positions_calc**(*df: pandas.core.frame.DataFrame*, *func: callable*) → callable
: Pass through method to return the results of the allocation calculation.

infertrade.base.**get_signal_calc**(*func: callable*, *adapter: Optional[callable] = None*) → callable
: An adapter to calculate a signal prior to usage within a trading rule.

2.4 Module contents

Base directory for the package

Files and sub-directory information:

/algos - contains trading strategies /data - sources of simulated and external data /utilities - performance calculations and other utilities __init__.py - package identifier _version - package version number.

CHAPTER

THREE

ALGOS

3.1 algos.community package

3.1.1 algos.community.allocations module

Allocation algorithms are functions used to compute allocations - % of your portfolio to invest in a market or asset.

`algos.community.allocations.`**`buy_and_hold`**(*dataframe: pandas.core.frame.DataFrame*) → pandas.core.frame.DataFrame

Allocates 100% of strategy budget to asset, holding to end of period (or security bankruptcy).

`algos.community.allocations.`**`chande_kroll_crossover_strategy`**(*dataframe: pandas.core.frame.DataFrame*) → pandas.core.frame.DataFrame

This simple all-or-nothing rule: (1) allocates 100% of the portofolio to a long position on the asset when the price of the asset is above both the Chande Kroll stop long line and Chande Kroll stop short line, and (2) according to the value set for the allow_short_selling parameter, either allocates 0% of the portofiolio to the asset or allocates 100% of the portfolio to a short position on the asset when the price of the asset is below both the Chande Kroll stop long line and the Chande Kroll stop short line.

`algos.community.allocations.`**`change_regression`**(*dataframe: pandas.core.frame.DataFrame*, *change_coefficient: float = 0.1*, *constant_coefficient: float = 0.1*) → pandas.core.frame.DataFrame

This is a regression-type approach that directly calculates allocation from change in the research level.

parameters: change_coefficient: The coefficient for allocation size versus the prior day fractional change in the research. constant_coefficient: The coefficient for the constant contribution.

`algos.community.allocations.`**`constant_allocation_size`**(*dataframe: pandas.core.frame.DataFrame*, *fixed_allocation_size: float = 1.0*) → pandas.core.frame.DataFrame

Returns a constant allocation, controlled by the fixed_allocation_size parameter.

parameters: fixed_allocation_size: determines allocation size.

`algos.community.allocations.`**`difference_regression`**(*dataframe: pandas.core.frame.DataFrame*, *difference_coefficient: float = 0.1*, *constant_coefficient: float = 0.1*) → pandas.core.frame.DataFrame

This trading rules regresses the 1-day price changes seen historical against the prior day's % change of the research series.

parameters: difference_coefficient: The coefficient for dependence on the log gap between the signal series and the price series. constant_coefficient: The coefficient for the constant contribution.

`algos.community.allocations.`**`fifty_fifty`**`(`*dataframe*`)` → pandas.core.frame.DataFrame

Allocates 50% of strategy budget to asset, 50% to cash.

`algos.community.allocations.`**`high_low_difference`**`(`*dataframe: pandas.core.frame.DataFrame*, *scale: float = 1.0*, *constant: float = 0.0*`)` → pandas.core.frame.DataFrame

Returns an allocation based on the difference in high and low values. This has been added as an example with multiple series and parameters.

parameters: scale: determines amplitude factor. constant: scalar value added to the allocation size.

`algos.community.allocations.`**`level_and_change_regression`**`(`*dataframe: pandas.core.frame.DataFrame*, *level_coefficient: float = 0.1*, *change_coefficient: float = 0.1*, *constant_coefficient: float = 0.1*`)` → pandas.core.frame.DataFrame

This trading rules regresses the 1-day price changes seen historical against the prior day's % change of the research series and level of research series.

parameters: level_coefficient: The coefficient for allocation size versus the level of the signal. change_coefficient: The coefficient for allocation size versus the prior day fractional change in the research. constant_coefficient: The coefficient for the constant contribution.

`algos.community.allocations.`**`level_regression`**`(`*dataframe: pandas.core.frame.DataFrame*, *level_coefficient: float = 0.1*, *constant_coefficient: float = 0.1*`)` → pandas.core.frame.DataFrame

This is a regression-type approach that directly calculates allocation from research level.

parameters: level_coefficient: The coefficient for allocation size versus the level of the signal. constant_coefficient: The coefficient for the constant contribution.

`algos.community.allocations.`**`sma_crossover_strategy`**`(`*dataframe: pandas.core.frame.DataFrame*, *fast: int = 0*, *slow: int = 0*`)` → pandas.core.frame.DataFrame

A Simple Moving Average crossover strategy, buys when short-term SMA crosses over a long-term SMA.

parameters: fast: determines the number of periods to be included in the short-term SMA. slow: determines the number of periods to be included in the long-term SMA.

`algos.community.allocations.`**`weighted_moving_averages`**`(`*dataframe: pandas.core.frame.DataFrame*, *avg_price_coeff: float = 1.0*, *avg_research_coeff: float = 1.0*, *avg_price_length: int = 2*, *avg_research_length: int = 2*`)` → pandas.core.frame.DataFrame

This rule uses weightings of two moving averages to determine trade positioning.

The parameters accepted are the integer lengths of each average (2 parameters - one for price, one for the research signal) and two corresponding coefficients that determine each average's weighting contribution. The total sum is divided by the current price to calculate a position size.

This strategy is suitable where the dimensionality of the signal/research series is the same as the dimensionality of the price series, e.g. where the signal is a price forecast or fair value estimate of the market or security.

parameters: avg_price_coeff: price contribution scalar multiple. avg_research_coeff: research or signal contribution scalar multiple. avg_price_length: determines length of average of price series. avg_research_length: determines length of average of research series.

3.1.2 algos.community.signals module

Functions used to compute signals. Signals may be used for visual inspection or as inputs to trading rules.

`algos.community.signals.`**`chande_kroll`**(*df: pandas.core.frame.DataFrame*, *average_true_range_periods: int = 10*, *average_true_range_multiplier: float = 1.0*, *stop_periods: int = 9*) → pandas.core.frame.DataFrame

Calculates signals for the Chande-Kroll stop.

See here: https://www.tradingview.com/support/solutions/43000589105-chande-kroll-stop

`algos.community.signals.`**`high_low_diff`**(*df: pandas.core.frame.DataFrame*) → pandas.core.frame.DataFrame

Calculates the range between low and high price.

`algos.community.signals.`**`high_low_diff_scaled`**(*df: pandas.core.frame.DataFrame*, *amplitude: float = 1*) → pandas.core.frame.DataFrame

Example signal based on high-low range times scalar.

`algos.community.signals.`**`normalised_close`**(*df: pandas.core.frame.DataFrame*) → pandas.core.frame.DataFrame

Scales the close by the maximum value of the close across the whole price history.

Note that this signal cannot be determined until the end of the historical period and so is unlikely to be suitable
as an input feature for a trading strategy.

`algos.community.signals.`**`scikit_signal_factory`**(*signal_function: callable*)

A class compatible with Sci-Kit Learn containing the signal function.

`algos.community.signals.`**`test_NormalisedCloseTransformer`**()

nct = NormalisedCloseTransformer()

3.1.3 Module contents

Functions for signals and positions created within this package.

3.2 algos.external package

3.2.1 algos.external.ta module

Functions to facilitate usage of TA functionality with infertrade's interface.

`algos.external.ta.`**`ta_adaptor`**(*indicator_mixin: Type[ta.utils.IndicatorMixin]*, *function_name: str*, ***kwargs*) → callable

Wraps strategies from ta to make them compatible with infertrade's interface.

3.2.2 Module contents

Functionality to adapt external libraries for usage with InferTrade.

3.3 Module contents

External algorithms directory.

CHAPTER FOUR

DATA

4.1 data.simulate_data module

Simple test data generator.

`data.simulate_data.`**`simulated_correlated_equities_4_years_gen`**`()`
This function creates ~4 years of simulated equity pair daily data for testing interfaces.

`data.simulate_data.`**`simulated_market_data_4_years_gen`**`()`
This function creates ~4 years of simulated daily data for testing interfaces.

4.2 Module contents

Data directory to facilitate bringing in external data or simulating a market.

CHAPTER FIVE

UTILITIES

5.1 utilities.operations module

This submodule includes facilities for operations such as converting positions to price predictions and vice versa.

class `utilities.operations.`**`PositionsFromPricePrediction`**

Bases: `sklearn.base.TransformerMixin`, `sklearn.base.BaseEstimator`

This class calculates the positions to take assuming Kelly Criterion.

`fit`(*X*, *y=None*)

This method is not used.

`transform`(*X: pandas.core.frame.DataFrame*, *y=None*) → pandas.core.frame.DataFrame

This method calculates the positions to be taken based on the forecast price, assuming the Kelly Criterion.

Args: X: A pandas.DataFrame object

Returns: A pandas.DataFrame object

class `utilities.operations.`**`PricePredictionFromPositions`**

Bases: `sklearn.base.TransformerMixin`, `sklearn.base.BaseEstimator`

This class converts positions into implicit price predictions based on the Kelly Criterion and an assumed volatility.

`fit`(*X*, *y=None*)

This method is not used.

`transform`(*X: pandas.core.frame.DataFrame*, *y=None*) → pandas.core.frame.DataFrame

This method converts allocations into the forecast one-day price changes.

Args: X: A pandas.DataFrame object

Returns: A pandas.DataFrame object

class `utilities.operations.`**`PricePredictionFromSignalRegression`**(*market_to_trade: Optional[str] = None*)

Bases: `sklearn.base.TransformerMixin`, `sklearn.base.BaseEstimator`

This class creates price predictions from signal values.

Attributes: market_to_trade: The name of the column which contains the historical prices.

`fit`(*X: numpy.array*, *y=None*)

`transform`(*X: pandas.core.frame.DataFrame*, *y=None*) → pandas.core.frame.DataFrame

This method transforms a signal input to a price prediction.

Args: X: A pandas.DataFrame object

Returns: A pandas.DataFrame object

class `utilities.operations.`**`ReturnsFromPositions`**

Bases: `sklearn.base.TransformerMixin`, `sklearn.base.BaseEstimator`

This class calculates returns from positions.

`fit`(*X*, *y=None*)

This method is not used.

`transform`(*X: pandas.core.frame.DataFrame*, *y=None*) → pandas.core.frame.DataFrame

This method converts positions into the cumulative portfolio return.

Args: X: A pandas.DataFrame object

Returns: A pandas.DataFrame object

`utilities.operations.`**`diff_log`**(*x: Union[numpy.ndarray, pandas.core.series.Series]*) → numpy.ndarray

Differencing and log transformation between the current and a prior element.

Args: x: A numpy.ndarray or pandas.Series object

Returns: A numpy.ndarray with the results

`utilities.operations.`**`dl_lag`**(*x: Union[numpy.ndarray, pandas.core.series.Series]*, *shift: int = 1*) → numpy.ndarray

Differencing and log transformation of lagged series.

Args: x: A numpy.ndarray or pandas.Series object shift: The number of periods by which to shift the input time series

Returns: A numpy.ndarray with the results

`utilities.operations.`**`lag`**(*x: Union[numpy.ndarray, pandas.core.series.Series]*, *shift: int = 1*) → numpy.ndarray

Lag (shift) series by desired number of periods.

Args: x: A numpy.ndarray or pandas.Series object shift: The number of periods by which to shift the input time series

Returns: A numpy.ndarray with the results

`utilities.operations.`**`log_price_minus_log_research`**(*x: Union[numpy.ndarray, pandas.core.series.Series]*, *shift: int*) → numpy.ndarray

Difference of two lagged log series.

Args: x: A numpy.ndarray or pandas.Series object with exactly two columns shift: The number of periods by which the lag both series

Returns: A numpy.ndarray with the results

`utilities.operations.`**`moving_average`**(*x: Union[numpy.ndarray, pandas.core.series.Series]*, *window: int*) → numpy.ndarray

Calculate moving average of series for desired number of periods (window).

Args: x: A numpy.ndarray or pandas.Series object window: The number of periods to be included in the moving average.

Returns: A numpy.ndarray with the results `utilities.operations.`**`pct_chg`**(*x: Union[numpy.ndarray, pandas.core.series.Series]*) → numpy.ndarray

Percentage change between the current and a prior element.

Args: x: A numpy.ndarray or pandas.Series object

Returns: A numpy.ndarray with the results

`utilities.operations.`**`research_over_price_minus_one`**(*x: Union[numpy.ndarray, pandas.core.series.Series], shift: int*) → numpy.ndarray

Difference of two lagged log series.

Args: x: A numpy.ndarray or pandas.Series object with exactly two columns shift: The number of periods by which the lag both series

Returns: A numpy.ndarray with the results

`utilities.operations.`**`scikit_allocation_factory`**(*allocation_function: callable*) → sklearn.preprocessing._function_transformer.FunctionTransforme

This function creates a SciKit Learn compatible Transformer embedding the position calculation.

Args: allocation_function: A function to be turned into a sklearn.preprocessing.FunctionTransformer

Returns: A sklearn.preprocessing.FunctionTransformer

`utilities.operations.`**`zero_one_dl`**(*x: Union[numpy.ndarray, pandas.core.series.Series]*) → numpy.ndarray

Returns ones for positive values of "diff-log" series, and zeros for negative values.

Args: x: A numpy.ndarray or pandas.Series object

Returns: A numpy.ndarray with the results

5.2 utilities.performance module

Performance calculation using the InferTrade interface.

`utilities.performance.`**`calculate_allocation_from_cash`**(*last_cash_after_trade: float, last_securities_after_transaction: float, spot_price: float*) → float

Calculates the current allocation.

`utilities.performance.`**`calculate_portfolio_performance_python`**(*df_with_positions: pandas.core.frame.DataFrame, skip_checks: bool = False, show_absolute_bankruptcies: bool = False, annual_strategy_fee: float = 0.0, daily_spread_percent_override: float = 0.0, minimum_allocation_change_to_adjust: float = 0.0, detailed_output: bool = True*)

This is the main vanilla Python calculation of portfolio performance.

`utilities.performance.`**`check_if_should_skip_return_calculation`**(*previous_portfolio_return: float*, *spot_price: float*, *day: int*, *day_of_return_to_calculate: int*, *show_absolute_bankruptcies: bool*, *bankrupt: bool = False) -> (<class 'bool'>, <class 'float'>*)

This function checks if we should skip the returns calculation for the requested day.

`utilities.performance.`**`portfolio_index`**(*position_on_last_good_price: float, spot_price_usd: float, last_good_price_usd: Optional[float], current_bid_offer_spread_percent: float, target_allocation_perc: float, annual_strategy_fee_perc: float, last_securities_volume: float, last_cash_after_trade_usd: float, show_working: bool = False) -> (<class 'float'>, <class 'float'>, <class 'float'>*)

A function for calculating the cumulative return of the portfolio.

`utilities.performance.`**`rounded_allocation_target`**(*unconstrained_target_position: float*, *minimum_allocation_change_to_adjust: float*) → float

Determines what allocation size to take if using rounded targets.

5.3 utilities.simple_functions module

Simple functions used across the package.

`utilities.simple_functions.`**`add_package`**(*dictionary: dict*, *string_label: str*) → dict

Adds a string to every item.

5.4 Module contents

Utilities directory for functions that uses the InferTrade interface.

PYTHON MODULE INDEX

INDEX