ml_investment

Artur Fattakhov

Feb 01, 2022

CONTENTS:

CHAPTER

ONE

INSTALLATION

PyPI version

```
$ pip install ml-investment
```

Latest version from source

```
$ pip install git+https://github.com/fartuk/ml_investment
```

Configuration

You may use config file *~/.ml_investment/config.json* to change repo parameters i.e. downloading datasets pathes, models pathes etc.

Private information (i.e. api tokens for private datasets downloading) should be located at *~/.ml_investment/secrets.json*

CHAPTER

TWO

QUICK START

2.1 Use application model

There are several pre-defined fitted models at `ml_investment.applications`. It incapsulating data and weights downloading, pipeline creation and model fitting. So you can just use it without knowing internal structure.

```
from ml_investment.applications.fair_marketcap_yahoo import FairMarketcapYahoo

fair_marketcap_yahoo = FairMarketcapYahoo()
fair_marketcap_yahoo.execute(['AAPL', 'FB', 'MSFT'])
```

ticker	date	fair_marketcap_yahoo
AAPL	2020-12-31	5.173328e+11
FB	2020-12-31	8.442045e+11
MSFT	2020-12-31	4.501329e+11

2.2 Create your own pipeline

1. Download data

You may download default datasets by `ml_investment.download_scripts`

```
from ml_investment.download_scripts import download_yahoo
from ml_investment.utils import load_config

# Config located at ~/.ml_investment/config.json
config = load_config()

download_yahoo.main(config['yahoo_data_path'])
```

```
>>> 1365it [03:32,  6.42it/s]
>>> 1365it [01:49,  12.51it/s]
```

2. Create dict with dataloaders

You may choose from default `ml_investment.data_loaders` or wrote your own. Each dataloader should have `load(index)` interface.

```
from ml_investment.data_loaders.yahoo import YahooQuarterlyData, YahooBaseData

data = {}
data['quarterly'] = YahooQuarterlyData(config['yahoo_data_path'])
data['base'] = YahooBaseData(config['yahoo_data_path'])
```

3. Define and fit pipeline

You may specify all steps of pipeline creation. Base pipeline consist of the folowing steps:

- Create data dict(it was done in previous step)
- Define features. Features is a number of values and characteristics that will be calculated for model trainig. Default feature calculators are located at `ml_investment.features`
- Define targets. Target is a final goal of the pipeline, it should represent some desired useful property. Default target calculators are located at `ml_investment.targets`
- Choose model. Model is machine learning algorithm, core of the pipeline. It also may incapsulate validation and other stuff. You may use wrappers from `ml_investment.models`

```
import lightgbm as lgbm
from ml_investment.utils import load_config, load_tickers
from ml_investment.features import QuarterlyFeatures, BaseCompanyFeatures,\
                                   FeatureMerger
from ml_investment.targets import BaseInfoTarget
from ml_investment.models import LogExpModel, GroupedOOFModel
from ml_investment.pipelines import Pipeline
from ml_investment.metrics import median_absolute_relative_error

fc1 = QuarterlyFeatures(data_key='quarterly',
                        columns=['netIncome',
                                 'cash',
                                 'totalAssets',
                                 'ebit'],
                        quarter_counts=[2, 4, 10],
                        max_back_quarter=1)

fc2 = BaseCompanyFeatures(data_key='base', cat_columns=['sector'])

feature = FeatureMerger(fc1, fc2, on='ticker')

target = BaseInfoTarget(data_key='base', col='enterpriseValue')

base_model = LogExpModel(lgbm.sklearn.LGBMRegressor())
model = GroupedOOFModel(base_model=base_model,
                        group_column='ticker',
                        fold_cnt=4)

pipeline = Pipeline(data=data,
                    feature=feature,
                    target=target,
                    model=model,
                    out_name='my_super_model')
```

(continues on next page)

(continued from previous page)

```
tickers = load_tickers()['base_us_stocks']
pipeline.fit(tickers, metric=median_absolute_relative_error)
```

```
>>> {'metric_my_super_model': 0.40599471294301914}
```

4. Inference your pipeline

Since `ml_investment.models.GroupedOOFModel` was used, there are no data leakage and you may use pipeline on the same company tickers.

```
pipeline.execute(['AAPL', 'FB', 'MSFT'])
```

ticker	date	my_super_model
AAPL	2020-12-31	8.170051e+11
FB	2020-12-31	3.898840e+11
MSFT	2020-12-31	3.540126e+11

CHAPTER

THREE

APPLICATIONS

Collection of pre-trained models

3.1 FairMarketcapYahoo

ml_investment.applications.fair_marketcap_yahoo.**FairMarketcapYahoo**(*pretrained=True*) → *ml_investment.pipelines.Pipeline*

Model is used to estimate fair company marketcap for *last* quarter. Pipeline uses features from *BaseCompanyFeatures*, *QuarterlyFeatures* and trained to predict real market capitalizations (using *QuarterlyTarget*). Since some companies are overvalued and some are undervalued, the model makes an average "fair" prediction. *yahoo* is used for loading data.

Parameters **pretrained** – use pretreined weights or not. If so, *fair_marketcap_yahoo.pickle* will be downloaded. Downloading directory path can be changed in *~/.ml_investment/config.json* `models_path`

ml_investment.applications.fair_marketcap_yahoo.**main**()

Default model training. Resulted model weights directory path can be changed in *~/.ml_investment/config.json* `models_path`

3.2 FairMarketcapSF1

ml_investment.applications.fair_marketcap_sf1.**FairMarketcapSF1**(*max_back_quarter: Optional[int] = None*, *min_back_quarter: Optional[int] = None*, *data_source: Optional[str] = None*, *pretrained: bool = True*, *verbose: Optional[bool] = None*) → *ml_investment.pipelines.Pipeline*

Model is used to estimate fair company marketcap for several last quarters. Pipeline uses features from *BaseCompanyFeatures*, *QuarterlyFeatures*, *DailyAggQuarterFeatures*, `CommoditiesAggQuarterFeatures` and trained to predict real market capitalizations (using *QuarterlyTarget*). Since some companies are overvalued and some are undervalued, the model makes an average "fair" prediction. *sf1* and *quandl_commodities* is used for loading data.

Note: SF1 dataset is paid, so for using this model you need to subscribe and paste quandl token to *~/.ml_investment/secrets.json* `quandl_api_key`

Parameters

- **max_back_quarter** – max quarter number which will be used in model
- **min_back_quarter** – min quarter number which will be used in model
- **data_source** – which data use for model. One of ['sf1', 'mongo']. If 'mongo', than data will be loaded from db, credentials specified at *~/.ml_investment/config.json*. If 'sf1' - from folder specified at `sf1_data_path` in *~/.ml_investment/secrets.json*.
- **pretrained** – use pretreined weights or not. Downloading directory path can be changed in *~/.ml_investment/config.json* `models_path`
- **verbose** – show progress or not

ml_investment.applications.fair_marketcap_sf1.**main**(*data_source*)

Default model training. Resulted model weights directory path can be changed in *~/.ml_investment/config.json* `models_path`

3.3 FairMarketcapDiffYahoo

ml_investment.applications.fair_marketcap_diff_yahoo.**FairMarketcapDiffYahoo**(*pretrained=True*) → *ml_investment.pipelines.Pipeline*

Model is used to evaluate quarter-to-quarter(q2q) company fundamental progress. Model uses *QuarterlyDiffFeatures* (q2q results progress, e.g. 30% revenue increase, decrease in debt by 15% etc), *BaseCompanyFeatures*, *QuarterlyFeatures* and trying to predict smoothed real q2q marketcap difference(*DailySmoothedQuarterlyDiffTarget*). So model prediction may be interpreted as "fair" marketcap change according this q2q fundamental change. *yahoo* and *daily_bars* are used for loading data.

Parameters **pretrained** – use pretreined weights or not. If so, *fair_marketcap_diff_yahoo.pickle* will be downloaded. Downloading directory path can be changed in *~/.ml_investment/config.json* `models_path`

ml_investment.applications.fair_marketcap_diff_yahoo.**main**()

Default model training. Resulted model weights directory path can be changed in *~/.ml_investment/config.json* `models_path`

3.4 FairMarketcapDiffSF1

ml_investment.applications.fair_marketcap_diff_sf1.**FairMarketcapDiffSF1**(*max_back_quarter: Optional[int] = None*, *min_back_quarter: Optional[int] = None*, *data_source: Optional[str] = None*, *pretrained: bool = True*, *verbose: Optional[bool] = None*) → *ml_investment.pipelines.Pipeline*

Model is used to evaluate quarter-to-quarter(q2q) company fundamental progress. Model uses *QuarterlyDiffFeatures* (q2q results progress, e.g. 30% revenue increase, decrease in debt by 15% etc), *BaseCompanyFeatures*, *QuarterlyFeatures* `CommoditiesAggQuarterFeatures` and trying to

predict real q2q marketcap difference(*QuarterlyDiffTarget*). So model prediction may be interpreted as "fair" marketcap change according this q2q fundamental change. *sf1* is used for loading data.

Note: SF1 dataset is paid, so for using this model you need to subscribe and paste quandl token to *~/.ml_investment/secrets.json* `quandl_api_key`

Parameters

- **max_back_quarter** – max quarter number which will be used in model
- **min_back_quarter** – min quarter number which will be used in model
- **data_source** – which data use for model. One of ['sf1', 'mongo']. If 'mongo', than data will be loaded from db, credentials specified at *~/.ml_investment/config.json*. If 'sf1' - from folder specified at `sf1_data_path` in *~/.ml_investment/secrets.json*.
- **pretrained** – use pretreined weights or not. Downloading directory path can be changed in *~/.ml_investment/config.json* `models_path`
- **verbose** – show progress or not

`ml_investment.applications.fair_marketcap_diff_sf1.`**main**(*data_source*)

Default model training. Resulted model weights directory path can be changed in *~/.ml_investment/config.json* `models_path`

3.5 MarketcapDownStdYahoo

`ml_investment.applications.marketcap_down_std_yahoo.`**MarketcapDownStdYahoo**(*pretrained=True*) → *ml_investment.pipelines.Pipeline*

Model is used to predict future down-std value. Pipeline consist of time-series model training(*TimeSeriesOOFModel*) and validation on real marketcap down-std values(*DailyAggTarget*). Model prediction may be interpreted as "risk" for the next quarter. *yahoo* is used for loading data.

Parameters **pretrained** – use pretreined weights or not. If so, *marketcap_down_std_yahoo.pickle* will be downloaded. Downloading directory path can be changed in *~/.ml_investment/config.json* `models_path`

`ml_investment.applications.marketcap_down_std_yahoo.`**main**()

Default model training. Resulted model weights directory path can be changed in *~/.ml_investment/config.json* `models_path`

3.6 MarketcapDownStdSF1

ml_investment.applications.marketcap_down_std_sf1.**MarketcapDownStdSF1**(*max_back_quarter: Optional[int] = None*, *min_back_quarter: Optional[int] = None*, *data_source: Optional[str] = None*, *pretrained: bool = True*, *verbose: Optional[bool] = None*) → *ml_investment.pipelines.Pipeline*

Model is used to predict future down-std value. Pipeline consist of time-series model training(*TimeSeriesOOFModel*) and validation on real marketcap down-std values(*DailyAggTarget*). Model prediction may be interpreted as "risk" for the next quarter. *sf1* is used for loading data.

Note: SF1 dataset is paid, so for using this model you need to subscribe and paste quandl token to *~/.ml_investment/secrets.json* `quandl_api_key`

Parameters

- **max_back_quarter** – max quarter number which will be used in model
- **min_back_quarter** – min quarter number which will be used in model
- **data_source** – which data use for model. One of ['sf1', 'mongo']. If 'mongo', than data will be loaded from db, credentials specified at *~/.ml_investment/config.json*. If 'sf1' - from folder specified at `sf1_data_path` in *~/.ml_investment/secrets.json*.
- **pretrained** – use pretreined weights or not. Downloading directory path can be changed in *~/.ml_investment/config.json* `models_path`
- **verbose** – show progress or not

ml_investment.applications.marketcap_down_std_sf1.**main**(*data_source*)

Default model training. Resulted model weights directory path can be changed in *~/.ml_investment/config.json* `models_path`

CHAPTER

FOUR

FEATURES

Collection of feature calculators

4.1 QuarterlyFeatures

class ml_investment.features.**QuarterlyFeatures**(*data_key: str, columns: typing.List[str], quarter_counts: typing.List[int] = [2, 4, 10], max_back_quarter: int = 10, min_back_quarter: int = 0, stats: typing.Dict[str, typing.Callable] = {'max': <function amax>, 'mean': <function mean>, 'median': <function median>, 'min': <function amin>, 'std': <function std>}, calc_stats_on_diffs: bool = True, data_preprocessing: typing.Optional[typing.Callable] = None, n_jobs: int = 2, verbose: bool = False*)

Bases: `object`

Feature calculator for qaurtrly-based statistics. Return features for company quarter slices.

Parameters

- **data_key** – key of dataloader in `data` argument during *calculate()*
- **columns** – column names for feature calculation(like revenue, debt etc)
- **quarter_counts** – list of number of quarters for statistics calculation. e.g. if `quarter_counts = [2]` than statistics will be calculated on current and previous quarter
- **max_back_quarter** – max bound of company slices in time. If `max_back_quarter = 1` than features will be calculated for only current company quarter. If max_back_quarter is larger than total number of quarters for company than features will be calculated for all quarters
- **min_back_quarter** – min bound of company slices in time. If `min_back_quarter = 0` (default) than features will be calculated for all quarters. If `min_back_quarter = 2` than current and previous quarter slices will not be used for feature calculation
- **stats** – aggregation functions for features calculation. Should be as `Dict[str, Callable]`. Keys of this dict will be used as features names prefixes. Values of this dict should implement `foo(x:List) -> float` interface
- **calc_stats_on_diffs** – calculate statistics on series diffs(`np.diff(series)`) or not
- **data_preprocessing** – function implemening `foo(x) -> x_` interface. It will be used before feature calculation.
- **n_jobs** – number of threads for calculation

- **verbose** – show progress or not

calculate(*data: Dict*, *index: List[str]*) → pandas.core.frame.DataFrame
Interface to calculate features for tickers based on data

Parameters

- **data** – dict having field named as value in `data_key` param of `__init__()` This field should contain class implementing `load(index) -> pd.DataFrame` interface
- **index** – list of tickers to calculate features for, i.e. `['AAPL', 'TSLA']`

Returns resulted features with index `['ticker', 'date']`. Each row contains features for `ticker` company at `date` quarter

Return type `pd.DataFrame`

4.2 QuarterlyDiffFeatures

class ml_investment.features.**QuarterlyDiffFeatures**(*data_key: str*, *columns: List[str]*, *compare_quarter_idxs: List[int] = [1, 4]*, *max_back_quarter: int = 10*, *min_back_quarter: int = 0*, *norm: bool = True*, *data_preprocessing: Optional[Callable] = None*, *n_jobs: int = 2*, *verbose: bool = False*)

Bases: `object`

Feature calculator for qaurtr-to-another-quarter company indicators(revenue, debt etc) progress evaluation. Return features for company quarter slices.

Parameters

- **data_key** – key of dataloader in `data` argument during *calculate()*
- **columns** – column names for feature calculation(like revenue, debt etc)
- **compare_quarter_idxs** – list of back quarter idxs for progress calculation. e.g. if `compare_quarter_idxs = [1]` than current quarter will be compared with previous quarter. If `compare_quarter_idxs = [4]` than current quarter will be compared with previous year quarter.
- **max_back_quarter** – max bound of company slices in time. If `max_back_quarter = 1` than features will be calculated for only current company quarter. If max_back_quarter is larger than total number of quarters for company than features will be calculated for all quarters
- **min_back_quarter** – min bound of company slices in time. If `min_back_quarter = 0` (default) than features will be calculated for all quarters. If `min_back_quarter = 2` than current and previous quarter slices will not be used for feature calculation
- **norm** – normalize to compare quarter or not
- **data_preprocessing** – function implemening `foo(x) -> x_` interface. It will be used before feature calculation.
- **n_jobs** – number of threads for calculation
- **verbose** – show progress or not

calculate(*data: Dict*, *index: List[str]*) → pandas.core.frame.DataFrame
Interface to calculate features for tickers based on data

Parameters

- **data** – dict having field named as value in `data_key` param of `__init__()` This field should contain class implementing `load(index) -> pd.DataFrame` interface
- **index** – list of tickers to calculate features for, i.e. `['AAPL', 'TSLA']`

Returns resulted features with index `['ticker', 'date']`. Each row contains features for `ticker` company at `date` quarter

Return type `pd.DataFrame`

4.3 BaseCompanyFeatures

class ml_investment.features.**BaseCompanyFeatures**(*data_key: str*, *cat_columns: List[str]*, *verbose: bool = False*)

Bases: `object`

Feature calculator for getting base company information(sector, industry etc). Encode categorical columns via hashing label encoding. Return features for current company state.

Parameters

- **data_key** – key of dataloader in `data` argument during *calculate()*
- **cat_columns** – column names of categorical features for encoding
- **verbose** – show progress or not

calculate(*data: Dict*, *index: List[str]*) → pandas.core.frame.DataFrame
Interface to calculate features for tickers based on data

Parameters

- **data** – dict having field named as value in `data_key` param of `__init__()` This field should contain class implementing `load(index) -> pd.DataFrame` interface
- **index** – list of tickers to calculate features for, i.e. `['AAPL', 'TSLA']`

Returns resulted features with index `['ticker']`. Each row contains features for `ticker` company

Return type `pd.DataFrame`

4.4 DailyAggQuarterFeatures

class ml_investment.features.**DailyAggQuarterFeatures**(*daily_data_key: str, quarterly_data_key: str, columns: typing.List[str], agg_day_counts: typing.List[typing.Union[int, numpy.timedelta64]] = [100, 200], max_back_quarter: int = 10, min_back_quarter: int = 0, daily_index=None, stats: typing.Dict[str, typing.Callable] = {'max': <function amax>, 'mean': <function mean>, 'median': <function median>, 'min': <function amin>, 'std': <function std>}, norm: bool = True, n_jobs: int = 2, verbose: bool = False*)

Bases: `object`

Feature calculator for daily-based statistics for quarter slices. Return features for company quarter slices.

Parameters

- **daily_data_key** – key of dataloader in `data` argument during *calculate()* for daily data loading
- **quarterly_data_key** – key of dataloader in `data` argument during *calculate()* for quarterly data loading
- **columns** – column names for feature calculation(like marketcap, pe)
- **agg_day_counts** – list of days counts to calculate statistics on. e.g. if `agg_day_counts = [100, 200]` statistics will be calculated based on last 100 and 200 days(separetly).
- **max_back_quarter** – max bound of company slices in time. If `max_back_quarter = 1` than features will be calculated for only current company quarter. If max_back_quarter is larger than total number of quarters for company than features will be calculated for all quarters
- **min_back_quarter** – min bound of company slices in time. If `min_back_quarter = 0` (default) than features will be calculated for all quarters. If `min_back_quarter = 2` than current and previous quarter slices will not be used for feature calculation
- **daily_index** – indexes for `data[daily_data_key]` dataloader. If `None` than index will be the same as for `data[quarterly]`. I.e. if you want to use this class for calculating commodities features, `daily_index` may be list of interesting commodities codes. If you want want to use it i.e. for calculating daily price features, `daily_index` should be `None`
- **stats** – aggregation functions for features calculation. Should be as `Dict[str, Callable]`. Keys of this dict will be used as features names prefixes. Values of this dict should implement `foo(x:List) -> float` interface
- **norm** – normalize daily stats or not
- **n_jobs** – number of threads for calculation
- **verbose** – show progress or not

calculate(*data: Dict*, *index: List[str]*) → pandas.core.frame.DataFrame

Interface to calculate features for tickers based on data

Parameters

- **data** – dict having fields named as values in `daily_data_key` and `quarterly_data_key` params of `__init__()` This fields should contain classes implementing `load(index) -> pd.DataFrame` interfaces
- **index** – list of tickers to calculate features for, i.e. `['AAPL', 'TSLA']`

Returns resulted features with index `['ticker', 'date']`. Each row contains features for `ticker` company at `date` quarter

Return type `pd.DataFrame`

4.5 RelativeGroupFeatures

class ml_investment.features.**RelativeGroupFeatures**(*feature_calculator*, *group_data_key: str*, *group_col: str*, *relation_foo=<function RelativeGroupFeatures.<lambda>>*, *keep_group_feats=False*, *verbose: bool = False*)

Bases: `object`

Feature calculator for features relative to some group median. I.e. calculate revenue growth relative to median in sector/industry.

Parameters

- **feature_calculator** – key of dataloader in `data` argument during *calculate()* for daily data loading
- **group_data_key** – key of dataloader in `data` argument during *calculate()* for loading data having `group_col`
- **group_col** – column name for groups in which median values will be calculated
- **relation_foo** – function implementing `foo(x, y) -> z` interface. E.g. if foo = lambda x: x - y, than resulted features will be calculated as difference between current company features and group median features.
- **keep_group_feats** – return group median features or not
- **verbose** – show progress or not

calculate(*data*, *index*)

Interface to calculate features for tickers based on data

Parameters

- **data** – dict having fields named as values in `group_data_key` and necessary for `feature_calculator` keys. This fields should contain classes implementing `load(index) -> pd.DataFrame` interfaces
- **index** – index needed for `feature_calculator.calculate()`

Returns resulted features with index as in ‘’feature_calculator.calculate``.

Return type `pd.DataFrame`

4.6 FeatureMerger

class ml_investment.features.**FeatureMerger**(*fc1*, *fc2*, *on=typing.Union[str, typing.List[str]]*)

Bases: object

Feature calculator that combined two other feature calculators. Merge is executed by left.

Parameters

- **fc1** – first feature calculator implements `calculate(data: Dict, index) -> pd.DataFrame` interface
- **fc2** – second feature calculator implements `calculate(data: Dict, index) -> pd.DataFrame` interface
- **on** – columns on which merge the results of executed calculate methods

calculate(*data: Dict*, *index*) → pandas.core.frame.DataFrame

Interface to calculate features for tickers based on data

Parameters

- **data** – dict having field names needed for `fc1` and `fc2` This fields should contain classes implementing `load(index) -> pd.DataFrame` interface
- **index** – indexes dor feature calculators. I.e. if features about companies than index may be list of tickers, like `['AAPL', 'TSLA']`

Returns resulted merged features

Return type `pd.DataFrame`

CHAPTER

FIVE

TARGETS

Collection of target calculators

5.1 QuarterlyTarget

class ml_investment.targets.**QuarterlyTarget**(*data_key: str*, *col: str*, *quarter_shift: int = 0*, *n_jobs: int = 2*)

Bases: `object`

Calculator of target represented as column in quarter-based data. Work with quarterly slices of company.

Parameters

- **data_key** – key of dataloader in `data` argument during *calculate()*
- **col** – column name for target calculation(like marketcap, revenue)
- **quarter_shift** – number of quarters to shift. e.g. if `quarter_shift = 0` than value for current quarter will be returned. If `quarter_shift = 1` than value for next quarter will be returned. If `quarter_shift = -1` than value for previous quarter will be returned.

calculate(*data: Dict*, *index: pandas.core.frame.DataFrame*) → pandas.core.frame.DataFrame

Interface to calculate targets for dates and tickers in index parameter based on data

Parameters

- **data** – dict having field named as value in `data_key` param of `__init__()` This field should contain class implementing `load(index) -> pd.DataFrame` interface
- **index** – `pd.DataFrame` containing information of tickers and dates to calculate targets for. Should have columns: `["ticker", "date"]`

Returns targets having 'y' column. Index of this dataframe has the same values as `index` param. Each row contains target for `ticker` company at `date` quarter

Return type `pd.DataFrame`

5.2 QuarterlyDiffTarget

class ml_investment.targets.**QuarterlyDiffTarget**(*data_key: str*, *col: str*, *norm: bool = True*, *n_jobs: int = 2*)

Bases: object

Calculator of target represented as difference between column values in current and previous quarter. Work with quarterly slices of company.

Parameters

- **data_key** – key of dataloader in data argument during *calculate()*
- **col** – column name for target calculation(like marketcap, revenue)
- **norm** – normalize difference to previous quarter or not
- **n_jobs** – number of threads for calculation

calculate(*data: Dict*, *index: pandas.core.frame.DataFrame*) → pandas.core.frame.DataFrame

Interface to calculate targets for dates and tickers in index parameter based on data

Parameters

- **data** – dict having field named as value in data_key param of __init__() This field should contain class implementing load(index) -> pd.DataFrame interface
- **index** – pd.DataFrame containing information of tickers and dates to calculate targets for. Should have columns: ["ticker", "date"]

Returns targets having 'y' column. Index of this dataframe has the same values as index param. Each row contains target for ticker company at date quarter

Return type pd.DataFrame

5.3 QuarterlyBinDiffTarget

class ml_investment.targets.**QuarterlyBinDiffTarget**(*data_key: str*, *col: str*, *n_jobs: int = 2*)

Bases: object

Calculator of target represented as binary difference between column values in current and previous quarter. Work with quarterly slices of company.

Parameters

- **data_key** – key of dataloader in data argument during *calculate()*
- **col** – column name for target calculation(like marketcap, revenue)
- **n_jobs** – number of threads for calculation

calculate(*data: Dict*, *index: pandas.core.frame.DataFrame*) → pandas.core.frame.DataFrame

Interface to calculate targets for dates and tickers in index parameter based on data

Parameters

- **data** – dict having field named as value in data_key param of __init__() This field should contain class implementing load(index) -> pd.DataFrame interface
- **index** – pd.DataFrame containing information of tickers and dates to calculate targets for. Should have columns: ["ticker", "date"]

Returns targets having 'y' column. Index of this dataframe has the same values as `index` param. Each row contains target for `ticker` company at `date` quarter

Return type `pd.DataFrame`

5.4 DailyAggTarget

class ml_investment.targets.**DailyAggTarget**(*data_key: str*, *col: str*, *horizon: int = 100*, *foo: typing.Callable = <function mean>*, *n_jobs: int = 2*)

Bases: `object`

Calculator of target represented as aggregation function of daily values. Work with daily slices of company.

Parameters

- **data_key** – key of dataloader in `data` argument during *calculate()*
- **col** – column name for target calculation(like marketcap, pe)
- **horizon** – number of days for target calculation. If `horizon > 0` than values will be get from the future of current date. If `horizon < 0` than values will be get from the past of current date
- **foo** – function processing target aggregation
- **n_jobs** – number of threads for calculation

calculate(*data: Dict*, *index: pandas.core.frame.DataFrame*) → pandas.core.frame.DataFrame

Interface to calculate targets for dates and tickers in index parameter based on data

Parameters

- **data** – dict having field named as value in `data_key` param of `__init__()` This field should contain class implementing `load(index) -> pd.DataFrame` interface
- **index** – `pd.DataFrame` containing information of tickers and dates to calculate targets for. Should have columns: `["ticker", "date"]`

Returns targets having 'y' column. Index of this dataframe has the same values as `index` param. Each row contains target for `ticker` company at `date` day

Return type `pd.DataFrame`

5.5 DailySmoothedQuarterlyDiffTarget

class ml_investment.targets.**DailySmoothedQuarterlyDiffTarget**(*daily_data_key: str*, *quarterly_data_key: str*, *col: str*, *smooth_horizon: int = 30*, *norm: bool = True*, *n_jobs: int = 2*)

Bases: `object`

Feature calculator getting difference between current and last quarter smoothed daily column values. Work with company quarter slices.

Parameters

- **daily_data_key** – key of dataloader in `data` argument during *calculate()* for daily data loading

- **quarterly_data_key** – key of dataloader in `data` argument during *calculate()* for quarterly data loading
- **col** – column name for target calculation(like marketcap, pe)
- **smooth_horizon** – number of days for target calculation. If `smooth_horizon > 0` than values for smoothing wiil be get from future of quarter date. If `smooth_horizon < 0` than values for smoothing will be get from the past of quarter date
- **norm** – normalize result or not
- **n_jobs** – number of threads for calculation

`calculate`(*data: Dict*, *index: pandas.core.frame.DataFrame*) → pandas.core.frame.DataFrame
Interface to calculate targets for dates and tickers in index parameter based on data

Parameters

- **data** – dict having field named as value in `data_key` param of `__init__()` This field should contain class implementing `load(index) -> pd.DataFrame` interface
- **index** – `pd.DataFrame` containing information of tickers and dates to calculate targets for. Should have columns: `["ticker", "date"]`

Returns targets having 'y' column. Index of this dataframe has the same values as `index` param. Each row contains target for `ticker` company at `date` quarter

Return type `pd.DataFrame`

5.6 ReportGapTarget

`class` ml_investment.targets.**ReportGapTarget**(*data_key: str*, *col: str*, *smooth_horizon: int = 1*, *norm: bool = True*, *n_jobs: int = 2*)

Bases: `object`

Calculator of target represented as smoothed gap at some date(i.e. report date). Work with daily slices of company.

Parameters

- **data_key** – key of dataloader in `data` argument during *calculate()*
- **col** – column name for target calculation(like marketcap, pe)
- **smooth_horizon** – number of days for column smoothing
- **norm** – normalize gap value or not
- **n_jobs** – number of threads for calculation

`calculate`(*data: Dict*, *index: pandas.core.frame.DataFrame*) → pandas.core.frame.DataFrame
Interface to calculate targets for dates and tickers in index parameter based on data

Parameters

- **data** – dict having field named as value in `data_key` param of `__init__()` This field should contain class implementing `load(index) -> pd.DataFrame` interface
- **index** – `pd.DataFrame` containing information of tickers and dates to calculate targets for. Should have columns: `["ticker", "date"]`

Returns targets having 'y' column. Index of this dataframe has the same values as `index` param. Each row contains target for `ticker` company at `date` time

Return type pd.DataFrame

5.7 BaseInfoTarget

class ml_investment.targets.**BaseInfoTarget**(*data_key: str*, *col: str*)
Bases: object

Calculator of target represented by base company information

Parameters

- **data_key** – key of dataloader in data argument during *calculate()*
- **col** – column name for target calculation(like sector, industry)

calculate(*data*, *index: pandas.core.frame.DataFrame*) → pandas.core.frame.DataFrame
Interface to calculate targets for tickers in index parameter based on data

Parameters

- **data** – dict having field named as value in data_key param of __init__() This field should contain class implementing load(index) -> pd.DataFrame interface
- **index** – pd.DataFrame containing information of tickers to calculate targets for. Should have columns: ["ticker"]

Returns targets having 'y' column. Index of this dataframe has the same values as index param. Each row contains target for ticker company

Return type pd.DataFrame

CHAPTER

SIX

MODELS

Collection of wrappers for machine learning models

6.1 LogExpModel

class ml_investment.models.**LogExpModel**(*base_model*)

Bases: object

Model wrapper to fit on log of target and exp produced prediction. May be usefull for some target distributions.

Parameters **base_model** – class implements `fit(X, y)`, `predict(X)`/`predict_proba(X)` interfaces

fit(*X: pandas.core.frame.DataFrame*, *y*)

Interface for model training

Parameters

- **X** – `pd.DataFrame` containing features
- **y** – target data

predict(*X*)

Interface for prediction

Parameters **X** – `pd.DataFrame` containing features

6.2 EnsembleModel

class ml_investment.models.**EnsembleModel**(*base_models: List*, *bagging_fraction: float = 0.8*, *model_cnt: int = 20*)

Bases: object

Class for training ansamble of base models.

Parameters

- **base_models** – list of classes implements `fit(X, y)`, `predict(X)`/`predict_proba(X)` interfaces
- **bagging_fraction** – part of random data subsample for training models
- **model_cnt** – total number of models in resulted ansamble

fit(*X: pandas.core.frame.DataFrame*, *y: pandas.core.series.Series*)
Interface for model training

Parameters

- **X** – pd.DataFrame containing features
- **y** – target data

predict(*X*)
Interface for prediction

Parameters **X** – pd.DataFrame containing features

6.3 GroupedOOFModel

class ml_investment.models.**GroupedOOFModel**(*base_model*, *group_column: str*, *fold_cnt: int = 5*)
Bases: object

Model wrapper incapsulate out of fold separation within data groups. Each sample in group can not be in training and validation fold at the same time.

Parameters

- **base_model** – model implements fit(X, y), predict(X)/predict_proba(X) interfaces
- **group_column** – name of column for grouping training data. X in fit(X, y) and predict(X) should contain this column. Samples with one group value will be placed only in one training fold.
- **fold_cnt** – number of folds for training

fit(*X: pandas.core.frame.DataFrame*, *y: pandas.core.series.Series*)
Interface for model training

Parameters

- **X** – pd.DataFrame containing features and self.group_column
- **y** – target data

predict(*X: pandas.core.frame.DataFrame*) → numpy.array
Interface for prediction

Parameters **X** – pd.DataFrame containing features and self.group_column

6.4 TimeSeriesOOFModel

class ml_investment.models.**TimeSeriesOOFModel**(*base_model*, *time_column: str*, *fold_cnt: int = 5*)
Bases: object

Model wrapper incapsulate out of fold time-series separation.

Parameters

- **base_model** – model implements fit(X, y), predict(X)/predict_proba(X) interfaces

- **time_column** – name of column for separating training data. `X` in `fit(X, y)` and `predict(X)` should contain this column. Samples from feature would not be used for training and prediction past.
- **fold_cnt** – number of folds for training

fit(*X: pandas.core.frame.DataFrame*, *y*)
Interface for model training

Parameters

- **X** – `pd.DataFrame` containing features and `self.time_column`
- **y** – target data

predict(*X: pandas.core.frame.DataFrame*) → numpy.array
Interface for prediction

Parameters **X** – `pd.DataFrame` containing features and `self.time_column`

CHAPTER

SEVEN

PIPELINES

Collection of pipelines

7.1 Pipeline

class ml_investment.pipelines.**Pipeline**(*data: Dict*, *feature*, *target*, *model*, *out_name=None*)
Bases: object

Class incapsulate feature and target calculation, model training and validation during fit-phase and feature calculation and model prediction during execute-phase. Support multi-target with different models and metrics.

Parameters

- **data** – dict having needed for features and targets fields. This field should contain classes implementing `load(index) -> pd.DataFrame` interfaces
- **feature** – feature calculator implements `calculate(data: Dict, index) -> pd.DataFrame` interface
- **target** – target calculator implements `calculate(data: Dict, index) -> pd.DataFrame` interface OR `List` of such target calculators
- **model** – class implements `fit(X, y)` and `predict(X)` interfaces. opy of the model will be used for every single target if type of target is `List`. OR `List` of such classes(len of this list should be equal to len of target)
- **out_name** – str column name of result in `pd.DataFrame` after *execute()* OR `List[str]` (len of this list should be equal to len of target) OR `None` (`List['y_0', 'y_1'...]` will be used in this case)

execute(*index*)
Interface for executing pipeline for tickers. Features will be based on data from data_loader

Parameters **index** – execute identification(i.e. list of tickers to predict model for)

Returns result values in columns named as `out_name` param in `__init__()`

Return type `pd.DataFrame`

export_core(*path=None*)
Interface for saving pipelines core

Parameters **path** – str with path to store pipeline core OR `None` (path will be generated automatically)

fit(*index: typing.List[str], metric=None, target_filter_foo=<function nan_mask>*)
Interface to fit pipeline model for tickers. Features and target will be based on data from data_loader

Parameters

- **index** – fit identification(i.e. list of tickers to fit model for)
- **metric** – function implements `foo(gt, y) -> float` interface. The same metric will be used for every single target if type of target is `List`. OR `List` of such functions(len of this list should be equal to len of target)
- **target_filter_foo** – function for filtering samples according target values/ Should implement `foo(arr) -> np.array[bool]` interface. Len of resulted array should be equal to len of arr. OR `List` of such functions(len of this list should be equal to len of target)

load_core(*path*)

Interface for loading pipeline core

Parameters **path** – str with path to load pipeline core from

7.2 MergePipeline

class ml_investment.pipelines.**MergePipeline**(*pipeline_list: List*, *execute_merge_on*)

Bases: `object`

Class combining list of pipelines to single pipilene.

Parameters

- **pipeline_list** – list of classes implementing `fit(index)` and `execute(index) -> pd.DataFrame()` interfaces. Order is important: merging results during *execute()* will be done from left to right.
- **execute_merge_on** – column names for merging pipelines results on.

execute(*index*, *batch_size=None*) → pandas.core.frame.DataFrame

Interface for executing pipeline for tickers. Features will be based on data from data_loader

Parameters

- **index** – identifiers for executing pipelines. I.e. list of companies tickers
- **batch_size** – size of batch for execute separation(may be usefull for lower memory usage). OR `None` (for full-size executing)

Returns combined pipelines execute result

Return type `pd.DataFrame`

fit(*index*)

Interface for training all pipelines

Parameters **index** – identifiers for fit pipelines. I.e. list of companies tickers

7.3 LoadingPipeline

class ml_investment.pipelines.**LoadingPipeline**(*data_loader*, *columns: List[str]*)

Bases: object

Wrapper for data loaders for loading data in execute(index) -> pd.DataFrame interface

Parameters

- **data_loader** – class implements load(index) -> pd.DataFrame interface
- **columns** – column names for loading

execute(*index*)

Interface for executing pipeline(lading data) for tickers.

Parameters **index** – inentification for loading data, i.e. list of tickers

Returns resulted data

Return type pd.DataFrame

fit(*index*)

CHAPTER

EIGHT

DATA LOADERS

Collection of data loaders and utils for it

8.1 Yahoo

Loader for dataset provided by yahoo. Data may be downloaded by script *main()*

Expected dataset structure:

path to Yahoo data folder with structure

```
Yahoo
├── quarterly
│   ├── AAPL.csv
│   ├── FB.csv
│   └── ...
├── base
      ├── AAPL.json
      ├── FB.json
      └── ...
```

class ml_investment.data_loaders.yahoo.**YahooBaseData**(*data_path: str*)

Bases: object

Loader for base information about company(like sector, industry etc)

Parameters **data_path** – path to *yahoo* dataset folder

load(*index: Optional[List[str]] = None*) → pandas.core.frame.DataFrame

Parameters **index** – list of tickers to load data for OR None (for loading all possible tickers)

Returns base companies information

Return type pd.DataFrame

class ml_investment.data_loaders.yahoo.**YahooQuarterlyData**(*data_path: str*, *quarter_count: Optional[int] = None*)

Bases: object

Loader for quartely fundamental information about companies(debt, revenue etc)

Parameters

- **data_path** – path to *yahoo* dataset folder

- **quarter_count** – maximum number of last quarters to return. Resulted number may be less due to short history in some companies

load(*index: List[str]*) → pandas.core.frame.DataFrame

Parameters **index** – list of tickers to load data for

Returns quarterly information about companies

Return type pd.DataFrame

8.2 SF1

Loaders for dataset provided by https://www.quandl.com/databases/SF1/data. Data may be downloaded by script *main()*

Expected structure of dataset

```
SF1
├── core_fundamental
│   ├── AAPL.json
│   ├── FB.json
│   └── ...
├── daily
│   ├── AAPL.json
│   ├── FB.json
│   └── ...
└── tickers.zip
```

class ml_investment.data_loaders.sf1.**SF1BaseData**(*data_path: Optional[str] = None*)

Bases: object

Load base information about company(like sector, industry etc)

Parameters **data_path** – path to *sf1* dataset folder If None, than will be used sf1_data_path from *~/.ml_investment/config.json*

existing_index()

Returns existing index values that can pe pushed to *load*

Return type List

load(*index: Optional[List[str]] = None*) → pandas.core.frame.DataFrame

Parameters **index** – list of ticker to load data for, i.e. ['AAPL', 'TSLA'] OR None (loading for all possible tickers)

Returns base companies information

Return type pd.DataFrame

class ml_investment.data_loaders.sf1.**SF1DailyData**(*data_path: Optional[str] = None*, *days_count: Optional[int] = None*)

Bases: object

Load daily information about company(marketcap, pe etc)

Parameters

- **data_path** – path to *sf1* dataset folder If None, than will be used `sf1_data_path` from *~/.ml_investment/config.json*
- **days_count** – maximum number of last days to return. Resulted number may be less due to short history in some companies

existing_index()

Returns existing index values that can pe pushed to *load*

Return type `List`

load(*index: List[str]*) → pandas.core.frame.DataFrame

Parameters **index** – list of ticker to load data for, i.e. `['AAPL', 'TSLA']`

Returns daily information about companies

Return type `pd.DataFrame`

class ml_investment.data_loaders.sf1.**SF1QuarterlyData**(*data_path: Optional[str] = None*, *quarter_count: Optional[int] = None*, *dimension: Optional[str] = 'ARQ'*)

Bases: `object`

Loader for quartely fundamental information about companies(debt, revenue etc)

Parameters

- **data_path** – path to *sf1* dataset folder If None, than will be used `sf1_data_path` from *~/.ml_investment/config.json*
- **quarter_count** – maximum number of last quarters to return. Resulted number may be less due to short history in some companies
- **dimension** – one of `['MRY', 'MRT', 'MRQ', 'ARY', 'ART', 'ARQ']`. SF1 dataset-based parameter

existing_index()

Returns existing index values that can pe pushed to *load*

Return type `List`

load(*index: List[str]*) → pandas.core.frame.DataFrame

Parameters **index** – list of tickers to load data for, i.e. `['AAPL', 'TSLA']`

Returns quarterly information about companies

Return type `pd.DataFrame`

class ml_investment.data_loaders.sf1.**SF1SNP500Data**(*data_path: Optional[str] = None*)

Bases: `object`

S&P500 historical constituents

Parameters **data_path** – path to *sf1* dataset folder If None, than will be used `sf1_data_path` from *~/.ml_investment/config.json*

existing_index()

Returns existing index values that can pe pushed to *load*

Return type List

load(*index: Optional[List[numpy.datetime64]] = None*) → pandas.core.frame.DataFrame

Parameters **index** – list of dates to load constituents for, i.e. [np.datetime64('2018-01-01'), np.datetime64('2018-05-10')] If there are no such date, than nearest past date will be used. OR None (loading for all dates when constituents was changed)

Returns constituents information

Return type pd.DataFrame

ml_investment.data_loaders.sf1.**translate_currency**(*df: pandas.core.frame.DataFrame*, *columns: Optional[List[str]] = None*)

Translate currency of columns to USD according course information in appropriate columns(like debtusd-debt)

Parameters

- **df** – quarterly-based data
- **columns** – columns to translate currency

Returns result with the same columns and shapes but with converted currency in columns

Return type pd.DataFrame

8.3 Quandl Commodities

Loader for commodities price information from https://blog.quandl.com/api-for-commodity-data. Data may be downloaded by script *main()*

Expected dataset structure

commodities
├— LBMA_GOLD.json
├— CHRIS_CME_CL1.json
└— ...

class ml_investment.data_loaders.quandl_commodities.**QuandlCommoditiesData**(*data_path: Optional[str] = None*)

Bases: object

Loader for commodities price information.

data_path: path to *quandl_commodities* dataset folder If None, than will be used commodities_data_path from *~/.ml_investment/config.json*

existing_index()

Returns existing index values that can pe pushed to *load*

Return type List

load(*index: List[str]*) → pandas.core.frame.DataFrame
Load time-series information about commodity price

Parameters **index** – list of commodities codes to load data for, i.e. `['LBMA/GOLD', 'JOHNMATT/PALL']`

Returns time series price information

Return type `pd.DataFrame`

8.4 Daily Price Bars

Loader for daily bars price information. Data may be downloaded by script *main()*

Expected dataset structure

daily_bars
├── AAPL.csv
├── TSLA.csv
└── ...

class `ml_investment.data_loaders.daily_bars.`**DailyBarsData**(*data_path: Optional[str] = None*, *days_count: Optional[int] = None*)

Bases: `object`

Loader for daywise price bars.

Parameters

- **data_path** – path to *daily_bars* dataset folder If None, than will be used `daily_bars_data_path` from *~/.ml_investment/config.json*
- **days_count** – maximum number of last days to return. Resulted number may be less due to short history in some companies

existing_index()

Returns existing index values that can pe pushed to *load*

Return type `List`

load(*index: List[str]*) → pandas.core.frame.DataFrame

Load daily price bars

Parameters **index** – list of tickers to load data for, i.e. `['AAPL', 'TSLA']`

Returns daily price bars

Return type `pd.DataFrame`

8.5 Data loading utils

CHAPTER

NINE

DOWNLOADING SCRIPTS

Collection of scripts for data downloading from different sources

9.1 SF1

ml_investment.download_scripts.download_sf1.**main**(*data_path: str = '/home/docs/.ml_investment/data/sf1'*, *verbose: bool = False*)

Download quarterly fundamental data from https://www.quandl.com/databases/SF1/data

Note: SF1 is paid, so you need to subscribe and paste quandl token to *~/.ml_investment/secrets.json* `quandl_api_key`

Parameters

- **data_path** – path to folder in which downloaded data will be stored. OR `None` (downloading path will be as `sf1_data_path` from *~/.ml_investment/config.json*
- **verbose** – show progress or not

9.2 Yahoo

ml_investment.download_scripts.download_yahoo.**main**(*data_path: Optional[str] = None*)

Download quarterly and base data from https://finance.yahoo.com

Parameters **data_path** – path to folder in which downloaded data will be stored. OR `None` (downloading path will be as `yahoo_data_path` from *~/.ml_investment/config.json*

9.3 Daily price bars

ml_investment.download_scripts.download_daily_bars.**main**(*data_path: str = '/home/docs/.ml_investment/data/daily_bars', tickers: Optional[List] = ['OKTA', 'HYLN', 'RSTI', 'CHE', 'WHD', 'USPH', 'TRHC', 'FGEN', 'JD', 'BLNK', 'IRDM', 'FOCS', 'IBM', 'LANC', 'GLW', 'FITB', 'TPTX', 'EXPE', 'UHS', 'FCNCA', 'JBT', 'DRQ', 'RRBI', 'CHWY', 'DGX', 'VXRT', 'CCK', 'PHM', 'SJM', 'XNCR', 'DLB', 'BWA', 'SITE', 'LAD', 'MCHP', 'YUM', 'BOX', 'LHCG', 'BBIO', 'GPI', 'BMRN', 'PII', 'GDDY', 'MLM', 'WORK', 'INTC', 'CHGG', 'CWST', 'RACE', 'ASIX', 'NJR', 'AEE', 'DKS', 'SLP', 'ABMD', 'TE', 'COF', 'PBH', 'OSK', 'BR', 'COWN', 'PRSP', 'RGR', 'CRL', 'SLDB', 'LYB', 'IIVI', 'AYX', 'CSCO', 'ROK', 'WYNN', 'ARE', 'APEI', 'CLR', 'BECN', 'IR', 'EPAY', 'TREE', 'BLL', 'BDC', 'RCL', 'AFL', 'WWW', 'XPO', 'NYT', 'FORR', 'EMN', 'AES', 'PPL', 'ADPT', 'LMT', 'RGEN', 'IART', 'FDX', 'GE', 'OGE', 'SPCE', 'CMCO', 'QLYS', 'VIPS', 'MCD', 'ALXN', 'BLK', 'KLAC', 'AMWD', 'FUL', 'RAVN', 'TM', 'CDNA', 'SYF', 'LLY', 'INCY', 'MU', 'TTMI', 'FTV', 'CMA', 'EEFT', 'ATRA', 'ARCT', 'PB', 'YUMC', 'DASH', 'IBP', 'SI', 'MMM', 'CCOI', 'LRN', 'TT', 'BJRI', 'CARG', 'TREX', 'NVS', 'DKNG', 'TSS', 'ALLY', 'CVLT', 'EPAM', 'LDOS', 'NSC', 'EWBC', 'SCI', 'WKHS', 'GHC', 'EBAY', 'MO', 'MDGL', 'VFC', 'MA', 'FLOW', 'CACC', 'PPG', 'VALE', 'DRE', 'NP', 'AGIO', 'YEXT', 'OII', 'CFX', 'GRA', 'AWI', 'DOCU', 'PFE', 'A', 'AVGO', 'QTS', 'PM', 'OSUR', 'PATK', 'INSP', 'GEF', 'DAL', 'KMX', 'CIEN', 'GD', 'SF', 'AVLR', 'MED', 'MDLZ', 'ABT', 'GMS', 'DOV', 'BLKB', 'COKE', 'BLUE', 'CMS', 'VREX', 'MANT', 'ZEN', 'SBAC', 'DVN', 'HNP', 'PCG', 'CHTR', 'GTN', 'SRI', 'SXT', 'NET', 'ALRS', 'SYNH', 'SFM', 'JNJ', 'DG', 'RXN', 'SDGR', 'ALB', 'ITW', 'PRTK', 'BEN', 'PSX', 'RTX', 'SAVA', 'UNF', 'LSTR', 'AZPN', 'OHI', 'ALV', 'COUP', 'EIX', 'KEYS', 'PKG', 'WELL', 'ILMN', 'WH', 'PFGC', 'CVM', 'AIZ', 'CCXI', 'ANF', 'GT', 'WMB', 'WEC', 'AVNT', 'ROG', 'BKR', 'CRTX', 'GPC', 'CEA', 'ACH', 'NVDA', 'MORN', 'LNTH', 'PTC', 'CGNT', 'EAR', 'MYGN', 'PEGA', 'SAFM', 'HLI', 'SRE', 'STZ', 'IOSP', 'NTGR', 'PAGS', 'GDOT', 'CNXC', 'XEC', 'Y', 'PNC', 'CABO', 'OLLI', 'J', 'TGT', 'TPH', 'NFE', 'DLTR', 'CW', 'VRNS', 'XRX', 'SIG', 'BDTX', 'CL', 'T', 'NVTA', 'SMTC', 'BBBY', 'CFG', 'VRSK', 'NARI', 'TW', 'DIS', 'TAP', 'QTWO', 'PLTR', 'CHNG', 'COLD', 'ABBV', 'JELD', 'UBER', 'CLSK', 'STE', 'ZUO', 'STLD', 'HAL', 'HQY', 'GS', 'FTDR', 'ABC', 'ARQT',*

Parameters

- **data_path** – path to folder in which downloaded data will be stored. OR None (downloading path will be as daily_bars_data_path from *~/.ml_investment/config.json*
- **tickers** – tickers to download daily bars for
- **from_date** – start date for loading data
- **to_date** – end day for loading data
- **verbose** – show progress or not

9.4 Commodities

ml_investment.download_scripts.download_commodities.**main**(*data_path: str = '/home/docs/.ml_investment/data/commodities'*, *verbose: bool = False*)

Download commodities price history from https://blog.quandl.com/api-for-commodity-data

Note: To download this dataset you need to register at quandl and paste token to *~/.ml_investment/secrets.json*

Parameters

- **data_path** – path to folder in which downloaded data will be stored. OR None (downloading path will be as commodities_data_path from *~/.ml_investment/config.json*
- **verbose** – show progress or not

CHAPTER

TEN

BACKTEST

Backtesting utils

10.1 Strategy

class ml_investment.backtest.strategy.**Strategy**

Bases: object

Base class for strategy backtesting. It contains overrideble method `step` for defining user strategy. This class incapsulate backtesting and metrics calculation process and also contains information about orders.

backtest(*data_loader*, *date_col: str*, *price_col: str*, *return_col: str*, *return_format: str*, *step_dates: Optional[List[numpy.timedelta64]] = None*, *cash: float = 100000*, *comission: float = 0.00025*, *latency: numpy.timedelta64 = numpy.timedelta64(0, 'h')*, *allow_short: bool = False*, *metrics=None*, *preload: bool = False*, *verbose: bool = True*)

Backtest strategy on provided data and other parameters. It will create and execute orders and calculate resulted equity and metrics.

Parameters

- **data_loader** – class implementing `load(index) -> pd.DataFrame` interface. index in this case is list of tickers to load market data for.
- **date_col** – name of column containing date (time) information in market data provided by `data_loader`.
- **price_col** – name of column containing price information in market data provided by `data_loader`.
- **return_col** – name of column containing total return information in data provided by `data_loader`. It may be differ from price due to dividends, stock splits and etc.
- **return_format** – format of data provided by `return_col` column. If `return_format = 'ratio'` than column should contain ratio between previous and current adjusted price. E.g. 1.2 means growth by 20% from the previous step. If `return_format = 'price'` than column should contain adjusted price (price, including dividends and etc.) If `return_format = 'change'` than column should contain relative change between current and previous step. E.g. 0.2 means growth by 20% from the previous step.
- **step_dates** – dates in which all actions can be taken. Include new market prices receiving, order creation and executing. `step` method will iterate over all those dates. If None than all possible dates, provided by `date_col` column in `data_loader` will be used. Possible only if `preload = True` and `data_loader` have `existing_index(index) -> List` interface.

- **cash** – initial amount of cash
- **comission** – commission charged for each trade (in percent of order value)
- **latency** – time between current step date and actual order posting. It emulates delays during `step` logic and in the Internet connection with the exchange.
- **allow_short** – allow short positions or not
- **preload** – load all data provided from `data_loader` to ram or not
- **verbose** – show progress or not

calc_metrics(*metrics: Dict*)

post_order(*ticker: str*, *direction: int*, *size: float*, *order_type: int = 0*, *lifetime: numpy.timedelta64 = numpy.timedelta64(300, 'D')*, *allow_partial: bool = True*)

Post new order to backtest. It may be used inside your strategy overriden `step` method.

Parameters

- **ticker** – ticker of company to post order for
- **direction** – one of `Order.BUY` (1), `Order.SELL` (-1)
- **size** – size of order in pieces
- **order_type** – one of `Order.MARKET` (0), `Order.LIMIT` (1)
- **lifetime** – amount of time before order closing if it can not be executed (e.g. if unsatisfactory price lasts a long time)
- **allow_partial** – may order be executed with not full size or not

post_order_value(*ticker: str*, *direction: int*, *value: float*, *order_type: int = 0*, *lifetime: numpy.timedelta64 = numpy.timedelta64(300, 'D')*, *allow_partial: bool = True*)

Post new order by value (instead of size) to backtest. It may be used inside your strategy overriden `step` method.

Parameters

- **ticker** – ticker of company to post order for
- **direction** – one of `Order.BUY` (1), `Order.SELL` (-1)
- **value** – value of order in money
- **order_type** – one of `Order.MARKET` (0), `Order.LIMIT` (1)
- **lifetime** – amount of time before order closing if it can not be executed (e.g. if unsatisfactory price lasts a long time)
- **allow_partial** – may order be executed with not full size or not

post_portfolio_part(*ticker: str*, *part: float*, *lifetime: numpy.timedelta64 = numpy.timedelta64(300, 'D')*, *allow_partial: bool = True*)

Post order to backtest to have desired part in portfolio. It will calculate difference between current and desired part to create appropriate order. It may be used inside your strategy overriden `step` method.

Parameters

- **ticker** – ticker of company to post order for
- **part** – desired part in all equity including other stocks and cash in portfolio (value between 0 and 1)

- **lifetime** – amount of time before order closing if it can not be executed (e.g. if unsatisfactory price lasts a long time)
- **allow_partial** – may order be executed with not full size or not

post_portfolio_size(*ticker: str*, *size: int*, *lifetime: numpy.timedelta64 = numpy.timedelta64(300, 'D')*, *allow_partial: bool = True*)

Post order to backtest to have desired size in portfolio. It will calculate difference between current and desired size to create appropriate order. It may be used inside your strategy overriden `step` method.

Parameters

- **ticker** – ticker of company to post order for
- **size** – desired size in portfolio (in pieces)
- **lifetime** – amount of time before order closing if it can not be executed (e.g. if unsatisfactory price lasts a long time)
- **allow_partial** – may order be executed with not full size or not

post_portfolio_value(*ticker: str*, *value: float*, *lifetime: numpy.timedelta64 = numpy.timedelta64(300, 'D')*, *allow_partial: bool = True*)

Post order to backtest to have desired value in portfolio. It will calculate difference between current and desired value to create appropriate order. It may be used inside your strategy overriden `step` method.

Parameters

- **ticker** – ticker of company to post order for
- **value** – desired value in portfolio (in money)
- **lifetime** – amount of time before order closing if it can not be executed (e.g. if unsatisfactory price lasts a long time)
- **allow_partial** – may order be executed with not full size or not

step()

CHAPTER

ELEVEN

INDICES AND TABLES

- genindex
- modindex
- search

PYTHON MODULE INDEX

m

INDEX

P

Q

R

S

T

Y