risk_dash Documentation

Release 0.0.2

Alexander van Oene

Jul 27, 2018

CONTENTS:

RISK_DASH

1.1 Overview

risk_dash is a framework to help simplify the data flow for a portfolio of assets and handle market risk metrics at the asset and portfolio level. If you clone the source repository, included is a Dash application to be an example of some of the uses for the package. To run the Dash app, documentation is here

1.2 Installation

Since the package is in heavy development, to install the package fork or clone the repository and run `pip install -e risk_dash/` from the directory above your local repository.

To see if installation was successful run `python -c 'import risk_dash; print(*dir(risk_dash), sep="\n")'` in the command line, currently the output should match the following:

```
$ python -c 'import risk_dash; print(*dir(risk_dash), sep="\n")'
__builtins__
__cached__
__doc__
__file__
__loader__
__name__
```

```
__package__
__path__
__spec__
market_data
name
securities
simgen
```

1.3 Getting Started

Now that we have the package installed, let's go through the object workflow to construct a simple long/short equity portfolio.

High level, we need to specify:

1. Portfolio Data

   - We need to know what's in the portfolio

     - Portfolio weights

     - Types of Assets/Securities

2. Security data

   - We need to know what is important to financially model the security

     - Identification data: Ticker, CUSIP, Exchange

     - Security specific data: expiry, valuation functions

     - Market data: Closing prices, YTM

3. Portfolio/security constructors to handle the above data

To visualize these constructors, the below chart shows how the data will sit:
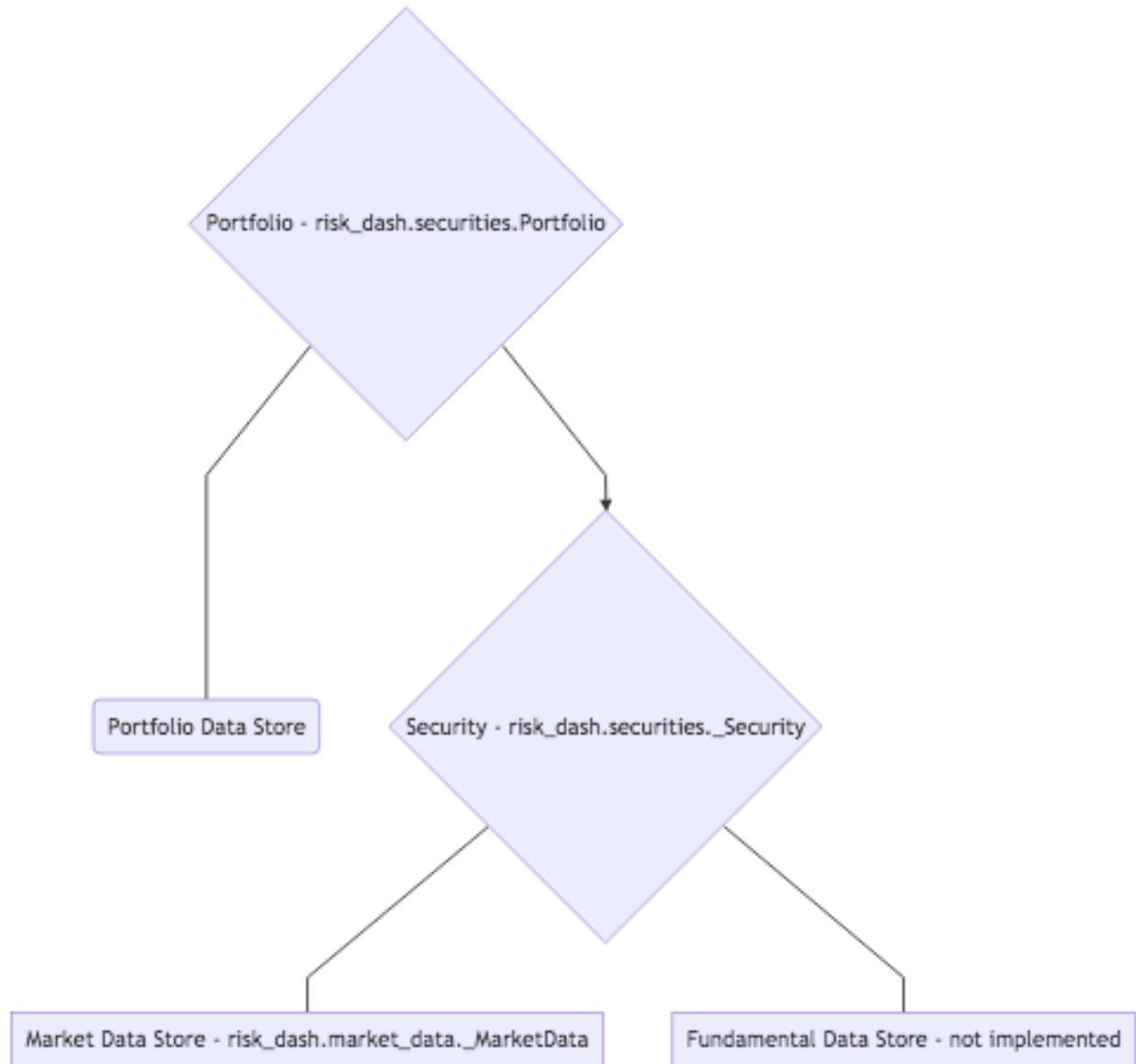
To do so, we'll need subclasses for the *_Security* and [_MarketData](#) classes to model specific types of securities. Currently supported is the Equity subclass. Once we have the portfolio constructed, we will specify and calculate parameters to simulate or look at historic distributions. We'll then create a subclass of *_Simulation* and *_RandomGen*

1.3.1 Security data, `_Security` objects, and creating Security Subclasses

The core of the package is in the `_Security` and `Portfolio` objects. `Portfolio` objects are naturally a collection of Securities, however we want to specify the type of securities that are in the portfolio. Since we're focusing on a long/short equity portfolio we want to create an Equity subclass.

Subclasses of `_Security` classes must have the following methods:

- valuation(current_price)

- mark_to_market(current_price)

- get_marketdata()

In addition, we want to pass them the associated `_MarketData` object to represent the security's historic pricing data. To build the `Equity` subclass, we first want to inherit any methods from the `_Security` class:

```python
class Equity(_Security):

    def __init__(
            self,
            ticker,
            market_data : md.QuandlStockData,
            ordered_price,
            quantity,
            date_ordered
        ):
        self.name = ticker
        self.market_data = market_data
        self.ordered_price = ordered_price
        self.quantity = quantity
        self.initial_value = ordered_price * quantity
        self.date_ordered = date_ordered
        self.type = 'Equity'
```

To break down the inputs, we want to keep in mind that the goal of this subclass of the _Security object is to provide an interface to model the Equity data.

- ticker is going to be the ticker code for the equity, such as 'AAPL'

- market_data is going to be a subclass of the _MarketData object

- ordered_price is going to be the price which the trade occurred

- quantity for Equity will be the number of shares

- date_ordered should be the date the order was placed

Note: Currently the implemented _MarketData subclass is QuandlStockData, which is a wrapper for this Quandl dataset api. This data is no longer being updated, for current market prices you must create a _MarketData subclass for your particular market data. Information to construct the subclass is on Building Custom Classes.

Required Inputs at the _Security level are intentionally limited, for example if we wanted to create a class for Fixed Income securities, we would want more information than this Equity subclass. An example Bond class might look like this:

```python
class Bond(_Security):
    def __init__(
            self,
            CUSIP,
            market_data,
            expiry,
            coupon,
            frequency,
            settlement_date,
            face_value
        ):
        self.name = CUSIP
        self.market_data = market_data
        self.expiry = expiry
        self.coupon = coupon
        self.frequency = frequency
```

(continues on next page)

```
        self.settlement_date = settlement_date
        self.face_value = face_value
        self.type = 'Bond'
```

Similarly to the Equity subclass, we want identification information, market data, and arguments that will either help in calculating valuation, current returns, or risk measures.

Returning to the Equity subclass, we now need to write the valuation and mark to market methods:

```
class Equity(_Security):
  # ...
  def valuation(self, price):
      value = (price - self.ordered_price) * self.quantity
      return(value)

  def mark_to_market(self, current_price):
      self.market_value = self.quantity * current_price
      self.marked_change = self.valuation(current_price)
      return(self.marked_change)
```

For linear instruments such as equities, valuation of a position is just the price observed minus the price ordered at the size of the position. `valuation` is then used to pass a hypothetical price into the valuation function, in this case (Price - Ordered) * Quantity, where as `mark_to_market` is used to pass the current EOD price and mark the value of the position. This is an important distinction, if we had a nonlinear instrument such as a call option on a company's equity price, the valuation function would then be:

$$Value = min\{0, S_T - K\}$$

Where S_T is the $spot$ $price$ for the $equity$ at $expiry$ and K is the $agreed$ $strike$ $price$. $Valuation$ $also$ is $dependent$ on $time$ for $option$ $data$, 0.

Our mark to market then would need to make the distinction between this valuation and the current market price for the call option. The mark would then keep track of what the current market value for the option to keep track of actualized returns.

The final piece to creating the Equity subclass is then to add a `get_marketdata()` method. Since we just want a copy of the reference of the `market_data`, we can just inherit the `get_marketdata()` from the _Security class.

The Equity subclass is already implemented in the package, we can create an instance from `risk_dash.securities`. Let's make an instance that represents an order of 50 shares of AAPL, Apple Inc, at close on March 9th, 2018:

```
>>> from risk_dash.market_data import QuandlStockData
>>> from risk_dash.securities import Equity
>>> from datetime import datetime
>>> apikey = 'valid-quandl-apikey'
>>> aapl_market_data = QuandlStockData(
  apikey = apikey,
  ticker = 'AAPL'
)
>>> aapl_stock = Equity(
  ticker = 'AAPL',
  market_data = aapl_market_data,
  ordered_price = 179.98,
  quantity = 50,
  date_ordered = datetime(2018,3,9)
)
```

```
>>> aapl_stock.valuation(180.98) # $1 increase in value
50.0
>>> aapl_stock.mark_to_market(180.98) # Same $1 increase
50.0
>>> aapl_stock.market_value
9049.0
>>> aapl_stock.marked_change
50.0
>>> vars(aapl_stock)
{'name': 'AAPL',
 'market_data': <risk_dash.market_data.QuandlStockData at 0x1147c2668>,
 'ordered_price': 179.98,
 'quantity': 50,
 'initial_value': 8999.0,
 'date_ordered': datetime.datetime(2018, 3, 9, 0, 0),
 'type': 'Equity',
 'market_value': 9049.0,
 'marked_change': 50.0}
```

As we can see `aapl_stock` now is a container that we can use to access it's attributes at the Portfolio level.

Note: Another important observation is that the Equity subclass will only keep a reference to the underlying Quandl-StockData, which will minimize duplication of data. However, at scale, you'd want minimize price calls to your data source, you could then do one call at the Portfolio level then pass a reference to that market_data at the individual level. Then your Equity or other _Security subclasses can share the same _MarketData, you would then just write methods to interact with that data.

Now that we have a feeling for the _Security class, we now want to build a Portfolio that contains the _Security instances.

1.3.2 Portfolio Data and creating a Portfolio

To iterate on what we said before, an equity position in your portfolio is represented by the quantity you ordered, the price ordered at, and when you ordered or settled the position. In this example, we'll use the following theoretical portfolio found in `portfolio_example.csv`:

Type	Ticker	Ordered Price	Ordered Date	Quantity
Equity	AAPL	179.98	3/9/18	50
Equity	AMD	11.7	3/9/18	100
Equity	INTC	52.19	3/9/18	-50
Equity	GOOG	1160.04	3/9/18	5

With this example, the portfolio is static, or just one snap shot of the weights at a given time. In practice, it might be useful to have multiple snapshots of your portfolio, one's portfolio would be changing as positions enter and leave thus having a time dimensionality. The Portfolio class could be easily adapted to handle that information to accurately plot historic performance by remarking through time. This seems more of an accounting exercise, risk metrics looking forward would probably still only want to account for the current positions in the portfolio. Due to this insight, the current Portfolio class only looks at one snap shot in time.

With a portfolio so small, it is very easily stored in a csv and each security can store the reference to the underlying market data independently. As such, there is an included portfolio constructor method in the portfolio class from csv, `construct_portfolio_csv`:

```
>>> from risk_dash.securities import Portfolio
>>> current_portfolio = Portfolio()
>>> port_dict = current_portfolio.construct_portfolio_csv(
  data_input='portfolio_example.csv',
  apikey=apikey
)
>>> vars(current_portfolio)
{'port': {'AAPL Equity': <risk_dash.securities.Equity at 0x11648b5c0>,
  'AMD Equity': <risk_dash.securities.Equity at 0x116442c50>,
  'INTC Equity': <risk_dash.securities.Equity at 0x1177b75c0>,
  'GOOG Equity': <risk_dash.securities.Equity at 0x1177bc390>}}
>>> vars(current_portfolio.port['AMD Equity'])
{'name': 'AMD',
 'market_data': <risk_dash.market_data.QuandlStockData at 0x11648b2e8>,
 'ordered_price': 11.699999999999999,
 'quantity': 100,
 'initial_value': 1170.0,
 'date_ordered': '3/9/18',
 'type': 'Equity'}
```

At this moment, the `current_portfolio` instance is only a wrapper for it's port attribute, a dictionary containing the securities in the Portfolio object. Soon we'll use this object to mark the portfolio, create a simulation to estimate value at risk, look at the covariance variance matrix to calculate a parameterized volatility measure, and much more.

The `Portfolio` class handles interactions with the portfolio data and the associated securities in the portfolio. If you have a list of securities you can also just pass the list into the Portfolio instance. The following code creates a portfolio of just the AAPL equity that we created earlier:

```
>>> aapl_portfolio = sec.Portfolio([aapl_stock])
>>> vars(aapl_portfolio)
{'port': {'AAPL Equity': <risk_dash.securities.Equity at 0x1164b2e80>}}
```

If we want to add a security to this portfolio, we can call the `add_security` method, to remove a security we call the `remove_security` method:

```
>>> amd_market_data = QuandlStockData(
  ticker='AMD',
  apikey=apikey
)
>>> amd_stock = Equity(
  ticker = 'AAPL',
  market_data = amd_market_data,
  ordered_price = 11.70,
  quantity = 100,
  date_ordered = datetime(2018,3,9)
)
>>> aapl_portfolio.add_security(amd_stock)
>>> aapl_portfolio.port
{'AAPL Equity': <risk_dash.securities.Equity at 0x1164b2e80>,
 'AMD Equity': <risk_dash.securities.Equity at 0x11791cc88>}
>>> aapl_portfolio.remove_security(amd_stock)
>>> aapl_portfolio.port
{'AAPL Equity': <risk_dash.securities.Equity at 0x1164b2e80>}
>>> aapl_portfolio.remove_security(aapl_stock)
>>> aapl_portfolio.port
{}
```

1.3.3 Calculating Risk Metrics and Using the Portfolio class

Now that we have our `Portfolio` constructed with the securities we have on the book let's use the class to calculate some market risk metrics.

Mark the Portfolio

Let's first mark the current portfolio. Since we want to know the current value of the portfolio, the mark method will calculate the value of the portfolio at the current price for each security. The current price is going to be the last known mark, the price at the closest date to today.

Note: Since the QuandlStockData source hasn't been updated since 3/27/2018, we would expect the last shared date to be 3/27/2018. However, you should use the last shared date as a flag to see if an asset's _MarketData isn't updating. With certain assets, such as Bonds or illiquid securities, marking daily might not make as much sense, so common shared date doesn't mean as much.

```
>>> current_portfolio.mark()
>>> vars(current_portfolio)
{'port': {'AAPL Equity': <risk_dash.securities.Equity at 0x10f8b2940>,
  'AMD Equity': <risk_dash.securities.Equity at 0x1a1f6b0908>,
  'INTC Equity': <risk_dash.securities.Equity at 0x110538d30>,
  'GOOG Equity': <risk_dash.securities.Equity at 0x110548e10>},
 'market_change': -1476.6999999999989,
 'marked_portfolio': {'AAPL Equity': (8999.0, 8417.0),
  'AMD Equity': (1170.0, 1000.0),
  'INTC Equity': (-2609.5, -2559.5),
  'GOOG Equity': (5800.1999999999998, 5025.5)},
 'date_marked': Timestamp('2018-03-27 00:00:00'),
 'initial_value': 13359.700000000001}
```

The `mark` method now creates the `marked_portfolio` dictionary that stores a tuple, (initial_value, market_value), for every security in the portfolio. We also now can calculate a quick holding period return, `holdingreturn = (current_portfolio.initial_value + current_portfolio.market_change)/current_portfolio.initial_value`

```
>>> holdingreturn = (current_portfolio.market_change)/current_portfolio.initial_value
>>> print(holdingreturn)
-0.11053391917483169
```

This hypothetical portfolio apparently hasn't performed over the month since inception, it's lost 11%, but let's look at historic returns before we give up on the portfolio. We can call `portfolio.quick_plot` to look at a `matplotlib` generated cumulative return series of the portfolio. If you wanted more control over plotting, you could use the returned `pandas DataFrame`. In fact, the current implementation is just using the `pandas DataFrame` method `plot()`:

```
>>> marketdata = current_portfolio.quick_plot()
```

Parametrically Calculating the Value at Risk

As we can see, this portfolio is pretty volatile, but has almost doubled over the last four years. Let's calculate what the portfolio daily volatility over the period based off the percent change by calling `get_port_volatility` using `percentchange` from the `market_data`:
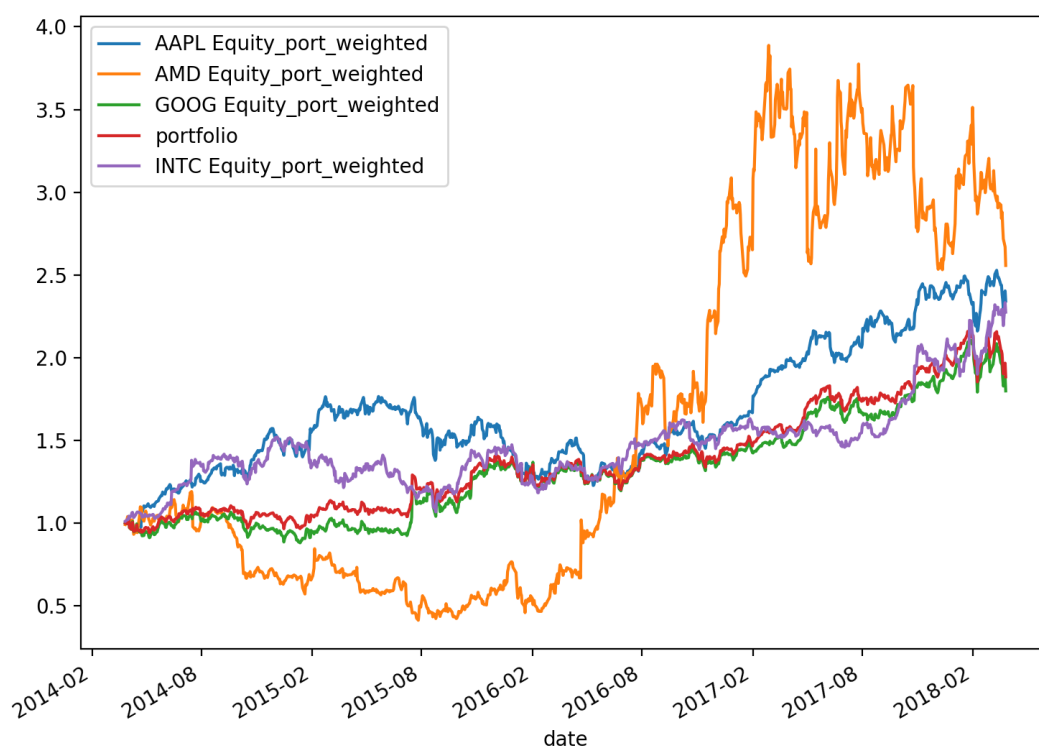
Fig. 1: quick_plot() Output

```
>>> variance, value_at_risk = current_portfolio.set_port_variance(
  key = 'percentchange'
)
>>> volatility = np.sqrt(variance)
>>> print(volatility)
0.01345831069378136
>>> mean = np.mean(current_portfolio.market_data['portfolio'])
>>> print(mean)
0.0007375242310493472
```

We calculated 1.3% daily standard deviation or daily volatility, if the distribution is normally distributed around zero, then we would expect that 95% of the data is contained within approximately 2 standard deviations. We can visually confirm, as well as look to see if there are other distributional aspects we can visually distinguish:

```
>>> import matplotlib.pyplot as plt
>>> marketdata['portfolio'].plot.hist(bins=20,title='Portfolio Historic Returns')
>>> plt.axvline(temp * 1.96, color='r', linestyle='--') # if centered around zero,
→then
>>> plt.axvline(-temp * 1.96, color='r', linestyle='--') #
```
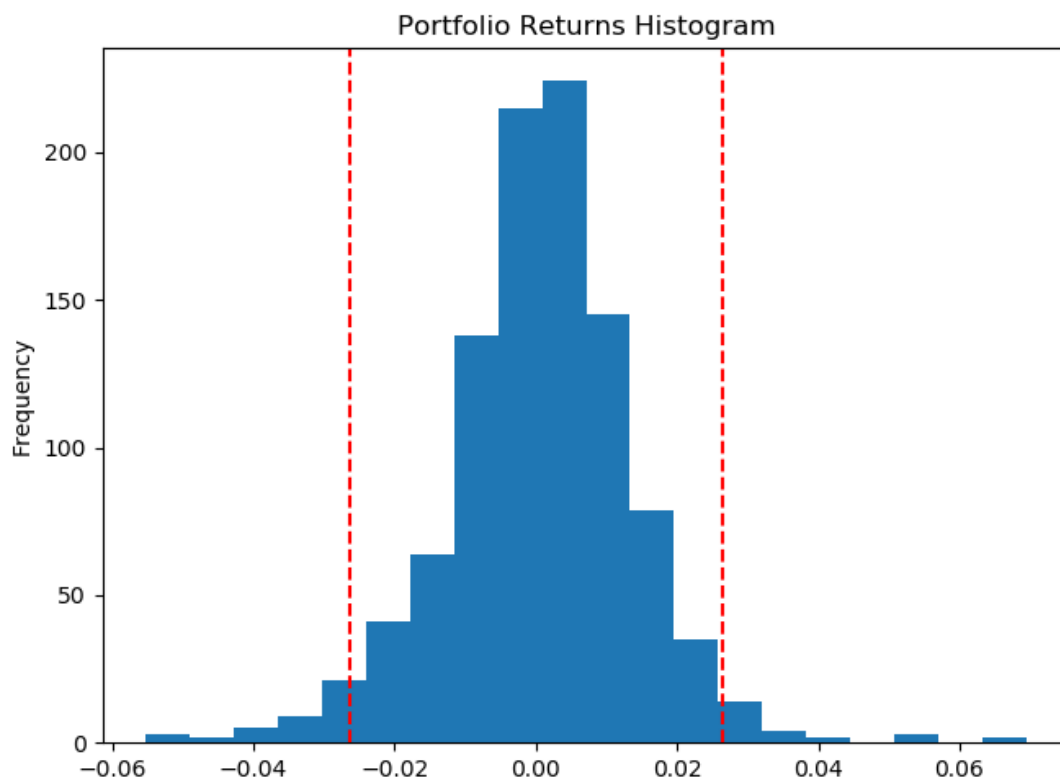

Fig. 2: Portfolio Returns

This distribution looks highly centered around zero, which could signal kurtosis. This seems indicative of equity data, especially for daily returns. Right now, a good place to start thinking about metric parameterization is to assume normality and independence in daily returns. While this assumption might not be very good or might vary between

security to security in the portfolio, which we can account for in simulation or purely using historic returns to calculate risk metrics, we can use this distribution assumption to quickly get a Value at Risk metric over a time horizon.

The default time horizon is 10 days at a 95% confidence level for the `set_port_variance` method, so if we look at the returned `value_at_risk`:

```
>>> print(value_at_risk)
-0.083413941112170473
```

This value is simply the standard deviation scaled by time, at the critical value specified:

$$VaR_{t,T} = \sigma \cdot \sqrt{T - t} \cdot Z^*_{p=\alpha}$$

We can interpret this Value at Risk as being the lower bound of the 95% confidence interval for the 10 day distribution. For this portfolio, a loss over 10 days less than 8.3% should occur 2.5% of the time, on average. To get the dollar value of the 10 Day Value at Risk, we would just multiply this percent change by the current portfolio market value.

```
>>> dollar_value_at_risk = value_at_risk * (current_portfolio.initial_value + current_
↪portfolio.marked_change)
>>> print(dollar_value_at_risk)
-991.20786223592188
```

Similarly, we could interpret as over the a 10 day period, on average, 2.5% of the time there could be an approximate loss over $991.21 dollars for this portfolio. However, this is relying on the assumption that the portfolio is: a) normally distributed, and b) daily returns are serially independent and identically distributed. One way we can go around this is to look at the historic distribution

```
>>> historic_distribution, historic_var = current_portfolio.historic_var()
>>> print(historic_var)
-0.073051970330112487
```

This is calculated by doing a cumulative sum of returns over each horizon time period, then taking the appropriate percentile of the distribution to get a VaR based on historic prices. This is is smaller than the parametric VaR due to the fact that the distribution looks more right skewed as shown below

This method is fairly simple, however it is based on the assumption that the previous distribution of outcomes is a good representation of the future distribution.

Another way we've implemented to calculate the value at risk is to simulate the portfolio distribution.

1.3.4 Simulating the Portfolio

When simulating portfolio returns, one's objective is to correctly specify the portfolio distribution. I consider two major approaches, "bottom-up" and "top-down".

The "bottom-up" approach would include simulating the underlying securities first and then valuing the portfolio through the simulated distributions. The major strength of this method is the ability to easily value the effect of derivative securities on the portfolio. Since one would simulate the derivative's underlier, you could easily then apply the associated value function through the simulated distribution to get the security's profit and loss distribution. Another benefit to this methodology is the analyst has the freedom to change the simulation process at a security level. General Brownian motion might be a good assumption for long/short equity positions, but maybe not as good when simulating yield curves for bonds. Another strength would be the ability to change portfolio weights of securities post simulation, if you simulate a base unit of the security you could then scale the weights accordingly to easily reweigh the portfolio. The biggest challenge to this methodology is to ensure that each simulation value represents the same market environment, meaning that each simulation pull represents the same environment state. While you can potentially do a convolution of the different simulations to get a representative joint distribution of the portfolio, you must ensure that one is capturing the covariance between the securities. For example, equities and bonds have
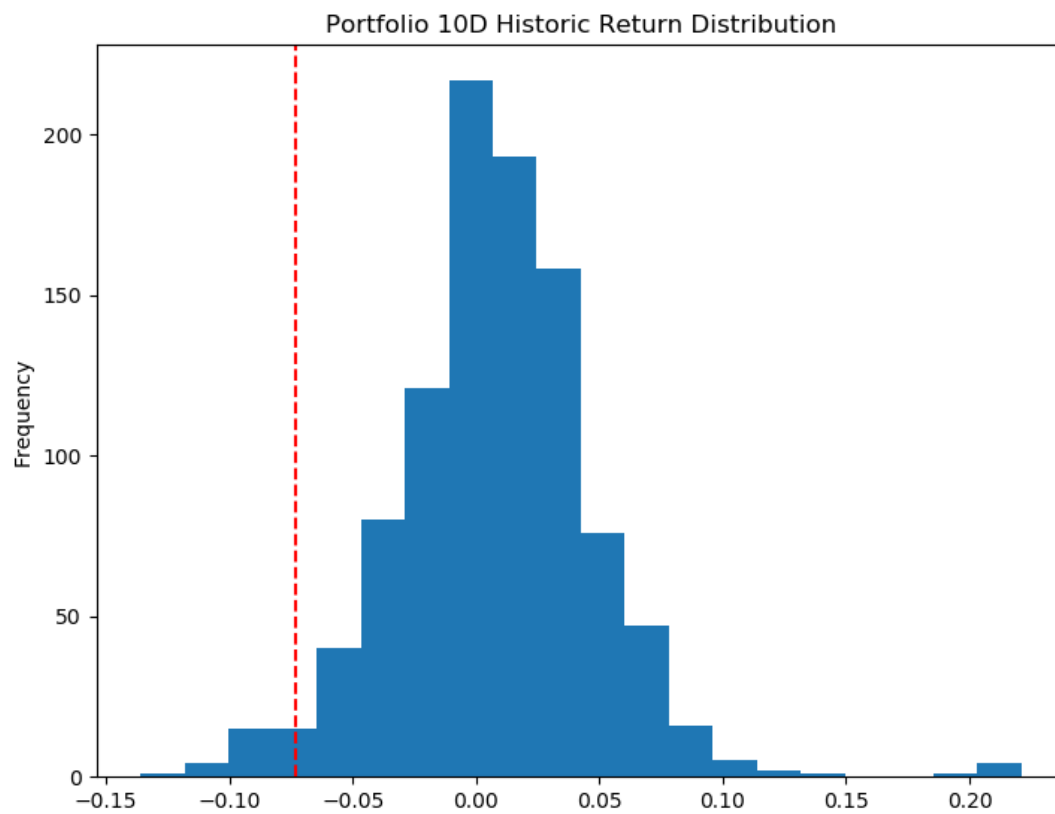
Fig. 3: Historic 10D VaR

historically had negative correlation to each other, thus a portfolio containing both would potentially have a lower variance than each security separate. To capture that in a simulation one would have to simulate directly from the variance-covariance matrix or do a convolution to combine separate simulations together. While both are possible, and in practice it is probably a preferred methodology, however it's not within the scope of this project.

The "top-down" approach would include aggregating the portfolio a priori and then simulating that distribution. Since the portfolio is made of the member securities, thus the aggregated distribution represents all covariance. While this method gets a little be trickier to handle with derivative securities, since you would need historic market prices per contract and potentially roll adjust through the time period, for securities like equities the assumption seems arguable. The benefit of this method would be having to deal with one simulation and verifying if it represents the underlying distribution vs having several different simulations and verifying if they accurately represent the covariance of constituent securities. The drawback is having less flexibility in the modeling of individual securities within the portfolio. Another drawback is to change the weighting or portfolio members, one must recombine and simulate the new portfolio, which could be computationally intensive depending on the methodology.

Either way, to implement simulation, the `_Simulation` and `_RandomGen` class that handle the calculation and generation respectfully. For example, to implement a naive return model, the included `NaiveMonteCarlo` class represents the following generation function for a single observation:

$$R_t = \phi$$

Where $\phi representing a pull from an imposed distribution. As such, we need to specify that imposed distribution, thus we include the$

Since the aim is to specify the portfolio distribution X days into the future, we want to simulate a cumulative return path through time. Under the assumption that each day is independent, the individual simulation path is then:

$$P = \sum_{t=1}^{X} R_t = \sum_{t=1}^{X} \phi$$

Now to fully specify the distribution via a Monte Carlo process, we will generate Y paths to represent the underlying X day distribution. To get the mean of the distribution at each t step from 1 to X:

$$E(R_t) = \frac{1}{Y} \sum_{i=1}^{Y} P_{t,i} + E((R_{t-1}))$$

To get the variance:

$$Var(R_t) = E\left((R_t - E(R_t))^2\right) + Var(R_t) = \frac{1}{Y^2} \sum_{i=1}^{Y} \left(P_{t,i} - \bar{P}_t\right)$$

Since each return is assumed independently and identically distributed, the above condenses to:

$$Var(R_t) = t \cdot Var(R_t)$$

Since we can now switch out the distribution of $\phi to represent the portfolio or constituent securities, this generation function is agno$

Simulating a Unit Resolution Distribution

First let's simulate a unit resolution distribution. By default, the resolution is one day, but depending on the market data resolution you could simulate to match. Since the default is one day, let's simulate a one day distribution and then simulate a X day forward path distribution using `aapl_stock`:

```
>>> from risk_dash.simgen import NormalDistribution, NaiveMonteCarlo
>>> log_return_generator = NormalDistribution(
    location = aapl_stock.market_data.currentexmean,
    scale = aapl_stock.market_data.currentexvol
)
```

Since we're just simulating one day, we can directly use the generator object simulate a one day return distribution. With our new `log_return_generator` instance, we are assuming a normally distributed return series. By default, using `currentexmean` will center the distribution around the closest 80 day exponentially weighted mean of daily AAPL returns. Similarly, using `currentexvol` will set the standard deviation to the closest 80 day exponentially weighted standard deviation of historic daily AAPL returns. To simulate one pull now from a normal distribution, we have an observation that represents a log return of AAPL.

```
>>> log_return_generator.generate(1)
array([-0.00948158])
```

One observation isn't really helpful for us, we now want to simulate an arbitrarily large amount of observations to converge to the underlying distribution. In this case, let's simulate 5000 observations:

```
>>> import numpy as np
>>> one_day_simluation = log_return_generator.generate(5000)
>>> len(one_day_simulation)
5000
>>> np.mean(one_day_simluation)
-0.00036139846164594291
>>> aapl_stock.market_data.currentexmean
-0.00040076765463907944
>>>np.std(one_day_simulation)
0.016497493178538599
>>>aapl_stock.market_data.currentexvol
0.016485817752205818
```

To parameterize the sampling distribution of the distribution, we can simulate an arbitrarily large amount of simulations to converge to the sampling distribution:

```
>>> multiple_one_day_simulations = np.array([log_return_generator.generate(5000) for
→i in 5000])
>>> np.mean(np.mean(multiple_one_day_simulations, axis = 0)) # sampling distribution
→mean of the simulation mean
-0.00039625427660093282
>>> np.std(np.mean(multiple_one_day_simulations, axis=0))
0.00023461080665209953
>>> np.mean(np.std(multiple_one_day_simulations, axis=0))
0.016484835778989758
>>> np.std(np.std(multiple_one_day_simulations, axis=0))
0.00016629650698145025
```

With the mean and standard deviation of the sampling distribution we can construct confidence intervals to see if our calculated mean and variance is contained. This would imply we have specified the imposed distribution based on our calculation of `currentexvol` and `currentexmean`. The calculation for a 95% confidence level is:

$$\bar{X} \pm Z^*_{p=\alpha} \frac{s}{\sqrt{n}}$$

So for this case, for a 95% confidence level, 1-α, $our confidence interval for the simulation mean is (-0.0004027, -0.0003897) and f$

Simulating a Path Distribution

To simulate a forward return path of independent returns, we now want to create a `NaiveMonteCarlo` object to simulate Y forward resolution paths.

```
>>> simulation_generator = NaiveMonteCarlo(log_return_generator)
```

The `NaiveMonteCarlo` accepts any `_RandomGen` object, so we could potentially pass a `_RandomGen` object that might more accurately represent our underlying data. For example, if we thought that AAPL was distributed with a Cauchy distribution to capture fatter tails, we could pass in a `_RandomGen` object that represented the distribution. Now we'll maintain the assumption that the log returns are normally distributed and use the `generator` instance we created earlier.

To simulate 5000 paths for a 5 day forward distribution, we would then call the `simulate` method passing the arguments `periods_forward=5` and `number_of_simulations=5000`. This will set the `simulation_mean`, `simulation_std`, and `simulated_distribution` attributes and return the simulated distribution.

```
>>> path_simulation = simulation_generator.simulate(periods_forward=5, number_of_
→simulations=5000)
>>> path_simulation.shape
(5000,5)
>>> simulation_generator.simulation_mean
array([-0.00050452, -0.00082389, -0.00105195, -0.00135753, -0.0019303 ])
>>> simulation_generator.simulation_std
array([ 0.01630096,  0.02311434,  0.0283451 ,  0.03211978,  0.03607576])
```

The simulation distribution now is 5000 individual 5 day paths, represented as a `numpy` array of shape (5000,5). The `simulation_mean` and `simulation_std` are then calculated across the column axis, giving us the simulated generation through time. Since this method is fairly naive, essentially the cumulative sum of independent random normals, it makes sense that the `simulation_mean` vector is essentially `E(R_{t}) = t \cdot E(R_{t=1})` and `S.D.(R_{t}) = \sqrt{t} * S.D.(R_{t=1})`. If we wanted to implement a more standard approach of simulating returns, we could then create a `_Simulation` class that would represent the value function. To simulate the portfolio from the top down approach, we would then just use the portfolio mean and variance to then simulate the portfolio.

1.3.5 Summary

While this is just the first introduction to the package, there are many expandable directions to go. The aim for the package is to help formalize the development process by providing clear template classes and use cases. The next steps are to write `_Security` classes that match the portfolio that the analyst is trying to model and `_MarketData` classes that match the specific data store for the application.

RISK_DASH.SECURITIES

class risk_dash.securities.**_Security**(*name*, *market_data: risk_dash.market_data._MarketData*, ***kwargs*)

Template for _Security subclasses

class risk_dash.securities.**Equity**(*ticker*, *market_data: risk_dash.market_data.QuandlStockData*, *ordered_price*, *quantity*, *date_ordered*)

class risk_dash.securities.**Portfolio**(*securities=None*, *data_input=None*, *apikey=None*)

The Portfolio class handles interactions with the portfolio data and the associated securities in the portfolio.

> **Parameters**
>
> - **securities** –
>
> - **data_input** –
>
> - **apikey** –

calculate_portfolio_returns(*market_data*)

Calculates market data returns from flat prices

> **Parameters** **market_data** – the market_data variable with a flat price values
>
> **Returns** the market_data variable with log returns

construct_portfolio_csv(*data_input*, *apikey*)

Built in portfolio constructor method

> **Parameters**
>
> - **data_input** – either pandas DataFrame or string path to a portfolio matching './portfolio_example.csv'
>
> - **apikey** – ApiKey for the market data object
>
> **Returns** dict for self.port

drawdown(*market_data=None*, *returns=True*, *key='adj_close'*)

Calculate Draw Down, the previous peak to current trough, through the historic market data

> **Parameters**
>
> - **market_data** – DataFrame, the portfolio level market_data. Use if made external changes to the market_data
>
> - **returns** – Bool, True if calculate DD based on continuous returns, False if on level prices
>
> - **key** – string, corresponds to the market_data column to be calculated
>
> **Returns** DataFrame representing the Draw Down for the portfolio and components

get_date()
    Grab the security with the most dates.

        Returns returns a DatetimeIndex to construct a shared portfolio market_data DataFrame

get_last_shared_date()
    Return the last shared date

        Returns DateTime object

get_port_variance(*recalc=False*, *confidence_interval=0.39881763041638185*, *var_horizon=10*, *lookback_periods=0*, *key=None*)
    Returns portfolio variance

        Parameters

- **recalc** – bool, If True recalculate variance

- **confidence_interval** – float, The critical value to implement parametric VaR

- **var_horizon** – int, how many days/periods forward the parametric VaR should be calculated

- **lookback_periods** – int, how many days/periods backward to condition the underlying distribution

- **key** – string, corresponds to the market_data column to be calculated

        Returns tuple: (float Var(Portfolio), float Parametric Portfolio V@R)

get_portfolio_marketdata(*key=None*, *recalc=False*)
    Returns self.market_data

        Parameters key – If not set, use key to set market_data

        Returns pandas DataFrame self.market_data

get_weights()
    Returns self.weights

        Returns dict for each security weight

historic_var(*market_data=None*, *returns=True*, *key='adj_close'*, *confidence=2.5*, *var_horizon=10*)
    Calulate Value at Risk from the historic distrubtion

        Parameters

- **market_data** – DataFrame, the portfolio level market_data. Use if made external changes to the market_data

- **returns** – Bool, True if calculate DD based on continuous returns, False if on level prices

- **key** – string, corresponds to the market_data column to be calculated

- **confidence** – float, percentile in percentage to pass into np.percentile

- **var_horizon** – int, how many days/periods forward the parametric VaR should be calculated

        Returns tuple, (DataFrame, market_data, float, Historic VaR)

mark()
    Mark portfolio with current market prices, sets marked_portfolio and market_change.

quick_plot (*returns=True*, *key='adj_close'*, *plot=True*, *figsize=(5, 8)*)
Quickly plot using matplotlib and pandas.plot()

> **Parameters**
>
> - **returns** – Bool, True if calculate DD based on continuous returns, False if on level prices
>
> - **key** – string, corresponds to the market_data column to be calculated
>
> - **plot** – Bool, if True, plot the data frame, if flase just return the market_data DataFrame
>
> **Returns** DataFrame market_data

set_port_variance (*market_data=None*, *confidence_interval=-1.9599639845400545*, *var_horizon=10*, *lookback_periods=0*, *key='percentchange'*)
Calulates parametric Variance by calculating Weight.T * Cov * Weight

> **Parameters**
>
> - **market_data** – DataFrame, the portfolio level market_data. Use if made external changes to the market_data
>
> - **confidence_interval** – float, The critical value to implement parametric VaR
>
> - **var_horizon** – int, how many days/periods forward the parametric VaR should be calculated
>
> - **lookback_periods** – int, how many days/periods backward to condition the underlying distribution
>
> - **key** – string, corresponds to the market_data column to be calculated
>
> **Returns** float portfolio variance, standard deviation ** 2, float parametric value at risk

set_portfolio_marketdata (*key*)
Combine individual market data into one pandas DataFrame.

> **Parameters** **key** – Common column name for each security
>
> **Returns** pandas DataFrame containing columns for each security's common market_data

set_weights ()
Calculate value weighted portfolio.

> **Returns** dict for each security weight

value ()
Value current portfolio with current market prices.

> **Returns** value of the portfolio

RISK_DASH.MARKET_DATA

class risk_dash.market_data.**_MarketData**

    current_price()
        This should return the current market price, the price at the last available time period

            Returns float the market price

    gather()
        This should gather data from the source and store it into memory or dictate how to interact with the source

            Returns a pandas DataFrame, market_data or other data type to interact with

class risk_dash.market_data.**QuandlStockData**(*apikey*, *ticker*, *days=80*)

    current_price()
        Returns the latest available market price

            Returns float, latest available market price

    gather()
        Gathers the data from the Quandl api and returns a pandas DataFrame

            Returns pandas DataFrame

    set_expected(*days*)
        Calculate exponentially and simply weighted rolling averages

            Parameters **days** – int, look back days to compute rolling averages

    set_price_changes()
        Set daily price changes and logged percent changes

    set_volatility(*days*)
        Calculate exponentially and simply weighted rolling standard deviations

            Parameters **days** – int, look back days to compute rolling std deviation

FOUR

INDICES AND TABLES

- genindex
- modindex
- search

PYTHON MODULE INDEX

r