project_3_starter

October 27, 2018

1 Project 3: Smart Beta Portfolio and Portfolio Optimization

1.1 Overview

Smart beta has a broad meaning, but we can say in practice that when we use the universe of stocks from an index, and then apply some weighting scheme other than market cap weighting, it can be considered a type of smart beta fund. A Smart Beta portfolio generally gives investors exposure or "beta" to one or more types of market characteristics (or factors) that are believed to predict prices while giving investors a diversified broad exposure to a particular market. Smart Beta portfolios generally target momentum, earnings quality, low volatility, and dividends or some combination. Smart Beta Portfolios are generally rebalanced infrequently and follow relatively simple rules or algorithms that are passively managed. Model changes to these types of funds are also rare requiring prospectus filings with US Security and Exchange Commission in the case of US focused mutual funds or ETFs.. Smart Beta portfolios are generally long-only, they do not short stocks.

In contrast, a purely alpha-focused quantitative fund may use multiple models or algorithms to create a portfolio. The portfolio manager retains discretion in upgrading or changing the types of models and how often to rebalance the portfolio in attempt to maximize performance in comparison to a stock benchmark. Managers may have discretion to short stocks in portfolios.

Imagine you're a portfolio manager, and wish to try out some different portfolio weighting methods.

One way to design portfolio is to look at certain accounting measures (fundamentals) that, based on past trends, indicate stocks that produce better results.

For instance, you may start with a hypothesis that dividend-issuing stocks tend to perform better than stocks that do not. This may not always be true of all companies; for instance, Apple does not issue dividends, but has had good historical performance. The hypothesis about dividend-paying stocks may go something like this:

Companies that regularly issue dividends may also be more prudent in allocating their available cash, and may indicate that they are more conscious of prioritizing shareholder interests. For example, a CEO may decide to reinvest cash into pet projects that produce low returns. Or, the CEO may do some analysis, identify that reinvesting within the company produces lower returns compared to a diversified portfolio, and so decide that shareholders would be better served if they were given the cash (in the form of dividends). So according to this hypothesis, dividends may be both a proxy for how the company is doing (in terms of earnings and cash flow), but also a signal that the company acts in the best interest of its shareholders. Of course, it's important to test whether this works in practice.

You may also have another hypothesis, with which you wish to design a portfolio that can then be made into an ETF. You may find that investors may wish to invest in passive beta funds, but wish to have less risk exposure (less volatility) in their investments. The goal of having a low volatility fund that still produces returns similar to an index may be appealing to investors who have a shorter investment time horizon, and so are more risk averse.

So the objective of your proposed portfolio is to design a portfolio that closely tracks an index, while also minimizing the portfolio variance. Also, if this portfolio can match the returns of the index with less volatility, then it has a higher risk-adjusted return (same return, lower volatility).

Smart Beta ETFs can be designed with both of these two general methods (among others): alternative weighting and minimum volatility ETF.

1.2 Instructions

Each problem consists of a function to implement and instructions on how to implement the function. The parts of the function that need to be implemented are marked with a `# TODO` comment. After implementing the function, run the cell to test it against the unit tests we've provided. For each problem, we provide one or more unit tests from our `project_tests` package. These unit tests won't tell you if your answer is correct, but will warn you of any major errors. Your code will be checked for the correct solution when you submit it to Udacity.

1.3 Packages

When you implement the functions, you'll only need to you use the packages you've used in the classroom, like Pandas and Numpy. These packages will be imported for you. We recommend you don't add any import statements, otherwise the grader might not be able to run your code.

The other packages that we're importing are `helper`, `project_helper`, and `project_tests`. These are custom packages built to help you solve the problems. The `helper` and `project_helper` module contains utility functions and graph functions. The `project_tests` contains the unit tests for all the problems. ### Install Packages

```
In [1]: import sys
        !{sys.executable} -m pip install -r requirements.txt
```

```
Requirement already satisfied: colour==0.1.5 in /opt/conda/lib/python3.6/site-packages (from -r
Collecting cvxpy==1.0.3 (from -r requirements.txt (line 2))
  Downloading https://files.pythonhosted.org/packages/a1/59/2613468ffbbe3a818934d06b81b9f4877fe0
    100% || 880kB 675kB/s ta 0:00:01    33% |                         | 296kB 4.9MB/s eta 0:00:01
Requirement already satisfied: cycler==0.10.0 in /opt/conda/lib/python3.6/site-packages/cycler-0
Collecting numpy==1.13.3 (from -r requirements.txt (line 4))
  Downloading https://files.pythonhosted.org/packages/57/a7/e3e6bd9d595125e1abbe162e323fd2d06f6f
    100% || 17.0MB 38kB/s  eta 0:00:01   10% |                              | 1.8MB 14.2MB/s eta 0
Collecting pandas==0.21.1 (from -r requirements.txt (line 5))
  Downloading https://files.pythonhosted.org/packages/3a/e1/6c514df670b887c77838ab856f57783c07e8
    100% || 26.2MB 24kB/s  eta 0:00:01  2% |                                 | 737kB 23.6MB/s eta
Collecting plotly==2.2.3 (from -r requirements.txt (line 6))
  Downloading https://files.pythonhosted.org/packages/99/a6/8214b6564bf4ace9bec8a26e7f89832792be
    100% || 1.1MB 606kB/s eta 0:00:01  6% |                                | 71kB 11.9MB/s eta 0:00
Requirement already satisfied: pyparsing==2.2.0 in /opt/conda/lib/python3.6/site-packages (from
```

```
Requirement already satisfied: python-dateutil==2.6.1 in /opt/conda/lib/python3.6/site-packages
Requirement already satisfied: pytz==2017.3 in /opt/conda/lib/python3.6/site-packages (from -r r
Requirement already satisfied: requests==2.18.4 in /opt/conda/lib/python3.6/site-packages (from
Collecting scipy==1.0.0 (from -r requirements.txt (line 11))
  Downloading https://files.pythonhosted.org/packages/d8/5e/caa01ba7be11600b6a9d39265440d7b3be3d
    100% || 50.0MB 12kB/s  eta 0:00:01   10% |                              | 5.3MB 21.3MB/s eta 0
Requirement already satisfied: scikit-learn==0.19.1 in /opt/conda/lib/python3.6/site-packages (f
Requirement already satisfied: six==1.11.0 in /opt/conda/lib/python3.6/site-packages (from -r re
Collecting tqdm==4.19.5 (from -r requirements.txt (line 14))
  Downloading https://files.pythonhosted.org/packages/71/3c/341b4fa23cb3abc335207dba057c790f3bb3
    99% || 51kB 9.7MB/s eta 0:00:011    100% || 61kB 4.5MB/s
Collecting osqp (from cvxpy==1.0.3->-r requirements.txt (line 2))
  Downloading https://files.pythonhosted.org/packages/43/f2/bbeb83c0da6fd89a6d835b98d85ec76c04f3
    100% || 153kB 3.4MB/s eta 0:00:01
Collecting ecos>=2 (from cvxpy==1.0.3->-r requirements.txt (line 2))
  Downloading https://files.pythonhosted.org/packages/b6/b4/988b15513b13e8ea2eac65e97d84221ac515
    100% || 122kB 4.6MB/s eta 0:00:01
Collecting scs>=1.1.3 (from cvxpy==1.0.3->-r requirements.txt (line 2))
  Downloading https://files.pythonhosted.org/packages/b3/fd/6e01c4f4a69fcc6c3db130ba55572089e78e
    100% || 143kB 3.2MB/s eta 0:00:01
Collecting multiprocess (from cvxpy==1.0.3->-r requirements.txt (line 2))
  Downloading https://files.pythonhosted.org/packages/7a/ee/b9bf3e171f936743758ef924622d8dd00516
    100% || 1.4MB 452kB/s eta 0:00:01    36% |                    | 501kB 22.0MB/s eta 0:00:01
Requirement already satisfied: fastcache in /opt/conda/lib/python3.6/site-packages (from cvxpy==
Requirement already satisfied: toolz in /opt/conda/lib/python3.6/site-packages (from cvxpy==1.0.
Requirement already satisfied: decorator>=4.0.6 in /opt/conda/lib/python3.6/site-packages (from
Requirement already satisfied: nbformat>=4.2 in /opt/conda/lib/python3.6/site-packages (from plo
Requirement already satisfied: chardet<3.1.0,>=3.0.2 in /opt/conda/lib/python3.6/site-packages (
Requirement already satisfied: idna<2.7,>=2.5 in /opt/conda/lib/python3.6/site-packages (from re
Requirement already satisfied: urllib3<1.23,>=1.21.1 in /opt/conda/lib/python3.6/site-packages (
Requirement already satisfied: certifi>=2017.4.17 in /opt/conda/lib/python3.6/site-packages (fro
Requirement already satisfied: future in /opt/conda/lib/python3.6/site-packages (from osqp->cvxp
Collecting dill>=0.2.8.1 (from multiprocess->cvxpy==1.0.3->-r requirements.txt (line 2))
  Downloading https://files.pythonhosted.org/packages/6f/78/8b96476f4ae426db71c6e86a8e6a81407f01
    100% || 153kB 3.2MB/s eta 0:00:01
Requirement already satisfied: ipython-genutils in /opt/conda/lib/python3.6/site-packages (from
Requirement already satisfied: traitlets>=4.1 in /opt/conda/lib/python3.6/site-packages (from nb
Requirement already satisfied: jsonschema!=2.5.0,>=2.4 in /opt/conda/lib/python3.6/site-packages
Requirement already satisfied: jupyter-core in /opt/conda/lib/python3.6/site-packages (from nbfo
Building wheels for collected packages: cvxpy, plotly, ecos, scs, multiprocess, dill
  Running setup.py bdist_wheel for cvxpy ... done
  Stored in directory: /root/.cache/pip/wheels/2b/60/0b/0c2596528665e21d698d6f84a3406c52044c7b4c
  Running setup.py bdist_wheel for plotly ... done
  Stored in directory: /root/.cache/pip/wheels/98/54/81/dd92d5b0858fac680cd7bdb8800eb26c001dd9f5
  Running setup.py bdist_wheel for ecos ... done
  Stored in directory: /root/.cache/pip/wheels/50/91/1b/568de3c087b3399b03d130e71b1fd048ec072c45
  Running setup.py bdist_wheel for scs ... done
  Stored in directory: /root/.cache/pip/wheels/ff/f0/aa/530ccd478d7d9900b4e9ef5bc5a39e895ce110be
```

```
  Running setup.py bdist_wheel for multiprocess ... done
  Stored in directory: /root/.cache/pip/wheels/8b/36/e5/96614ab62baf927e9bc06889ea794a8e87552b84
  Running setup.py bdist_wheel for dill ... done
  Stored in directory: /root/.cache/pip/wheels/e2/5d/17/f87cb7751896ac629b435a8696f83ee75b11029f
Successfully built cvxpy plotly ecos scs multiprocess dill
Installing collected packages: numpy, scipy, osqp, ecos, scs, dill, multiprocess, cvxpy, pandas,
  Found existing installation: numpy 1.12.1
    Uninstalling numpy-1.12.1:
      Successfully uninstalled numpy-1.12.1
  Found existing installation: scipy 0.19.1
    Uninstalling scipy-0.19.1:
      Successfully uninstalled scipy-0.19.1
  Found existing installation: dill 0.2.7.1
    Uninstalling dill-0.2.7.1:
      Successfully uninstalled dill-0.2.7.1
  Found existing installation: pandas 0.20.3
    Uninstalling pandas-0.20.3:
      Successfully uninstalled pandas-0.20.3
  Found existing installation: plotly 2.0.15
    Uninstalling plotly-2.0.15:
      Successfully uninstalled plotly-2.0.15
  Found existing installation: tqdm 4.11.2
    Uninstalling tqdm-4.11.2:
      Successfully uninstalled tqdm-4.11.2
Successfully installed cvxpy-1.0.3 dill-0.2.8.2 ecos-2.0.5 multiprocess-0.70.6.1 numpy-1.13.3 os
You are using pip version 9.0.1, however version 18.1 is available.You should consider upgrading
```

1.3.1 Load Packages

```
In [2]: import pandas as pd
        import numpy as np
        import helper
        import project_helper
        import project_tests
```

1.4 Market Data

1.4.1 Load Data

For this universe of stocks, we'll be selecting large dollar volume stocks. We're using this universe, since it is highly liquid.

```
In [3]: df = pd.read_csv('../../data/project_3/eod-quotemedia.csv')

        percent_top_dollar = 0.2
        high_volume_symbols = project_helper.large_dollar_volume_stocks(df, 'adj_close', 'adj_vo
        df = df[df['ticker'].isin(high_volume_symbols)]
```

```
close = df.reset_index().pivot(index='date', columns='ticker', values='adj_close')
volume = df.reset_index().pivot(index='date', columns='ticker', values='adj_volume')
dividends = df.reset_index().pivot(index='date', columns='ticker', values='dividends')
```

1.4.2 View Data

To see what one of these 2-d matrices looks like, let's take a look at the closing prices matrix.

```
In [4]: project_helper.print_dataframe(close)
```

2 Part 1: Smart Beta Portfolio

In Part 1 of this project, you'll build a portfolio using dividend yield to choose the portfolio weights. A portfolio such as this could be incorporated into a smart beta ETF. You'll compare this portfolio to a market cap weighted index to see how well it performs.

Note that in practice, you'll probably get the index weights from a data vendor (such as companies that create indices, like MSCI, FTSE, Standard and Poor's), but for this exercise we will simulate a market cap weighted index.

2.1 Index Weights

The index we'll be using is based on large dollar volume stocks. Implement `generate_dollar_volume_weights` to generate the weights for this index. For each date, generate the weights based on dollar volume traded for that date. For example, assume the following is close prices and volume data:

```
                  Prices
               A          B          ...
2013-07-08     2          2          ...
2013-07-09     5          6          ...
2013-07-10     1          2          ...
2013-07-11     6          5          ...
...            ...        ...        ...

                  Volume
               A          B          ...
2013-07-08     100        340        ...
2013-07-09     240        220        ...
2013-07-10     120        500        ...
2013-07-11     10         100        ...
...            ...        ...        ...
```

The weights created from the function `generate_dollar_volume_weights` should be the following:

```
               A          B          ...
2013-07-08     0.126..    0.194..    ...
2013-07-09     0.759..    0.377..    ...
```

```
2013-07-10     0.075..   0.285..   ...
2013-07-11     0.037..   0.142..   ...
...            ...       ...       ...
```

```
In [5]: def generate_dollar_volume_weights(close, volume):
            """
            Generate dollar volume weights.

            Parameters
            ----------
            close : DataFrame
                Close price for each ticker and date
            volume : str
                Volume for each ticker and date

            Returns
            -------
            dollar_volume_weights : DataFrame
                The dollar volume weights for each ticker and date
            """
            assert close.index.equals(volume.index)
            assert close.columns.equals(volume.columns)

            #TODO: Implement function
            df_weights = close*volume

            # I think the example given is misleading.
            # The way it is illustrated, it looks like you look at each stock
            # and weight that individual stock to the observed dates.
            #
            # What I think they want is for each date, what is the weight of
            # the stocks.
            #
            # .sum(axis=1) means add up along the row (the date) and not the stock
            # the column.
            #
            # .div(axis='index') means
            # Divide each row of a DataFrame by another DataFrame vector
            # https://stackoverflow.com/questions/22642162/python-divide-each-row-of-a-dataframe
            dollar_volume_weights = df_weights.div( df_weights.sum(axis=1),
                                                    axis='index')



    #     # DEBUG
    #     #
    #     print("DEBUG - close")
    #     print(close.head())
```

```
#     print("DEBUG - volume")
#     print(volume.head())
#     print("DEBUG - df_weights")
#     print(df_weights.head())
#     print("DEBUG - df_weights.sum()")
#     print(df_weights.sum())
#     print("DEBUG - dollar_volume_weights")
#     print(dollar_volume_weights)

    return dollar_volume_weights

project_tests.test_generate_dollar_volume_weights(generate_dollar_volume_weights)
```

```
Tests Passed
```

2.1.1 View Data

Let's generate the index weights using `generate_dollar_volume_weights` and view them using a heatmap.

```
In [6]: index_weights = generate_dollar_volume_weights(close, volume)
        project_helper.plot_weights(index_weights, 'Index Weights')
```

```
<IPython.core.display.HTML object>
```

2.2 Portfolio Weights

Now that we have the index weights, let's choose the portfolio weights based on dividend. You would normally calculate the weights based on trailing dividend yield, but we'll simplify this by just calculating the total dividend yield over time.

Implement `calculate_dividend_weights` to return the weights for each stock based on its total dividend yield over time. This is similar to generating the weight for the index, but it's using dividend data instead. For example, assume the following is `dividends` data:

```
                 Prices
               A          B
2013-07-08     0          0
2013-07-09     0          1
2013-07-10     0.5        0
2013-07-11     0          0
2013-07-12     2          0
...            ...        ...
```

The weights created from the function `calculate_dividend_weights` should be the following:

```
               A          B
2013-07-08     NaN        NaN
```

```
2013-07-09     0         1
2013-07-10     0.333..   0.666..
2013-07-11     0.333..   0.666..
2013-07-12     0.714..   0.285..
...            ...       ...
```

```
In [7]: def calculate_dividend_weights(dividends):
            """
            Calculate dividend weights.

            Parameters
            ----------
            dividends : DataFrame
                Dividend for each stock and date

            Returns
            -------
            dividend_weights : DataFrame
                Weights for each stock and date
            """
            #TODO: Implement function

            # DEBUG
            #
            #print(dividends)

            # How I believe this works is that for each stock, you do the cumulative sum or the
            # Then for that date, you check the weight of the return for all the stock.
            df_cumlative_sum = dividends.cumsum()

            # This is similar to the last problem cell.
            #
            # .sum(axis=1) means add up along the row (the date) and not the stock
            # the column.
            #
            # .div(axis='index') means
            # Divide each row of a DataFrame by another DataFrame vector
            # https://stackoverflow.com/questions/22642162/python-divide-each-row-of-a-dataframe
            dividend_weights = df_cumlative_sum.div( df_cumlative_sum.sum(axis=1),
                                                     axis='index')

            return dividend_weights

        project_tests.test_calculate_dividend_weights(calculate_dividend_weights)
```

```
Tests Passed
```

2.2.1 View Data

Just like the index weights, let's generate the ETF weights and view them using a heatmap.

```
In [8]: etf_weights = calculate_dividend_weights(dividends)
        project_helper.plot_weights(etf_weights, 'ETF Weights')
```

```
<IPython.core.display.HTML object>
```

2.3 Returns

Implement `generate_returns` to generate returns data for all the stocks and dates from price data. You might notice we're implementing returns and not log returns. Since we're not dealing with volatility, we don't have to use log returns.

```
In [9]: def generate_returns(prices):
            """
            Generate returns for ticker and date.

            Parameters
            ----------
            prices : DataFrame
                Price for each ticker and date

            Returns
            -------
            returns : Dataframe
                The returns for each ticker and date
            """
            #TODO: Implement function

            # returns (not log returns) is (today's price minus yesterday's price) divided by ye
            # NOTE: .shift(1) goes back one day.
            #       https://pandas.pydata.org/pandas-docs/stable/generated/pandas.DataFrame.shif
            returns = (prices - prices.shift(1))/prices.shift(1)

            return returns

        project_tests.test_generate_returns(generate_returns)
```

```
Tests Passed
```

2.3.1 View Data

Let's generate the closing returns using `generate_returns` and view them using a heatmap.

```
In [10]: returns = generate_returns(close)
         project_helper.plot_returns(returns, 'Close Returns')
```

```
<IPython.core.display.HTML object>
```

2.4 Weighted Returns

With the returns of each stock computed, we can use it to compute the returns for an index or ETF. Implement `generate_weighted_returns` to create weighted returns using the returns and weights.

```
In [11]: def generate_weighted_returns(returns, weights):
             """
             Generate weighted returns.

             Parameters
             ----------
             returns : DataFrame
                 Returns for each ticker and date
             weights : DataFrame
                 Weights for each ticker and date

             Returns
             -------
             weighted_returns : DataFrame
                 Weighted returns for each ticker and date
             """
             assert returns.index.equals(weights.index)
             assert returns.columns.equals(weights.columns)

         #      # DEBUG
         #      #
         #      print("returns")
         #      print(returns)
         #      print("weights")
         #      print(weights)

             #TODO: Implement function
             weighted_returns = returns.mul(weights)

             return weighted_returns

         project_tests.test_generate_weighted_returns(generate_weighted_returns)
```

```
Tests Passed
```

2.4.1 View Data

Let's generate the ETF and index returns using `generate_weighted_returns` and view them using a heatmap.

```
In [12]: index_weighted_returns = generate_weighted_returns(returns, index_weights)
         etf_weighted_returns = generate_weighted_returns(returns, etf_weights)
         project_helper.plot_returns(index_weighted_returns, 'Index Returns')
         project_helper.plot_returns(etf_weighted_returns, 'ETF Returns')
```

```
<IPython.core.display.HTML object>
```

```
<IPython.core.display.HTML object>
```

2.5 Cumulative Returns

To compare performance between the ETF and Index, we're going to calculate the tracking error. Before we do that, we first need to calculate the index and ETF comulative returns. Implement `calculate_cumulative_returns` to calculate the cumulative returns over time given the returns.

```
In [13]: def calculate_cumulative_returns(returns):
             """
             Calculate cumulative returns.

             Parameters
             ----------
             returns : DataFrame
                 Returns for each ticker and date

             Returns
             -------
             cumulative_returns : Pandas Series
                 Cumulative returns for each date
             """
             #TODO: Implement function


         #     # DEBUG
         #     #
         #     print('returns')
         #     print(returns)

             df_return_of_stocks_per_day = returns.sum(axis=1)

             # The cumulative return formula
             # Cumulative return is the cumprod of (1+r_t) where r_t is the daily return.
             #
             # ex: cumulative = (1 + return1) * (1 + return2) * (1 + return3) - 1
             # https://stackoverflow.com/questions/40811246/pandas-cumulative-return-function
             df_temp_a = 1.0 + df_return_of_stocks_per_day
             cumulative_returns = df_temp_a.cumprod()
```

```
#      # DEBUG
#      #
#      print('returns')
#      print(returns)
#      print('cumulative_returns')
#      print(cumulative_returns)

    return cumulative_returns

project_tests.test_calculate_cumulative_returns(calculate_cumulative_returns)
```

```
Tests Passed
```

2.5.1 View Data

Let's generate the ETF and index cumulative returns using `calculate_cumulative_returns` and compare the two.

```
In [14]: index_weighted_cumulative_returns = calculate_cumulative_returns(index_weighted_returns
         etf_weighted_cumulative_returns = calculate_cumulative_returns(etf_weighted_returns)
         project_helper.plot_benchmark_returns(index_weighted_cumulative_returns, etf_weighted_c
```

2.6 Tracking Error

In order to check the performance of the smart beta portfolio, we can calculate the annualized tracking error against the index. Implement `tracking_error` to return the tracking error between the ETF and benchmark.

For reference, we'll be using the following annualized tracking error function:

$$TE = \sqrt{252} * SampleStdev(r_p - r_b)$$

Where $ r_p $ is the portfolio/ETF returns and $ r_b $ is the benchmark returns.

Note: When calculating the sample standard deviation, the delta degrees of freedom is 1, which is the also the default value.

```
In [15]: def tracking_error(benchmark_returns_by_date, etf_returns_by_date):
             """
             Calculate the tracking error.

             Parameters
             ----------
             benchmark_returns_by_date : Pandas Series
                 The benchmark returns for each date
             etf_returns_by_date : Pandas Series
                 The ETF returns for each date

             Returns
             -------
```

```
        tracking_error : float
            The tracking error
        """
        assert benchmark_returns_by_date.index.equals(etf_returns_by_date.index)

        #TODO: Implement function
        ds_delta = benchmark_returns_by_date - etf_returns_by_date
        tracking_error = np.sqrt(252) * ds_delta.std()

        return tracking_error

    project_tests.test_tracking_error(tracking_error)
```

```
Tests Passed
```

2.6.1 View Data

Let's generate the tracking error using `tracking_error`.

```
In [16]: smart_beta_tracking_error = tracking_error(np.sum(index_weighted_returns, 1), np.sum(et
         print('Smart Beta Tracking Error: {}'.format(smart_beta_tracking_error))
```

```
Smart Beta Tracking Error: 0.1020761483200753
```

3 Part 2: Portfolio Optimization

Now, let's create a second portfolio. We'll still reuse the market cap weighted index, but this will be independent of the dividend-weighted portfolio that we created in part 1.

We want to both minimize the portfolio variance and also want to closely track a market cap weighted index. In other words, we're trying to minimize the distance between the weights of our portfolio and the weights of the index.

$Minimize \left[\sigma^2_p + \lambda\sqrt{\sum_1^m(weight_i - indexWeight_i)^2}\right]$ where m is the number of stocks in the portfolio, and λ is a scaling factor that you can choose.

Why are we doing this? One way that investors evaluate a fund is by how well it tracks its index. The fund is still expected to deviate from the index within a certain range in order to improve fund performance. A way for a fund to track the performance of its benchmark is by keeping its asset weights similar to the weights of the index. We'd expect that if the fund has the same stocks as the benchmark, and also the same weights for each stock as the benchmark, the fund would yield about the same returns as the benchmark. By minimizing a linear combination of both the portfolio risk and distance between portfolio and benchmark weights, we attempt to balance the desire to minimize portfolio variance with the goal of tracking the index.

3.1 Covariance

Implement `get_covariance_returns` to calculate the covariance of the `returns`. We'll use this to calculate the portfolio variance.

If we have m stock series, the covariance matrix is an $m \times m$ matrix containing the covariance between each pair of stocks. We can use `Numpy.cov` to get the covariance. We give it a 2D array in which each row is a stock series, and each column is an observation at the same period of time. For any `NaN` values, you can replace them with zeros using the `DataFrame.fillna` function.

The covariance matrix $\mathbf{P} = \begin{bmatrix} \sigma^2_{1,1} & \dots & \sigma^2_{1,m} \\ \dots & \dots & \dots \\ \sigma_{m,1} & \dots & \sigma^2_{m,m} \end{bmatrix}$

```
In [17]: def get_covariance_returns(returns):
             """
             Calculate covariance matrices.

             Parameters
             ----------
             returns : DataFrame
                 Returns for each ticker and date

             Returns
             -------
             returns_covariance  : 2 dimensional Ndarray
                 The covariance of the returns
             """
             #TODO: Implement function

             # NOTE: rowvar
             #       If rowvar is True (default), then each row represents a variable,
             #       with observations in the columns. Otherwise, the relationship is
             #       transposed: each column represents a variable, while the rows
             #       contain observations.
             #       https://docs.scipy.org/doc/numpy/reference/generated/numpy.cov.html
             returns_covariance = np.cov( returns.fillna( value=0),
                                          rowvar=False )

             return returns_covariance

         project_tests.test_get_covariance_returns(get_covariance_returns)
```

```
Tests Passed
```

3.1.1 View Data

Let's look at the covariance generated from `get_covariance_returns`.

```
In [18]: covariance_returns = get_covariance_returns(returns)
         covariance_returns = pd.DataFrame(covariance_returns, returns.columns, returns.columns)

         covariance_returns_correlation = np.linalg.inv(np.diag(np.sqrt(np.diag(covariance_retur
         covariance_returns_correlation = pd.DataFrame(
```

```
        covariance_returns_correlation.dot(covariance_returns).dot(covariance_returns_corre
        covariance_returns.index,
        covariance_returns.columns)

    project_helper.plot_covariance_returns_correlation(
        covariance_returns_correlation,
        'Covariance Returns Correlation Matrix')
```

```
<IPython.core.display.HTML object>
```

3.1.2 portfolio variance

We can write the portfolio variance $\sigma^2_p = \mathbf{x^T} \mathbf{P} \mathbf{x}$

Recall that the $\mathbf{x^T} \mathbf{P} \mathbf{x}$ is called the quadratic form. We can use the cvxpy function `quad_form(x,P)` to get the quadratic form.

3.1.3 Distance from index weights

We want portfolio weights that track the index closely. So we want to minimize the distance between them. Recall from the Pythagorean theorem that you can get the distance between two points in an x,y plane by adding the square of the x and y distances and taking the square root. Extending this to any number of dimensions is called the L2 norm. So: $\sqrt{\sum_{1}^{n}(weight_i - indexWeight_i)^2}$ Can also be written as $\left\| \mathbf{x} - \mathbf{index} \right\|_2$. There's a cvxpy function called norm() `norm(x, p=2, axis=None)`. The default is already set to find an L2 norm, so you would pass in one argument, which is the difference between your portfolio weights and the index weights.

3.1.4 objective function

We want to minimize both the portfolio variance and the distance of the portfolio weights from the index weights. We also want to choose a `scale` constant, which is λ in the expression.

$\mathbf{x^T} \mathbf{P} \mathbf{x} + \lambda \left\| \mathbf{x} - \mathbf{index} \right\|_2$

This lets us choose how much priority we give to minimizing the difference from the index, relative to minimizing the variance of the portfolio. If you choose a higher value for `scale` (λ).

We can find the objective function using cvxpy `objective = cvx.Minimize()`. Can you guess what to pass into this function?

3.1.5 constraints

We can also define our constraints in a list. For example, you'd want the weights to sum to one. So $\sum_{1}^{n} x = 1$. You may also need to go long only, which means no shorting, so no negative weights. So $x_i >0 $ for all i. you could save a variable as `[x >= 0, sum(x) == 1]`, where x was created using `cvx.Variable()`.

3.1.6 optimization

So now that we have our objective function and constraints, we can solve for the values of $\mathbf{x}$. cvxpy has the constructor `Problem(objective, constraints)`, which returns a `Problem` object.

The Problem object has a function solve(), which returns the minimum of the solution. In this case, this is the minimum variance of the portfolio.

It also updates the vector **x**.

We can check out the values of x_A and x_B that gave the minimum portfolio variance by using x.value

```
In [19]: import cvxpy as cvx

         def get_optimal_weights(covariance_returns, index_weights, scale=2.0):
             """
             Find the optimal weights.

             Parameters
             ----------
             covariance_returns : 2 dimensional Ndarray
                 The covariance of the returns
             index_weights : Pandas Series
                 Index weights for all tickers at a period in time
             scale : int
                 The penalty factor for weights the deviate from the index
             Returns
             -------
             x : 1 dimensional Ndarray
                 The solution for x
             """
             assert len(covariance_returns.shape) == 2
             assert len(index_weights.shape) == 1
             assert covariance_returns.shape[0] == covariance_returns.shape[1]  == index_weights

             #TODO: Implement function

             # Based off Lesson 18 - Porfolio Optimization
             #            9. Exercise: cvxpy advanced optimization.
             m = covariance_returns.shape[0]
             x = cvx.Variable(m)
             portfolio_variance = cvx.quad_form(x, covariance_returns)
             distance_to_index = cvx.norm( x - index_weights )
             objective = cvx.Minimize(portfolio_variance + scale*distance_to_index)
             constraints = [x >= 0, sum(x) == 1 ]
             cvx.Problem( objective,
                          constraints ).solve()

             # NOTE: Notice the period for x.value
             #        .value is a python property
             x_values = x.value
             return x_values

         project_tests.test_get_optimal_weights(get_optimal_weights)
```

```
---------------------------------------------------------------------------

RuntimeError                              Traceback (most recent call last)

RuntimeError: module compiled against API version 0xc but this version of numpy is 0xb
```

```
Tests Passed
```

3.2 Optimized Portfolio

Using the `get_optimal_weights` function, let's generate the optimal ETF weights without rebalanceing. We can do this by feeding in the covariance of the entire history of data. We also need to feed in a set of index weights. We'll go with the average weights of the index over time.

```
In [20]: raw_optimal_single_rebalance_etf_weights = get_optimal_weights(covariance_returns.value
         optimal_single_rebalance_etf_weights = pd.DataFrame(
             np.tile(raw_optimal_single_rebalance_etf_weights, (len(returns.index), 1)),
             returns.index,
             returns.columns)
```

With our ETF weights built, let's compare it to the index. Run the next cell to calculate the ETF returns and compare it to the index returns.

```
In [21]: optim_etf_returns = generate_weighted_returns(returns, optimal_single_rebalance_etf_wei
         optim_etf_cumulative_returns = calculate_cumulative_returns(optim_etf_returns)
         project_helper.plot_benchmark_returns(index_weighted_cumulative_returns, optim_etf_cumu

         optim_etf_tracking_error = tracking_error(np.sum(index_weighted_returns, 1), np.sum(opt
         print('Optimized ETF Tracking Error: {}'.format(optim_etf_tracking_error))
```

```
Optimized ETF Tracking Error: 0.05795012630412267
```

3.3 Rebalance Portfolio Over Time

The single optimized ETF portfolio used the same weights for the entire history. This might not be the optimal weights for the entire period. Let's rebalance the portfolio over the same period instead of using the same weights. Implement `rebalance_portfolio` to rebalance a portfolio.

Reblance the portfolio every n number of days, which is given as `shift_size`. When rebalancing, you should look back a certain number of days of data in the past, denoted as `chunk_size`. Using this data, compute the optoimal weights using `get_optimal_weights` and `get_covariance_returns`.

```
In [22]: def rebalance_portfolio(returns, index_weights, shift_size, chunk_size):
             """
             Get weights for each rebalancing of the portfolio.
```

```
    Parameters
    ----------
    returns : DataFrame
        Returns for each ticker and date
    index_weights : DataFrame
        Index weight for each ticker and date
    shift_size : int
        The number of days between each rebalance
    chunk_size : int
        The number of days to look in the past for rebalancing

    Returns
    -------
    all_rebalance_weights  : list of Ndarrays
        The ETF weights for each point they are rebalanced
    """
    assert returns.index.equals(index_weights.index)
    assert returns.columns.equals(index_weights.columns)
    assert shift_size > 0
    assert chunk_size >= 0

    #TODO: Implement function

#     # DEBUG
#     #
#     print('returns')
#     print(returns)
#     print('index_weights')
#     print(index_weights)
#     print('shift_size')
#     print(shift_size)
#     print('chunk_size')
#     print(chunk_size)
#     print('index_weights.shape')
#     print(index_weights.shape)


    # Michael W suggested steps:
    # Rebalance every n days (shift_size)
    # When rebalancing look back however many days (chunk size) to find calculate the r
    # Note, don't start our loop with an index lower than chunk_size
    # Pass these into get_covariance_returns & get_optimal_weights
    # Append values to an array (all_rebalance_weights)
    # return all_rebalance_weights
    #
    # The solution pseudo-code
    # 1. For each chunk_size in the returns, first calculate covariance with get_covari
```