Snakes on a Plane

Shrinking the Sample Covariance Matrix: A Pythonic Approach

Jason Strimpel
Illinois Institute of Technology

For Dr. B. Boonstra
Independent Study, MSF 597

Introduction

When estimating the sample covariance matrix of stock returns as the input to a mean-variance optimization problem and the number of stocks under consideration is large relative to the number of historical return observations, the most extreme values in the sample covariance matrix have a lot of error. The mean-variance optimization routines will place large bets on the extreme value which is unreliable because these extreme values are largely error.

Ledoit and Wolf show that following the methods described in their paper, "Honey, I Shrunk the Sample Covariance Matrix" (http://papers.ssrn.com/sol3/papers.cfm?abstract_id=433840), tracking error is reduced relative to a benchmark index and the information ratio is increased. This has obvious benefits in the market for active equity portfolio managers. While the authors implement their method in MATLAB, this experiment follows the empirical study outlined in the paper in Python which has become a compelling choice for work in quantitative finance, trading, and statistical applications.

Environment Used

Below describes the environment (including the Python modules) used for the experiment.

- Mac OS X 10.6 (http://store.apple.com/us/product/MC573Z/A?fnode=MTY1NDAzOA)
- Python 2.7 (http://www.python.org)
- Numpy 1.6.0 (http://numpy.scipy.org/)
- SciPy 0.9.0 (http://www.scipy.org/)
- Pandas 0.4.0dev (http://code.google.com/p/pandas/)
- PyTables 2.2.1 (http://www.pytables.org/moin)
- matplotlib 1.0.1 (http://matplotlib.sourceforge.net/)
- CVXOPT 1.1.3 (http://abel.ee.ucla.edu/cvxopt/)

Usage

In the `covshrink/core` directory:

```
In [1]: import eval
In [2]: eval.run(runs=10, index=[15, 30, 50, 75, 100])
```

Custom Modules, Classes, and Functions Used

Three custom built class were used in this experiment in addition to several custom modules and functions. They are outlined here.

Class `Portfolio` is initialized with the following arguments:

- `portfolio` which is a dictionary containing at least the following key-value pairs:
    - `expected_returns` which contains a dictionary of expected returns for the stocks in the portfolio
    - `holding_periods` which contains the start and end dates for which to analyze the stocks in the portfolio
    - `shares` which contains the number of shares held in the portfolio
    - `constraints` which contains the constraints for the optimizer
    - `defaults` which contains defaults such as frequency, global start date, and global end date
- `start` which is either a string or datetime object representing the beginning of the time frame in which to analyze the portfolio. This can be used as a global override
- `end` which is either a string or datetime object representing the end of the time frame in which to analyze the portfolio
- `proxy` is a dictionary for those behind a proxy (such as me)

`Portfolio` has the following methods:

- `Portfolio._get_historic_data(ticker, start, end)` returns a `pandas.DataFrame` containing the historic data for `ticker` from `start` to `end`
- `Portfolio._get_historic_returns(ticker, start, end, offset=1)` returns a `pandas.Series` with `offset`-period return data for `ticker` from `start` to `end`
- `Portfolio.get_portfolio_historic_returns()` returns a `pandas.DataFrame` containing the historic returns of the portfolio with the default parameters
- `Portfolio.get_portfolio_historic_position_values(shares=None)` returns a `pandas.DataFrame` containing the values of the portfolio constituents
- `Portfolio.get_portfolio_historic_values(shares=None)` returns a `pandas.DataFrame` containing the historic portfolio values
- `Portfolio.get_benchmark_weights()` returns a `pandas.DataFrame` containing the weights of the benchmark constituents
- `Portfolio.get_benchmark_returns()` returns a `pandas.dataFrame` with returns of the benchmark constituents
- `Portfolio.get_active_weights()` returns a `pandas.DataFrame` with active weights for each portfolio component

- `Portfolio.get_portfolio_weights(shares=None)` returns a `pandas.DataFrame` containing the portfolio weights of the portfolio
- `Portfolio.get_expected_stock_returns()` returns a `pandas.DataFrame` which contains the expected stock returns on the portfolio constituents as defined in the default parameters
- `Portfolio.get_active_returns()` returns a `pandas.DataFrame` which contains the active stock returns on the portfolio constituents
- `Portfolio.get_expected_excess_stock_returns()` returns a `pandas.DataFrame` which contains the expected excess stock returns on the portfolio constituents as defined by the authors
- `Portfolio.get_covariance_matrix(historic_returns)` returns an NxN `pandas.DataFrame` covariance matrix
- `Portfolio.get_shrunk_covariance_matrix(x, shrink=None)` takes a sample covariance matrix and returns a tuple: `pandas.DataFrame` which contains the shrunk covariance matrix; and a float which is the shrinkage intensity factor
- `Portfolio.get_expected_benchmark_return()` returns a `pandas.DataFrame` which contains the expected returns on the benchmark constituents
- `Portfolio.get_expected_portfolio_return()` returns `pandas.DataFrame` which contains the expected returns on the portfolio constituents
- `Portfolio.get_portfolio_size()` returns an integer representing the number of stocks in the portfolio
- `Portfolio.get_trading_dates()` returns a list of datetime objects corresponding to the dates for which there are historic returns
- `Portfolio.information_ratio(historic_returns)` returns a float representing the information ratio as defined by Grinold and Kahn

Class `Yahoo` is initialized with the following arguments:

- `ticker_list` which is a list of ticker symbols
- `proxy` which is a dictionary containing a valid proxy argument for `urllib2`

`Yahoo` has many methods, of which one is used:

- `Yahoo.get_LastTradePriceOnly(symbol)` which returns Yahoo's last traded price (20 minute or 15 minute delayed) for `symbol`

Class `InsertPriceHist` is initialized with the following arguments:

- `proxy` which is a dictionary containing a valid proxy argument for `urllib2`

`InsertPriceHist` has the following methods:

- `InsertPriceHist._fetch_historical_yahoo()` which is a version of the matplotlib method modified to accept a proxy and a user-defined period (daily, weekly, monthly, yearly). This method returns a file handle with stock data suitable for iterating
- `InsertPriceHist.insert()` returns a boolean after successfully inserting the stock data returned by `_fetch_historical_yahoo()` into a PyTables h5f flatfile

The following modules are also used:

- `createdailytable.reset_table()` which creates a PyTables h5f flatfile and returns an open file handle
- `optimize.optimize(a, S)` which accepts a sample covariance matrix and stock return data and returns a `cvxopt.matrix` of optimized portfolio weights
- `params.get_portfolio_params(index=30, start=None, end=None)` returns user-defined data on index number of stocks. start and end override the default start and end dates for the data
- `eval.eval(type, index=30)` which actually runs the experiment. The type argument is either the string 'sample' or 'shrunk' which dictates which covariance matrix should be used in the optimizer to get the optimal portfolio weights. The argument index represents the number of stocks in the portfolio and benchmark
- `eval.run(runs=10, index=[15, 30, 50, 75, 100])` runs `eval()` to execute the experiment and print the results to the screen

Test Suite

Ledoit and Wolf published MATLAB code that generates the shrunken covariance matrix as well as the shrinkage intensity factor (http://www.ledoit.net/covCor.m) along with the paper, the results of which are used as the test suite to which my results (computed in Python) are compared.

I computed a covariance matrix of monthly stock returns between 2/1990 and 12/2005 for 30 stocks in Python and re-formatted the results to build a 30x30 element matrix in MATLAB. I passed the covariance matrix to the shrinkage method and returned a shrinkage intensity factor of 0.1852. The shrinkage intensity factor returned from the Python implementation was 0.1852, accurate to at least four decimal places.

To test the accuracy of the shrunk matrix, I computed the covariance matrix in Python for the return series described above and re-formatted the results to build a 30x30 element matrix in MATLAB. I passed the covariance matrix to the shrinkage method and returned the shrunk matrix to a MATLAB variable `sigma`. I then computed the shrunk matrix in Python and re-formatted the results to build a 30x30 element matrix in MATLAB and named the variable `python_sigma`. I then ran the following

command in MATLAB to compute the elementwise sum of squared errors between the Python implementation and the authors' MATLAB implementation:

```
>> sse = sum(sum((python_sigma-sigma).^2))

sse =

  2.3336e-015
```

I found this result acceptable.

Range of Inputs and Process

Five faux-indexes of 15, 30, 50, 75, and 100 stocks were used to mimic value weighted indexes. Each period the benchmark weights were recalculated as a value-weighted portfolio meaning the benchmark weight of each constituent stock equals the that period's closing price divided by the sum of all closing prices of all stocks in the portfolio for that period.

Each stock was assigned an expected return of 3% (see Further Investigation). This of course is unrealistic but is functional for the experiment.

The authors build the expected excess returns, α, by adding random noise to the realized excess returns. I did this by using a one-period lognormal model of returns ($\mu+\sigma*\varepsilon(0,1)$) where ε is a NxM matrix of normally distributed variables with mean 0 and standard deviation 1, N is the number of stocks (30), and *M* is the number of periods. μ was assumed 0.03 and σ was assumed 0.05 (see Further Investigation). The authors then build α in such a way that the unconstrained annualized ex-ante information ratio (IR) is 1.5. This procedure is described in Appendix C of "Honey, I Shrunk the Sample Covariance Matrix". Because the realized excess returns were generated by a random process, each test was run 10 times with each metric averaged. (I attempted to conduct the experiment with 50 runs and got memory errors.)

Monthly adjusted closing prices were used between 2/1990 and 1/2001.

I roughly followed the authors' empirical study. The steps are outlined here:

- Compute the active returns for a rolling 60-period window and use those returns to compute the sample covariance matrix and the shrunken covariance matrix
- Compute the realized returns for that time period
- Feed the expected excess returns, α, and either the sample covariance matrix or the shrunken covariance matrix (depending on the experiment) into the quadratic optimizer (see Further Investigation) which computes the optimal weights of each portfolio position which minimizes variance of the portfolio
- Compute the optimized portfolio return by multiplying the optimized weights by the current monthly realized return

- Because the expected excess returns are generated by normally distributed random variables, I run the experiment 10 times and compute the mean of the information ratio, mean expected return, mean standard deviation, and mean tracking error

Analysis and Comparison of Results

In contrast to the authors' results, at each index size (15, 30, 50, 75, 100), the information ratio (as computed in the paper) was less for the portfolio with component weights optimized with the shrunk covariance matrix than the portfolio with component weights optimized with the sample covariance matrix (see Appendix A).

The mean return is generally higher for the portfolio with component weights optimized with the shrunk covariance matrix than the portfolio with component weights optimized with the sample covariance matrix however the standard deviation of these returns is greater as well. This explains why the information ratio is less for the shrunk-weighted portfolio: lower risk adjusted returns. One point I found interesting is that the tracking error (standard deviation of excess – or active – returns) is less for the shrunk-weighted portfolio at all index levels by a consistent four five to six points which is consistent with the authors' results.

In further contrast to the authors' results, as the index size increases, the information ratio actually increases as well (see Appendix C). I believe this is an artifact of the lack of constraints added to the optimizer.

Further Investigation

The main purpose of this project was to implement a relevant experiment in Python using many of the modules available for quantitative financial researchers. The secondary purpose was to build a framework with which one has the ability to build from and enhance as time goes on. Given the main point was to show that using a "shrunk" covariance matrix reduces tracking error relative to a benchmark index and increases the information ratio, some details were left out and (strong) assumptions left in.

- Index sizes. I had to rely on freely available price and index data which was a major hamper in my ability to fully analyze the strength of using the shrunk covariance matrix when the number of stocks is high compared to the number of data points. Not only do index providers (Standard and Poor's, Nikkei, etc.) not provide index components, it is increasingly difficult to find an index that has had the same components during the date ranges of the experiment. I built an arbitrary list of 127 symbols and allow the user to select an index size up to 127 components to analyze how the shrunk covariance matrix affects the mean statistics computed by the authors.
- Expected return model. There are many models available to forecast expected returns for use in a mean-variance optimized portfolio. This experiment assumes these expected returns are given. However, the program was built with the ability to incorporate a forecasting model in the future.

- Optimizer. A Python module CVXOPT was used as the optimizer in the problem mainly because the authors of the module provide a workable example of using the quadratic optimizer to solve for the optimal active weights in the mean-variance minimization problem. I certainly left out some of the constraints the authors include due to ignorance using the code. I spent a considerable amount of time attempting to properly build the optimization problem to no avail. A certain improvement in the future is to include all constraints provided by the authors. (See Appendix B.)
- Using CAPM for estimating μ and σ. In building the excess expected returns on stocks, I assumed a μ = 0.03 and σ = 0.05. It would be easy to use a model such as CAPM to better estimate these parameters.

Conclusion

I plan to continue modifying the project to enhance its usability and relevance. One of the major issues I faced was data. For example, around 50 of the symbols used have a data point on 2/1/2001 and around 50 of the symbols have a data point on 2/4/2001. Further, some symbols I attempted to use didn't even have a data point for February 2002!

As Python continues to mature as a logical choice for quantitative finance, more libraries and modules are sure to follow. For example, there are wrapper APIs for Bloomberg which allow Python users to programmatically access data through Bloomberg's COM interface. This would help avoid many of the issues I faced attempting to use free methods to access data (through Yahoo's web interface). There are also Python wrapper APIs for accessing macro economic data through FRED (St. Louis Fed's data portal).

I attempted to build the code generally as a framework that would allow enhancements as described in the Further Investigation section above. For example, a simple factor model could be written and imported to generate expected excess returns. In fact, a whole library of factor models could be written and imported. One could iterate through these models during an in-sample backtest and apply the model with the most favorable outcome.

Appendix A

Results of experiment with shrunk covariance matrix. N is the index size, Runs is the number of runs used to calculate the mean of the statistics.

	IR	Mean	SD	TE
Sample	**0.8098**	**0.0153**	**0.0653**	**0.1148**
Shrink	**0.4512**	**0.0148**	**0.1160**	**0.0636**

computed in 57.9 seconds
N=15 Runs=10

	IR	Mean	SD	TE
Sample	**1.2418**	**0.0165**	**0.0462**	**0.1120**
Shrink	**0.3865**	**0.0118**	**0.1124**	**0.0453**

computed in 109.48 seconds
N=30 Runs=10

	IR	Mean	SD	TE
Sample	**1.3064**	**0.0191**	**0.0508**	**0.1070**
Shrink	**0.7210**	**0.0227**	**0.1074**	**0.0502**

computed in 185.03 seconds
N=50 Runs=10

	IR	Mean	SD	TE
Sample	**1.4108**	**0.0208**	**0.0509**	**0.1074**
Shrink	**0.8307**	**0.0254**	**0.1076**	**0.0505**

computed in 292.53 seconds
N=75 Runs=10

	IR	Mean	SD	TE
Sample	**1.6109**	**0.0229**	**0.0492**	**0.1089**
Shrink	**0.7195**	**0.0215**	**0.1091**	**0.0489**

computed in 437.2 seconds
N=100 Runs=10

total run 18.04 minutes

Appendix B

Constraints used in the mean-variance optimization problem v. those the authors used.

Constraint	Description	Included
$\mathbf{Min}(x'\Sigma x)$	Minimize the tracking error variance	Yes
$x'\alpha >= g$	Meet or exceed manager's target gain	No
$1'x = 1$	Portfolio weights add to unity (fully invested)	Yes
$x >= -w_B$	Portfolio is long only	No
$x <= c1 - W_B$	Total position in a stock cannot exceed c% of the portfolio	No

Appendix C

The information ratio increases as the size of the index increases.