Equity Hedging

Farhang Tarlan
Farhang.tarlan@gmail.com | 416-709-3161 | LinkedIn: /farhangtarlan

The purpose of this assignment is to find a basket of securities, from a universe of securities, to hedge a target security with. Security prices are tracked over a year from November 2014 to November 2015. In this project, I devise a deep learning-based time series forecasting algorithm to learn and predict security prices. I will then use these predictions to construct a basket of securities by locally optimizing an objective function.

1. Introduction

In this project, I will devise an algorithm to build a basket of security that works as a proxy for a target security, known as hedging. Securities in the hedging basket are often highly liquid; the hedging basket is constructed in a way to ensure average market performance.

This document explains some of the design decisions in the project and elaborates on the math behind them.

2. Data exploration

Data imputation. I begin by noticing that the dataset misses some security prices. To impute the missing data points, I linearly interpolated the data point preceding and following the missing value. This assumes that security prices behave linearly in short time scales.

Price trends. To visualize security prices over time, **Fig. 1** depicts the price of three securities over the tracking period. Notice that securities may exhibit a wide range of patterns – some are more volatile than others. A good model needs to be able to predict various types of patterns. The function `plot_security(…)` plots the trends of any number of securities.

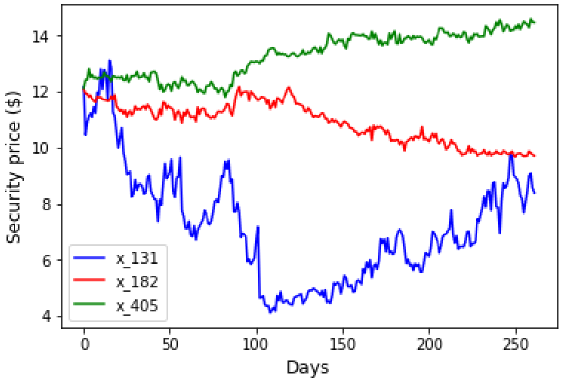

Fig. 1 *| prices of three securities tracked over time. X-axis indicates the number of days since start of the tracking. Y-axis is the price of the security. Notice that the security plotted in blue exhibits a relatively volatile trend while the red and green securities are less volatile.*

Data distribution. An important check in machine learning pipelines is to observe the distribution of the data.

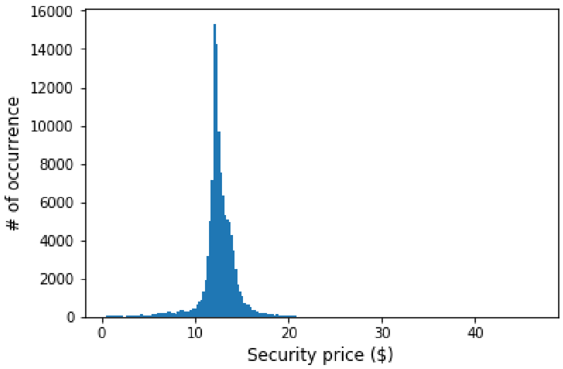

Fig. 2 *| security price distribution. The graph above combines the prices of all 444 securities.*

Observe that the data is normally distributed with a mean of 12.60 and standard deviation of 2.26.

Data normalization. Machine learning, especially deep learning, models benefit from a normalized input dataset (mean 0, standard deviation 1). A normalized input data speeds up the learning process and often yields better performance. Intuitively, this is because the gradient of the objective function is often very close to zero when the input is away from zero. Hence at each optimization iteration, the weights of the model take very small steps towards convergence, and even sometime diverge with large learning rates.

I scale the dataset, so that all the values will lie in range (0, 1). This range is somewhat empirical. I have tried other scaling schemes, but this worked best. The function `scale(…)` scales the dataset to a given range using the following formulas:

$$s = \frac{upper - lower}{D_{max} - D_{min}} \quad (1)$$

$$D_s = s * (D - D_{min}) + D_{min} \quad (2)$$

Where *upper* and *lower* are the upper and lower bounds to scale the data to (1 and 0 in our case, respectively), and D_{max} and D_{min} are the maximum and minimum values of the data, respectively. Each security is scaled independently of other securities. The function `inverse_scale(…)` reverses the above scaling, and will be used to inverse scale the data back to its original scale. These formulas are taken from the scikit-learn preprocessing MinMaxScaler module. The two scaling functions are locally implemented to support some additional vector operations but are functionally the same as the one in sciki-learn preprocessing library.

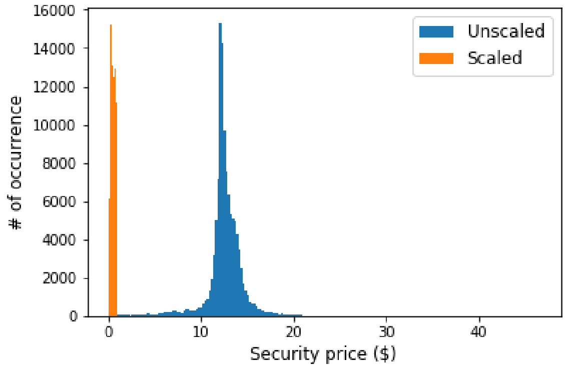

Fig. 3 | *distribution of the data before (blue) and after (orange) scaling.*

Do not let the narrower orange plot deceive you that there are fewer elements post-scaling. The narrower orange distribution is because of the finer bin sizes when plotting the histogram – both graphs consist of equal number of data points.

Train-validation split. I now split the data into training and validation sets. To build each of these sets, I need to construct a (X, y) mapping. To train the model, I use T_x security prices just prior to each data point. Hence T_x will be a hyperparameter of the model. For now, we set T_x = 10, equaling to two business weeks worth of data.

Also, I allocate 80% of the data to training and 20% to validation. The model will be trained on the training set, and its hyperparameters will be tuned using the validation set.

We now have four arrays – (X, y) pairings for training and validation – as below

- `x_train`: of shape (m_{train}, T_x, $n_{features}$)

- `y_train`: of shape (m_{train}, n_{labels})
- `x_valid`: of shape $(m_{valid}, T_x, n_{features})$
- `y_valid`: of shape (m_{valid}, n_{labels})

With $T_x = 10$, and m = 262 data points, we have a training set of $m_{train} = 199$ training examples, each having $T_x = 10$ time steps. At each time step, the input is a $n_{features} = 446$ dimensional vector (444 security prices + 2 categorical variables). Similarly, $n_{valid} = 43$.

3. Model development

I use a Recurrent Neural Network (RNN), where there is a Long-Short Term Memory (LSTM) cell at each time step. LSTMs are particularly good at maintaining long-range dependencies and are hence appropriate for such an application. A schematic of the model is depicted in **Fig. 4**.

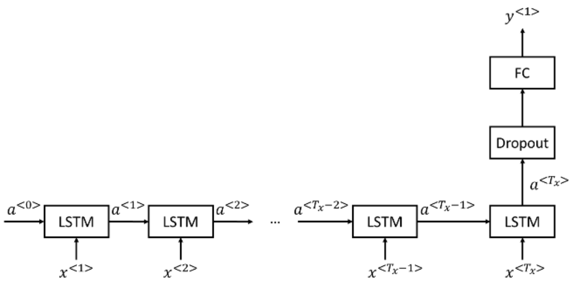

Fig. 4 | *schematic of the architecture of the RNN model, where we have a LSTM unit at every time step. There sis a dropout (to mitigate overfitting) and fully connected layer (to solve the regression problem) at the last time step.*

The model is designed to be only one layer deep. This is because of the limited number of data points available. Larger models run the chance of overfitting the training set and poorly generalizing to an unseen test set. `rnn_model(…)` implements the above architecture in Keras with Tensorflow backend. `compile_run_model(…)` compiles and runs the model. Adam optimizer is used with the learning rate as one of the hyperparameters (the other three constants in Adam optimizer are taken as default and not tuned in the hyperparameter optimization process). Mean squared error is the cost function to optimize.

Training the model with the batch size of 16, learning rate of 0.01, and dropout rate of 0.2 for 100 epochs yields the following graph.

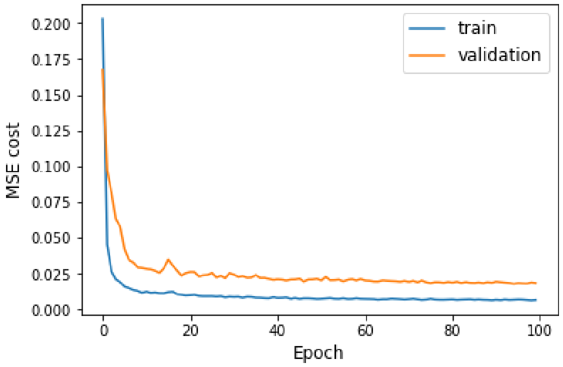

Fig. 5 | *mean squared loss (MSE) plotted over the training epochs. The blue curve is the MSE loss on the training set. The orange curve shows the MSE loss on the validation set.*

Notice that the training error is smaller than the validation error – as expected. Both error measures decrease as a function of training epochs – as expected.

We now tune the hyperparameters used in the model. Only the ones deemed to be more important to the performance of the model will be tuned. Below is a list of hyperparameters tuned:

- Dropout
- Epoch
- Learning rate
- Batch size

Looking at the root mean squared (RMS) cost on the training (0.065) and validation sets (0.135), it looks like the model is more suffering from high variance than bias. This conclusion is implicitly based on determination of the Bayes error (the lowest possible error any model can reach). The difference between the training set cost and the Bayes error would be an indication of bias (<0.065); the difference between the training set cost and validation set cost would be an indication of the variance (>0.07).

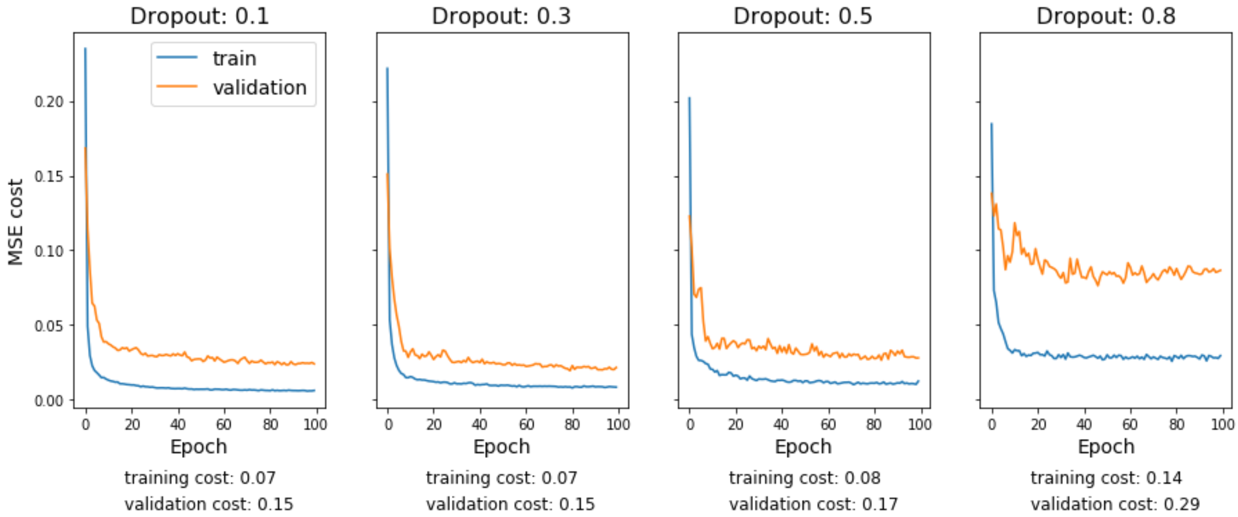

Fig. 6 | *performance of four models in terms of MSE cost over 100 training epochs on the training (blue) and validation set (orange). The RMS cost of each model on the training and validation sets is written below the corresponding graph.*

To reduce variance, I focus on regularization techniques, mainly Dropout. Function `optimize_dropout(…)` compiles and runs the model with different values of dropout. The MSE loss as a function of epochs is plotted in **Fig. 6** above. The original dropout rate of 0.2 performs best on the validation set and has the lowest cost.

Next, I focus on the number of epochs to train the model on. It was originally set to 100. Training epochs need to be tuned to obtain the most optimized model weights. However, training the model for too long causes the model to overfit. Hence, we need to stop the training process at the point of lowest cost on the validation set, known as early stopping.

Notice that the validation cost decreases as we train the model for more epochs up to ~200 epochs, at which point the validation cost starts to increase. We further ensure this observation is indeed the case by averaging out the validation costs of the last 10 epochs to smooth out small fluctuations in the graph. Function `optimize_epoch(…)` performs what just described to build and run the models with different training epochs.

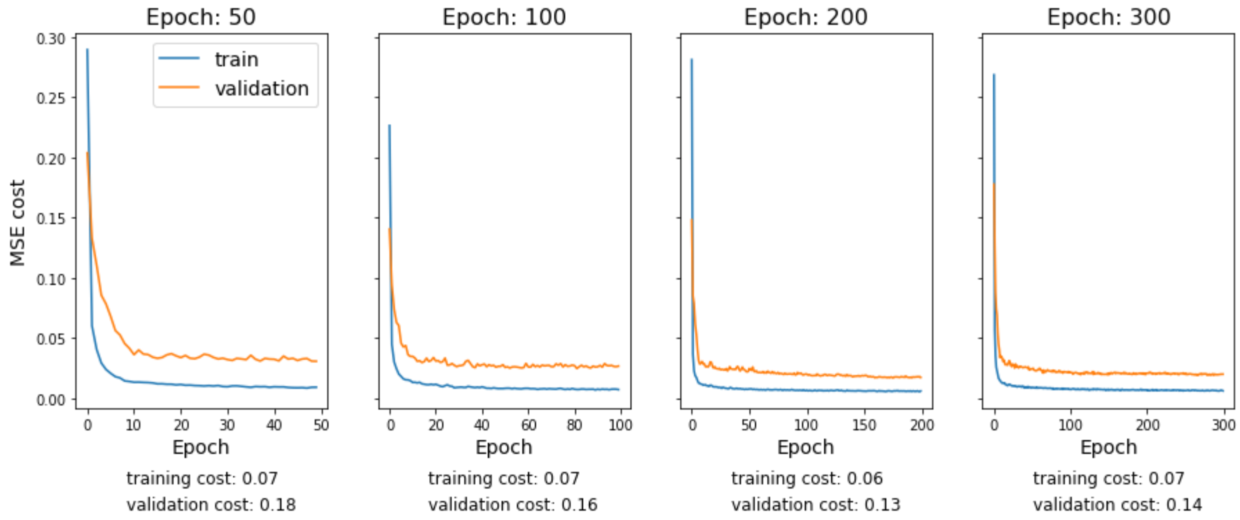

Fig. 7 | *performance of four models with different training epochs on the training (blue) and validation (orange) sets. The RMS loss of each model on the training and validation sets is written below the corresponding graph. Notice that the model trained for 200 epochs performs best on the validation set.*

We now optimize the learning rate. The initial learning rate of 0.01 was purely based on my experience running similar models. To get a better understanding of the order of magnitude of an appropriate learning rate, I quickly ran the four models, each with a different learning rate, for 50 epochs (data not shown). This would narrow down my search to a finer region.

This quick test gave me the intuition to try learning rates from 10^{-3} to 10^{-5}. Function `optimize_lr(…)` builds and runs each of these models. The cost of each of these models over the training epochs is plotted in **Fig. 8** below.

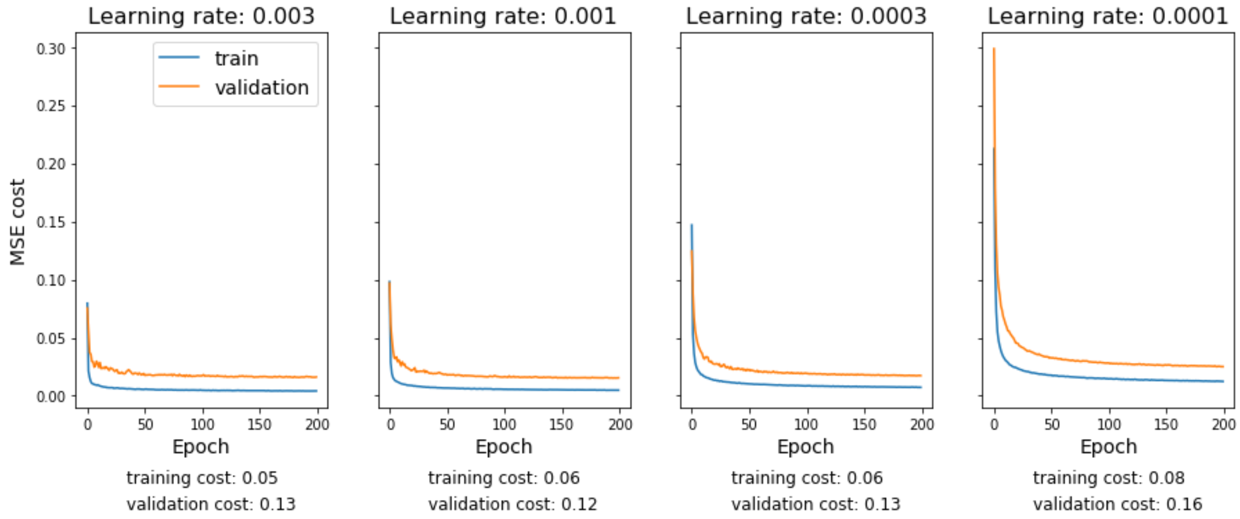

Fig. 8 | *performance of four models with different learning rates over 200 training epochs on the training (blue) and validation (orange) sets. The RMS cost of each model on both sets is written below the corresponding graphs.*

Notice that the model with learning rate of 0.001 has the best performance on the validation set. Too large learning rates may cause the model to diverge. Too small learning rates take a long time to optimize model parameters. 0.001 seems to be the happy medium.

We now optimize the batch size. Batch size is the number of training examples the model takes in before a single model parameter update. Often, machine learning models perform just as well by not taking in the entire training set, given they are trained on sufficient epochs. The function `optimize_batch_size(…)` builds and runs models each with a different batch size. We then plot their performance on the training and validation sets in the same manner we optimized other hyperparameters.

Batch sizes were chosen to be multiples of two. This is due to the computer architecture and the memory layout in a computer. Often operations of matrices that are not multiples of two are slower as different registers in the memory need to used.

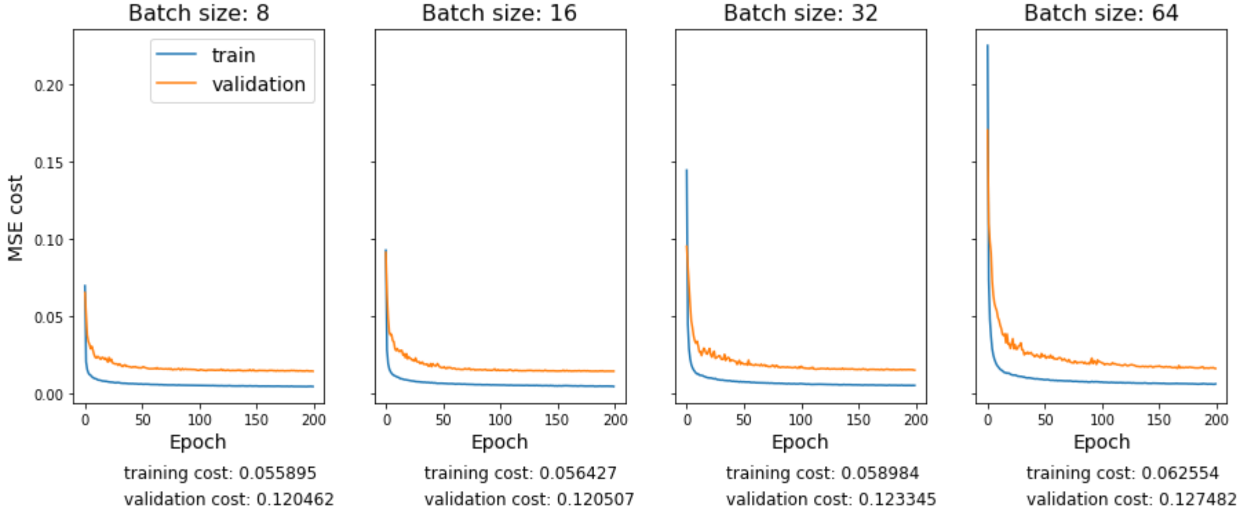

Fig. 9 | *performance of four models with different batch sizes on the training (blue) and validation (orange) sets. The RMS cost of each model after 200 epochs is written below the corresponding graph.*

There are other hyperparameters to the model we could optimize. Most notably would be T_x that defines the number of time steps in the model. Increasing T_x would provide the model with more information per data point to train the model on. However, as T_x increases, the size of the training and validation sets decrease. This hyperparameter is not optimized.

Using these hyperparameters, we plot the predictions of the model on the training and validation sets and compare them to true security values. Note that we need to inverse scale model predicitons back to the original scale of the input data. This is done by `inverse_scale(…)`.

The function `plot_predictions(…)` plots the prediction of a model along with true security prices for any given security. **Fig. 10** below depicts the model predictions on security x_280 (chosen at random).

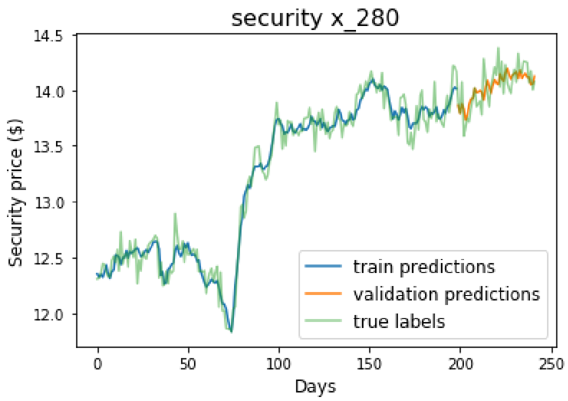

Fig. 10 | *predictions of the model on the training (blue) and validation (orange) sets. True security prices are plotted in green.*

4. Hedging basket proposal

The RNN model allows us to forecast security behaviour into the future. In finding the basket of securities, we have two subproblems to solve:

1. Finding X_i: securities in the basket
2. Finding W_i: weight of each security

This section proposes an algorithm to solve for X_i. In section 5 on page 8, I will solve the other subproblem of finding corresponding weights for each security.

I define a metric, deviation, to quantitatively evaluate the goodness of a potential security X_i to hedge the target security y as below

$$\Delta_i = X_i - y \tag{3}$$

$$\pi = \frac{1}{k}\sum_{i=1}^{k}\sigma(\Delta_i) \tag{4}$$

Where *Δ* is the difference between the target security and the securities in the hedging basket, *σ(Δ)* is the standard deviation of *Δ*, and *k* is the number of securities in the basket. In other words, the deviation of a basket (π) from a target security is the average of the standard deviation of the difference between the securities in the basket and the target security.

Intuitively, the smaller the difference between a security and the target security, the smaller the deviation. In the extreme case, a basket only consisting of the target security itself has the deviation of 0. The function `basket_deviation(…)` computes the deviation of a given basket for a given target security. The smaller the deviation, the better the basket.

To find the basket of securities, I propose the following algorithm – *selective inclusion* – inspired by [1].

```
1. $B = \emptyset$
2. For i = 1 to $k$
     i.  Find a new security with smallest deviation from the target
     ii. Include the security to $B$ if it has not been already added
```

Where *B* denotes the basket of securities. Selective inclusion works by finding a local minimum to our objective function (Equation 4). Functions `tryin(…)` and `include(…)` perform these operations.

Assuming equal weights for the securities in the basket, **Fig. 11** plots the performance of the basket selected by selective inclusion. The starting and end of the planning horizon were set to the beginning and end of the recording period, respectively.

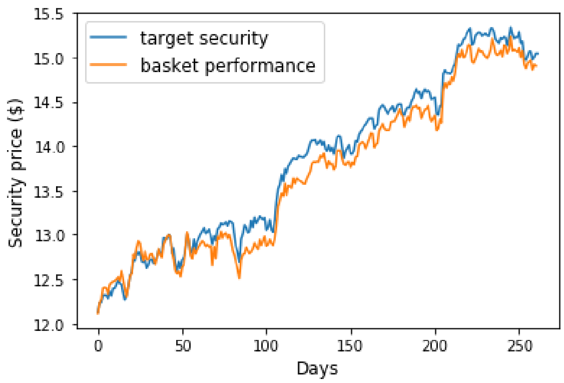

Fig. 11 | *the performance of the basket selected through selective inclusion as a proxy to target security y_1 assuming equal weightings.*

Selective inclusion works by adding one security at a time to get the greatest reduction in the objective function. One alternative to the above algorithm is to start with a basket containing all the securities from the universe of securities and removing one security at a time – *selective exclusion*. A pseudocode for selective exclusion would look like

```
1. $B = \{X_1, \dots, X_m\}$
2. For i = $m$ down to $k$
      i.  Find the security with the largest deviation from the target
      ii. Remove the security from $B$ if it stills exists in the set
```

Where m is the number of securities available in the universe of securities. Functions `tryout(…)` and `exclude(…)` would perform the above operations.

One superiority of selective inclusion over selective exclusion is the fewer number of computations it requires given that $k < m/2$, which is often the case.

To generalize the proposed solution, at the cost of computational complexity, we can start with an initial basket – that is not necessarily empty nor contains all available securities – and take 'right' steps towards a local optimum. The pseudocode for a more general algorithm would be as follows:

```
1. Set $B$ to an initial basket with $k$ securities
2. For all $s \in B$ and for all $t \notin B$
      i.  Compute the deviation of the basket $B \cup \{t\} \setminus \{s\}$
      ii. If the new basket has smaller deviation than the current basket, include $t$, exclude $s$ from the basket
```

Central to the algorithm proposed above, is starting with an initially 'good' solution. This is because all three algorithms are greedy algorithms, meaning that they all take the best decision at the current iteration of the algorithm. The functions `tryswop(…)` is implemented for the above algorithm.

I would suspect that general algorithm is more robust than both selective inclusion and selective exclusion. This is especially true when randomly initializing the algorithm multiple times to ensure the algorithm does not get trapped in local minima. However, the general algorithm is computationally more expensive than both other algorithms.

5. Capital allocation

To determine the importance of each of basket securities in hedging the target, we need to find the corresponding weights (the second subproblem referred to in section 4). This would also allow us to invest our funds ($10 million for example) according to the weight of each security.

Assuming W_i as the weight of security X_i, we can formulate the following equation to track the target security y.

$$\sum_{i=1}^{k} W_i X_i = y \tag{5}$$

Which can be write in the matrix form as

$$\begin{bmatrix} | & | & & | \\ X_1 & X_2 & \dots & X_k \\ | & | & & | \end{bmatrix} \begin{bmatrix} W_1 \\ \cdot \\ \cdot \\ \cdot \\ W_k \end{bmatrix} = y \tag{7}$$

Or more succinctly as

$$XW = y \tag{8}$$

In which X is a $(\Delta T, k)$, W is $(k, 1)$, and y is $(\Delta T, 1)$, where ΔT is the number of time steps in the planning horizon.

I use the least squares to solve the above equation to find weights of the securities in the basket. Note that equations 5 through 8 are subject to

$$\sum_{i=1}^{k} W_i = 1 \tag{9}$$

The solution to the above equation would result in a basket with the following performance over the planning horizon.

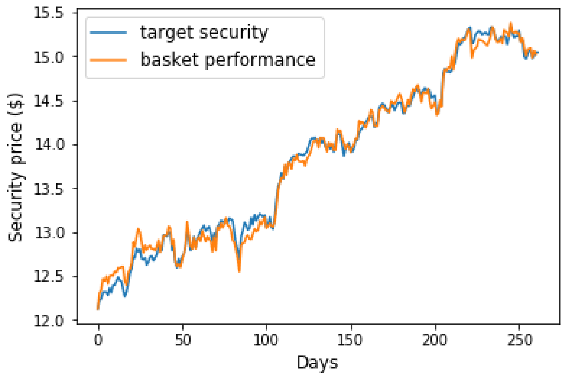

Fig. 12 | *performance of the same basket of securities to track target y_1, assuming unequal weights solved through least square solution to equation (8) subject to condition (9).*

For the sake of completeness, Table 1 below summarizes the basket securities as well as their corresponding weights for target y_1.

Table 1 | summarizes the basket securities as well as their corresponding weights to track target y_1.

Security #	125	73	140	162	318
Weight	0.27	0.19	0.21	0.19	0.14

Note that the above results take the planning horizon as the beginning to the end of the tracking period, consisting of 261 timesteps.

To evaluate the performance of the basket, I define the gain of a security, or a basket of securities, as the following

$$g(X; t_1, t_2) = {}^{\ln(X(t_2))}\big/_{\ln(X(t_1))} \tag{10}$$

Where *g(X; t_1, t_2)* is the gain of security *X* from *t_1* to *t_2*, and *X(t)* is the value of security *X* at time *t*.

For a basket of securities that track a target, we would like the basket to have a similar gain as the target. The target security of our example, y_1, has the gain of -0.216. The proposed basket, with the least square weightings, for this target security over the same period has the gain of -0.217 (0.4% difference). The gain of the same basket over the same period assuming equal weighting would be -0.208 (3.7% difference).

6. Scaling

If security prices were recorded at a much higher frequency (eg. thousands of observations per day), I probably would have need to use a deeper and/or longer model to fit the data. As result of higher frequency data acquisition, we will have a lot more data; smaller models may not be able to fit the training set well enough. Another hyperparameter that would be affected by the larger amount of data would be number of datapoints the model takes in to predict the next data point (T_x). Probably a larger T_x would be more appropriate.

The model can be retrained online as more data comes available. However, it can be deployed once the prediction of the model falls below an acceptable threshold of performance.

The code is vectorized to run fast on large datasets. This vectorization would especially be significant if we were to hedge thousands of securities instead of just four. Note that further vectorization to build the hedging basket is possible.

Storing the data in a cloud storage service is recommended. This is because of the pay-per-use model, backup, and the security such services provide. Google Cloud Platform (GCP), Amazon Web Services (AWS), and Microsoft Azure are the most notable cloud services. On AWS, for storing such datasets, I would suggest using AWS S3. The S3 instance is low latency, durable, and allows in-place query access to the data without having to move it to a separate analytics platform. The data in S3 is redundantly stored in S3 standard, S3 infrequent access, and S3 Glacier based on the frequency the data is used, making the entire dataset available for big data queries with simple SQL commands. The S3 instance is also scalable – a necessary feature for security price storage that can add up quickly.

7. Production-level code

The code is written in Jupyter notebook for the sake presentation. However, I tried to organize the code modularly, in functions or in classes.

In designing and implementing the algorithms, computational complexity was kept in mind, and the code is vectorized to quicken the runtime.

Each function includes docstrings that explain the task the function performs, as well as their expected input and output. The code is also heavily commented to help for future improvements. Function and variables were given meaningful names.

I used offline private versioning to keep track of my progress. However, this code is not publicly available though a public repo on GitHub or a similar database for privacy purposes.

8. References

[1] J. W. Kwiatkowski, “Algorithms for index tracking,” *IMA J. Manag. Math.*, vol. 4, no. 3, pp. 279–299, 1992.