MSBA7021 GROUP PROJECT

Portfolio Choices with Orthogonal Bandit Learning

By Group 1 From Subclass B

Song Cancan	3035675452
Wang Yanyuan	3035675684
Zeng Ni	3035675725
Zhou Zezhong	3035674812

Work Allocation

Name	UID	Responsibility
Song Cancan	3035675452	Modelling, Model Testing, Report
Wang Yanyuan	3035675684	Modelling, Model Testing, Report
Zeng Ni	3035675725	Data, Model Testing, Report
Zhou Zezhong	3035674812	Modelling, Model Testing, Report

Table of Contents

1. Introduction

Portfolio choice problems have had a profound influence on the finance industry. Modern portfolio theory and analysis build upon the seminal work of Markowitz. However, it has been proved to be highly sensitive to the expected rate of return and the covariance matrix which are hard to estimate, leading to an incomprehensible result in investment practices. Therefore, motivated by this noticeably poor performance in out-of-sample settings, we turn to the multi-armed bandit method which is a potent tool for designing on-line sequential decision strategies.

However, standard multi-armed bandits assume the rewards of each arm are drawn from i.i.d (independent and identically distributed) random variables, whereas in practice financial asset returns are generally correlated. Moreover, standard bandit learning attempts to choose the best arm for action, while in portfolio choice problems investors tend to select multiple assets for investment.

To grapple with those challenges in applying conventional bandit algorithms to portfolio choice problems, we use an orthogonal bandit learning algorithm to effectively make portfolio choices.

In particular, we take advantage of linear shrinkage, nonlinear shrinkage and multi-factor model for a better covariance matrix estimation, choose LSTM and prophet methods to predict the expected rate of return, adopt 3 different search policies: Softmax, Annealing ϵ-Greedy and the UCB1.

Further, we take advantage of the principal component decomposition to orthogonalize correlated assets based on the parallel back-testing, choose Sharpe Ratio as the risk adjusted reward function in the upper confidence bound, and combine the generated passive and active portfolio weights to construct a low-risk portfolio.

Moreover, to validate the proposed strategy, we evaluate the performance from cumulative wealth, compared with EW portfolio, Minimum-Variance portfolio, Naïve Bandit Portfolio (no correlation among arms).

2. Literature Review

2.1 Covariance Matrix Estimate

Covariance matrices play a central role in risk management and portfolio allocation, therefore, we need a good covariance matrix estimator which does not excessively amplify the estimation error. The standard method computes the sample covariance matrix using the history of past stock returns, which leads to a lot of errors (Jobson and Korkie, 1980). The estimated coefficients in the sample covariance matrix that are extremely high tend to contain a lot of positive and negative error, and Linear shrinkage method (Ledoit and Wolf, 2004) is proposed to compensate for those errors by pulling the most extremes coefficients towards the central values. A Nonlinear shrinkage method (Ledoit and Wolf, 2017) with nonparametric kernel estimation is provided. Motivated by the Arbitrage Pricing Theory in finance, a Fama-French three-factor model (Fan et al., 2008) is employed to reduce dimensionality and to estimate the covariance matrix.

2.2 Bandit Learning

Up to now, the mean-variance strategy cannot be adapted to the changing environment. The multi-armed bandit problem has been studied since the early 1950s (Robbins, 1952). It was designed to acquire new information while optimizing rewards based on the empirical observations motivated by dynamic rise and fall of the asset prices, which is known as the tradeoff between exploitation and

exploration. Therefore, this tradeoff naturally establishes a connection to the sequential decision process in portfolio choice problems, a multi-armed bandit strategy (Hoffman et al., 2011) was used to design the portfolio of acquisition functions in Bayesian optimization. To address the issue of risk in multi-armed bandit problems, Vakili and Zhao (2016) develop parallel results under the measure of mean-variance. Also, Huo and Fu (2017) incorporate risk-awareness into the classic multi-armed bandit setting and introduce a novel algorithm for portfolio construction. To grapple with the two challenges - one is standard multi-armed bandits assume the rewards of each arm are drawn from i.i.d random variables, whereas in practice financial asset returns are generally correlated, the other is although combinatorial multi-armed bandit algorithms have been proposed to select multiple arms, it makes binary decisions of selecting arms and equally distributes investments among them, in contrast, the crux of portfolio choice problems lies in determining the optimal distributing weights among assets, an orthogonal bandit learning (Chen, et al., 2015) was provided. The algorithm intends to maximize the cumulative wealth at the end of the multi-period investment horizon. For our comparative study, we also consider the Naive bandit portfolio (NBP) without considering significant and insignificant.

3. Methodology

In this section, we will introduce the methodology that we compute the weights $\boldsymbol{w_k} = [w_{k,1}, \dots, w_{k,n}]$ of n assets in a portfolio.

For a given historical return from *k-τ* to *k-1* of n assets, we can predict the expected return $\boldsymbol{E}(\boldsymbol{R}_k)$ and covariance matrix of their return $\boldsymbol{\Sigma}_k$ at time k. Then we implement Principal Component Decomposition (or Singular Value Decomposition) on the covariance matrix, the result derives:

$$\boldsymbol{\Sigma}_k = \boldsymbol{H_k \Lambda_k H_k^T} = \begin{bmatrix} | & & | \\ H_{k,1} & \dots & H_{k,n} \\ | & & | \end{bmatrix} \begin{bmatrix} \lambda_{k,1} & & 0 \\ & \ddots & \\ 0 & & \lambda_{k,n} \end{bmatrix} \begin{bmatrix} | & & | \\ H_{k,1} & \dots & H_{k,n} \\ | & & | \end{bmatrix}^{\mathrm{T}}, \quad (1)$$

where the matrix $\boldsymbol{H_k}$ is an orthogonal matrix which contains all the eigenvectors of the covariance matrix and the corresponding eigenvalues are at the second matrix, a diagonal matrix containing all the eigenvalues of covariance and permute in descending order.

To make the eigenvectors as the weights in a portfolio choice, the sum of all elements in an eigenvector should equal to 1. Therefore, we further normalize the eigenvector by:

$$\tilde{H}_{k,i} = \frac{H_{k,i}}{H_{k,i}^{\mathrm{T}}\mathbf{1}}, \quad (2)$$

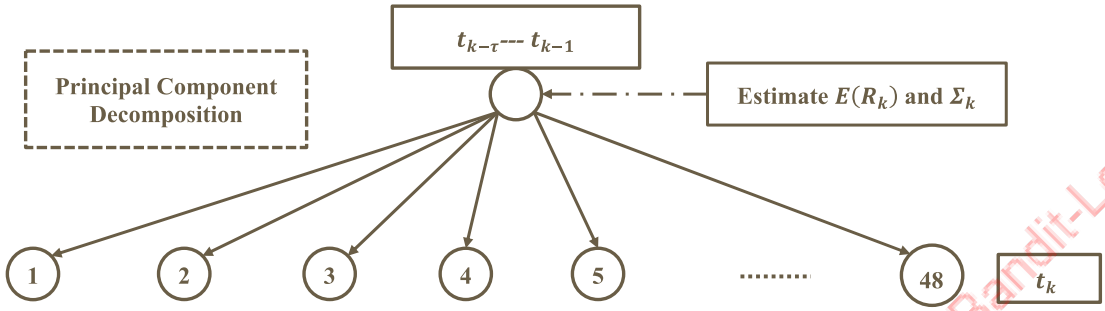
after this process, the normalized eigenvector $\tilde{H}_{k,i}$ can be represented as the i^{th} orthogonal portfolio strategy at time k. The reason we call it the orthogonal portfolio is that we have applied principal component decomposition, and thus every eigenvector, representing a portfolio choice, is orthogonal and uncorrelated with each other. In other words, by using the principal component decomposition, we convert n naturally correlated assets to n uncorrelated portfolios, and the return of the i^{th} portfolio at time k can be calculated as $\tilde{\boldsymbol{H}}_k^{\ T} \boldsymbol{R}_k$. Also, we can compute the variance of the set of n orthogonal portfolios by:

$$\tilde{\boldsymbol{\Sigma}}_{\boldsymbol{k}} = \tilde{\boldsymbol{H}}_k \boldsymbol{\Sigma}_k \tilde{\boldsymbol{H}}_k^{\ T} = \tilde{\boldsymbol{\Lambda}}_k = \begin{bmatrix} \tilde{\lambda}_{k,1} & & 0 \\ & \ddots & \\ 0 & & \tilde{\lambda}_{k,n} \end{bmatrix}, \quad (3)$$

where $\tilde{\lambda}_{k,i}$ is calculated by $\frac{\lambda_{k,1}}{(\tilde{H}_k^{\ T}1)^2}$.

Figure 1 demonstrates the process in our project that we use the historical data of 48 assets from time $k-\tau$ to $k-1$ to build 48 orthogonal portfolios by Principal Component Decomposition.

Figure 1: Bandit Learning Process-1

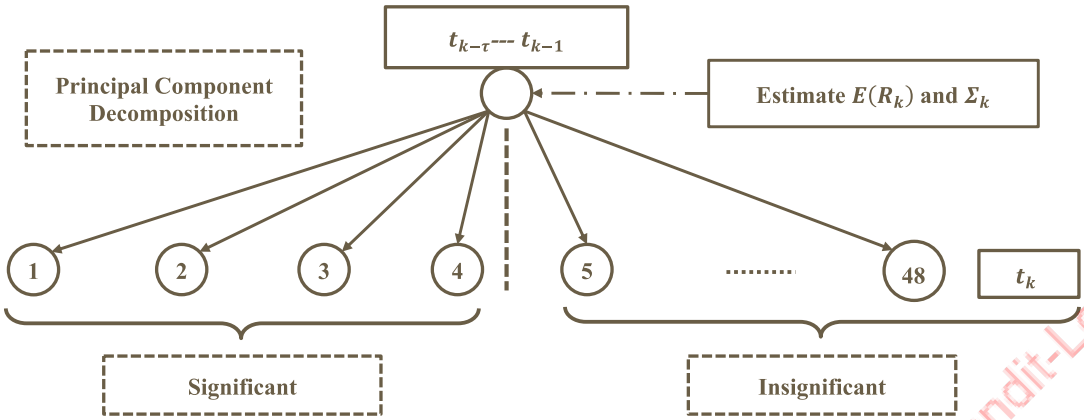


In our project, we compare two different bandit learning strategies, one is called naive bandit strategies (NB for simplicity) and the other is our proposed method (OB for simplicity). For the NB method, we just use bandit learning to choose the best arm each time from the pool of n orthogonal portfolios at time k. For the OB method, we first split the orthogonal portfolios into two sub-classes, and treat each of the class as an independent slot machine, which is defined as the Bi-Slot Machine Policy. At each time k, we will choose one best arm from each of the slot machines, respectively. The rationale is that according to Meucci (2009), the orthogonal portfolios we get above represent the risk factors in the market. At time t_k the return and the variance of the i^{th} orthogonal portfolio are estimated as $\tilde{\lambda}_{k,i}$, respectively. As many studies in finance indicated that only a few factors in the covariance matrix are significant and the rest are insignificant (Bai and Ng, 2002; Meucci, 2009), we split the decomposed covariance matrix into 2 classes based on the magnitude of $\tilde{\lambda}_{k,i}$ by the formula:

$$\tilde{\Sigma}_k = \underbrace{\sum_{i=1}^{l} \tilde{\lambda}_{k,i} \tilde{H}_{k,i} \tilde{H}_{k,i}^{T}}_{significant} + \underbrace{\sum_{i=l+1}^{n} \tilde{\lambda}_{k,i} \tilde{H}_{k,i} \tilde{H}_{k,i}^{T}}_{insignificant}, \quad (4)$$

We treat the first l factors as the significant factors of the covariance because they hold a large magnitude of lambda, meaning they make up a large part of variance from the covariance matrix and indicate the market systematic movement. The rest $n-l$ factors are viewed as the idiosyncratic risks which investors need to make more efforts to explore the opportunity and generate the extra return. There is no golden rule about how to choose the hyper-parameter l and most of the time we need to choose by checking the relevant difference between each of the neighboring λ, like the criterion we choose the number of principal components in PCA. According to Fama and French (1993) , the first 3-5 factors will be considered as the significant one in practice, so for simplicity, we set $l=4$ in our model. Figure 2 demonstrates the process we mentioned above.

Figure 2: Bandit Learning Process-2



For both the NB and OB method, we use a new proxy of reward, Sharpe Ratio (SR). A strength of SR is that it considers both the risk and return but doesn't need self-defined parameters. The formula of calculating Sharpe Ratio is:

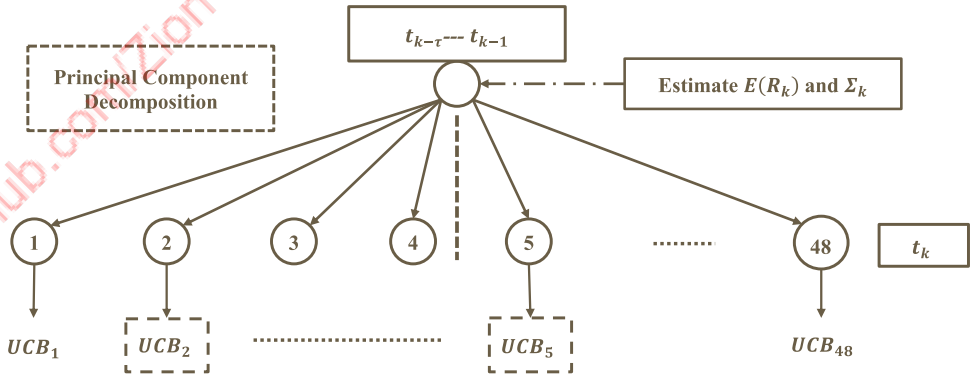
$$SR_{k,i} = \frac{E\left[\widetilde{H}_{k,i}R_{k,i}\right]}{\sqrt{\tilde{\lambda}_{k,i}}}, \tag{5}$$

For example, in UCB1 algorithm, the adjusted reward function would be:

$$UCB_i = \bar{r}_{i,k} + \sqrt{\frac{2\ln(K+\tau)}{\tau + k_i}} = \frac{E[\widetilde{H}_{k,i}R_{k,i}]}{\sqrt{\tilde{\lambda}_{k,i}}} + \sqrt{\frac{2\ln(K+\tau)}{\tau + k_i}}, \tag{6}$$

At the end the process of OB, we get two best arms from the two slot machines, as shown in Figure 3 (Arm 2 and Arm 5 are selected, for instance).

Figure 3: Bandit Learning Process-3



Then, we use θ to combine the two portfolios plans into one merged portfolio plan by using the equation:

$$\omega_k = (1-\theta_k)\widetilde{H}_{k,ik} + \theta_k \widetilde{H}_{k,jk}, \tag{7}$$

The method of choosing optimal θ is to minimize the total variance of the merged portfolio. More precisely, the optimal θ^* is computed by

$$\theta_k^* = \underset{\theta_k}{\operatorname{argmin}} \lambda_{k,p} = \frac{\tilde{\lambda}_{k,i}}{\tilde{\lambda}_{k,i} + \tilde{\lambda}_{k,j}}, \tag{8}$$

Finally, we attain a mixed portfolio containing both significant and insignificant factors.

During the back-testing process, we first collect all the choices made at each time $k = 1, 2, \ldots, m$. Then we use the real return rate of the n asset at time k to compute the realized revenue we get. For the performance metrics, we use the cumulative wealth (CW). Starting the investment period with c dollars (by default, $c = 1$), CW is computed by:

$$CW = c \times \prod_{k=1}^{m} \boldsymbol{\omega}_k{}^T R_k, \tag{9}$$

4. Experiments

4.1 Datasets

In our experiments, we choose two benchmark datasets for performance validation and comparison, which are FF48 and FF100 from the Fama and French (FF) datasets. With the raw data from the US stock market, the FF benchmarks construct the portfolios for different financial segments. Specifically, the FF48 dataset contains monthly returns of 48 portfolios representing different industrial sectors, and the FF100 dataset includes monthly returns of 100 portfolios based on size and book-to-market ratio. The reason why we decided to use them is that they are recognized as standard evaluation datasets and frequently adopted as testbeds in the finance community due to its extensive coverage to asset classes and lengthy periods.

4.2 Experimental Settings

4.2.1 Rolling-horizon

We use sliding windows with the size of 120 months of training data from Jan 1980 to Dec 2019

4.2.2 Comparative Methods

On one hand, we choose two typical baselines circulated in the finance community.

(1) Equally-weighted Portfolio (EW)

The EW portfolio is a naive approach yet has been empirically shown to mostly outperform 14 models across seven empirical datasets (DeMiguel et al., 2009).

(2) Minimum-variance Portfolio (MVP)

In 1952, Markowitz introduced mean-variance analysis and suggested choosing the allocation that maximizes the expected return for a certain risk level quantified by variance.

On the other hand, we choose two sequential decision-making approaches, NB and proposed OB. We present these two bandit learning algorithms in the three search frameworks: Softmax, ϵ-Annealing Greedy and UCB1 algorithm to compare their performance.

4.2.3 Covariance Matrix Estimation

The first step in our experiment is to estimate the covariance matrix of the return. The following are three methods we used:

(1) Linear Shrinkage

The first method is to apply a linear shrinkage estimation. The goal is to remedy the sample covariance matrix and pull the most extreme coefficients towards more central values，thereby systematically reducing estimation errors where it matters most (Ledoit & Wolf, 2004).

(2) Non-linear Shrinkage

More recently, some non-linear shrinkage method has been developed to improve the speed and accuracy of the linear one. In our experiments, we used the nonparametric kernel estimation from Ledoit and wolf (Ledoit and Wolf, 2017).

(3) Multi-factor Model

The last method is to take advantage of a multi-factor model to reduce dimensionality and then to estimate the covariance matrix (Fan et al., 2008).

We first establish a Fama-French three-factor model which needs $4p$ instead of $p(p+1)/2$ parameters to be estimated. The first factor f_1 is the excess return of the proxy of the market portfolio. The other two factors are constructed using sixteen value-weighted portfolios formed on size and book-to-market.

Then we plug in the least-squares estimators of regression coefficients $\hat{B}_n$, sample covariance matrix of the factors , $\widehat{cov}(f)$,and the diagonal matrix of residuals $\hat{\Sigma}_{n,0} = diag(n^{-1}\hat{E}\hat{E}')$ with $\hat{E} = Y - \hat{B}X$.Therefore, we have a substitution estimator:

$$\hat{\Sigma}_n = \hat{B}_n \widehat{cov}(f) \widehat{B_n}' + \hat{\Sigma}_{n,0}, \tag{10}$$

4.2.4 Expected Return Estimation

In the next step, we need to estimate the average return rate for the next month.

For our comparative study, we implement three methods to do the forecasting.

Firstly, we adopt a naive method: Mean of return method. Specifically, we take the average of the last 120 months and treat this value as our naive estimation for return of the next month.

Then, we apply a deep-learning method: Long Short Term Memory model. It is a type of Recurrent Neural Network that captures both short-term and long-term information. In our LSTM model, we set one hidden layer with 6 nodes, which was validated to have good performance in predicting assets return.

Besides, we also use the Prophet (Facebook's library for time series forecasting) to forecast the return rate on time series.

Prophet uses a decomposable time series model with three main model components: growth, seasonality and holidays. They are combined using the equation:

$$y(t) = g(t) + s(t) + h(t) + \epsilon_t, \tag{11}$$

In the above formula, where $g(t)$ models trend, the long-term increase or decrease, $s(t)$ models seasonality with Fourier series, $h(t)$ models the effects of holidays or large events, e.g. Black-Monday and ϵ_t represents an irreducible error term.

4.2.5 Search Policy

To determine how to invest in those two subsets, we apply three search algorithms to the multi-armed bandit.

(1) Softmax Exploration Algorithm

Moving beyond the greedy algorithm,the Softmax incorporates exploitation by using it to increase the chance of picking the higher return arm, while also making it possible to pick the lower return arm (which is some form of exploration). Specifically, it proposes the following probability distribution of choosing each arm at each given round:

$$\frac{\exp\left(\frac{\widehat{\mu_t}(k)}{\text{temperature}}\right)}{\sum_{i=1}^{K}\exp\left(\frac{\widehat{\mu_t}(i)}{\text{temperature}}\right)}, \tag{12}$$

The temperature parameter is a hyperparameter that ultimately determines how much randomisation. In our experiment, we fix the value of temperature to be 0.1

(2) Annealing ϵ-Greedy Algorithm

The improvement to the traditional ϵ-greedy algorithm is that it makes the algorithm parameter-free. You can specify the rule of decaying ϵ with time. The rule of we will use here is:

$$\epsilon = \frac{1}{\log(\text{time} + 0.0001)}, \tag{13}$$

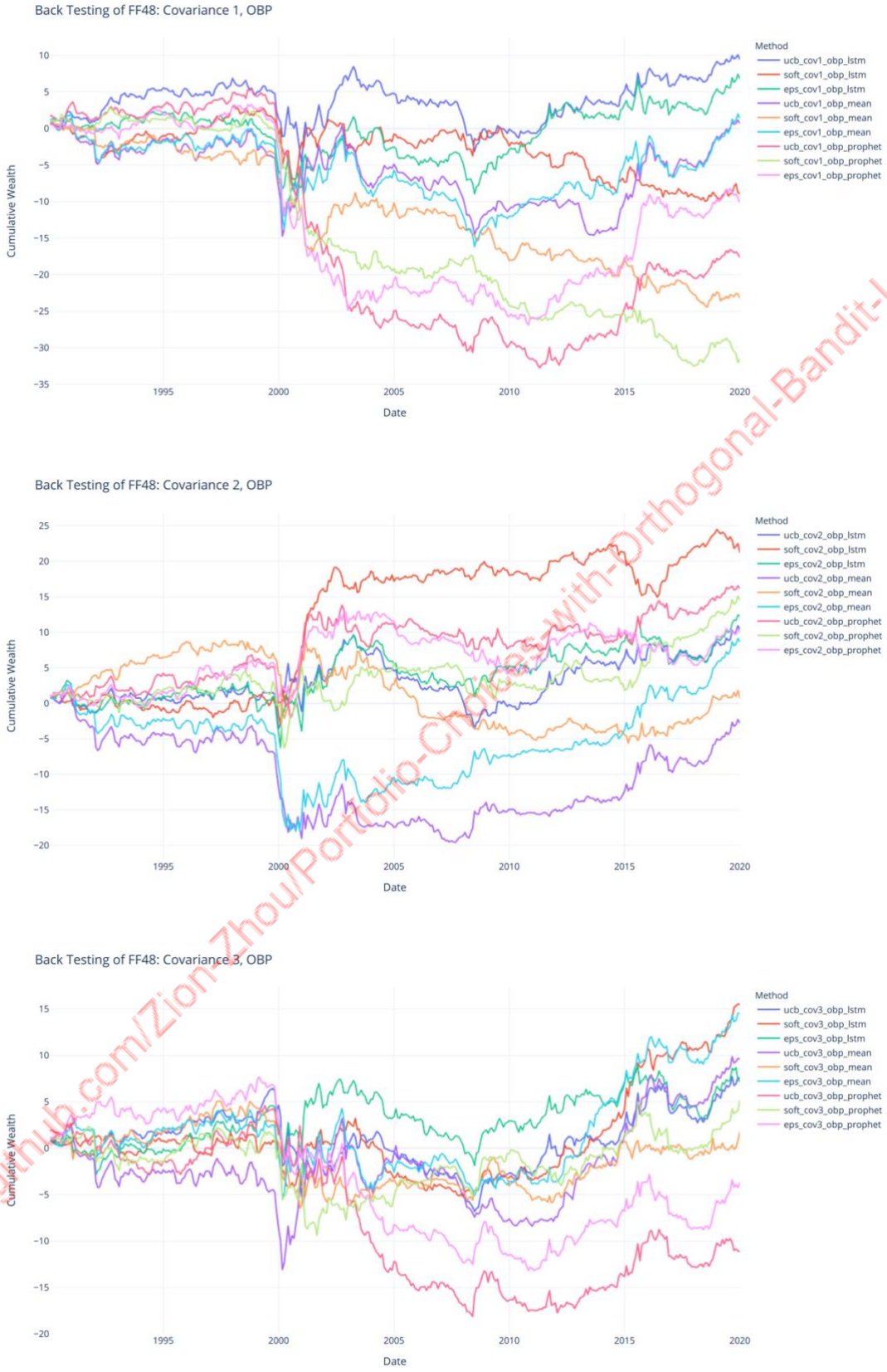
(3) Upper Confidence bound

On the other hand, different from the randomized policy as the ϵ-greedy approach, the upper confidence bounds (UCB1) strategy has emerged as another popular choice for multi-armed bandit problems (Lai and Robbins, 1985]. Under UCB1 we make our selections based on how uncertain we are about a given selection.

5. Analysis

Based upon the rolling horizon setting, the back-testing is accomplished in the two datasets: FF48 and FF100, from January 1990 to December 2019 (the first sidling window is from January 1980 to December 1990).

Figure 4: Back-Testing for FF48 With OB



According to Figure 4, the cumulative wealth of the orthogonal bandit portfolio strategy with the linear shrinkage estimated covariance matrix ranges from -35 to 10, while that of the one with the kernel

shrinkage estimated covariance matrix ranges from -20 to 25. It appears that the performance of OB strategy using the Fama French Three Factor Model is relatively stable, whose cumulative return is somewhere between -20 and 15.

More specifically, the OB strategy using the kernel shrinkage method along with LSTM and Softmax algorithm in bandit learning is the optimal one among the 27 variants of OB.

Taken together, in terms of estimating the Covariance matrices, nonlinear shrinkage with non parametric kernel estimation has a better performance. Linear shrinkage method gives more and greater negative cumulative wealth. Multi-factor model method is more sensitive to the financial crisis,resulting in a large loss of cumulative wealth during the period. On the other hand, nonlinear shrinkage method is relatively robustter in shock resistance.

As for the comparison in predictive approaches for expected return, convolution neural network is the most powerful one, which stably contributes profitable choices, whereas mean estimate and Prophet are somewhat sensitive to the methods adopted for estimating the covariance matrix as mean estimate performs better under the multi-factor model, and Prophet presents a modest performance only when the kernel shrinkage method is used.

Figure 5: Back-Testing for FF48 With NB

Figure 5 indicates that when Naïve Bandit strategy is applied, the cumulative return in back-testing is less volatile and less sensitive to the methods we used on the estimation or the searching algorithm. Such a situation is reasonable as after the principal component decomposition is performed, the eigenvectors will not be further decomposed into the passive class and the active class, implying that it's the issue of a single multi-armed bandit problem. At each one of the decision points, only one arm out of the 48 alternatives will be chosen based upon the reward function in bandit learning. With a larger alternative set, and no additional requirement imposed on the composition of the portfolio (passive part, negative part), the probability of choosing a portfolio that follows the general trend of the stock market will increase substantially. Higher level of return is always corresponding to higher level of risk, thus, a smaller probability to allocate the capitals to the active asset seeking for extra return will significantly decrease the magnitude of the fluctuation. However, the investors who are risk-neutral or risk-tolerant will benefit more from the OB strategy rather than the NB strategy.

The NB strategy presents a pattern that is similar to the OB strategy: the kernel shrinkage method used to estimate the covariance matrix with convolution neural network contributes most to the improvement of the performance in the back-testing. It is noticed that after adopting the kernel shrinkage method with convolution neural networks, the portfolios had become significantly more resilient to shocks, falling only slightly during the financial crisis of 2000, but quickly retracted and accumulated wealth has steadily increased even when facing the financial crisis in 2008.

Based on the discovery that the kernel shrinkage appears to be the best method in terms of covariance matrix estimation, the following analysis will further figure out the impacts on the prediction methods used for expected return under the assumption that the kernel shrinkage is adopted.

Figure 6: Back-Testing for FF48 With OB Based on Kernel Shrinkage

As shown in Figure 6, the OB strategy with LSTM steadily makes portfolios with cumulative wealth all greater than 10 times. Moreover, Softmax algorithm dominates over the rest of methods in terms of

the general cumulative wealth level and the magnitude of the volatility. Among all the search policies, Softmax shows the adaptability, obtaining relatively more stable performances even in the declining year.

Beyond that, Prophet stably gives a cumulative wealth between 10 and 15 times of the initial investment. When the Prophet is adopted, UCB1 turns out to be the optimal exploration-exploitation policy but not a conservative one as it is usually along with high volatility. In years with severe market fluctuations, the portfolio will also produce large fluctuations with the market, but the final result can be neutralized. However, when using the historical mean return as the prediction, all policies suffer from high volatility, and annealing ϵ-greedy policy has a better performance only after the year of 2015.

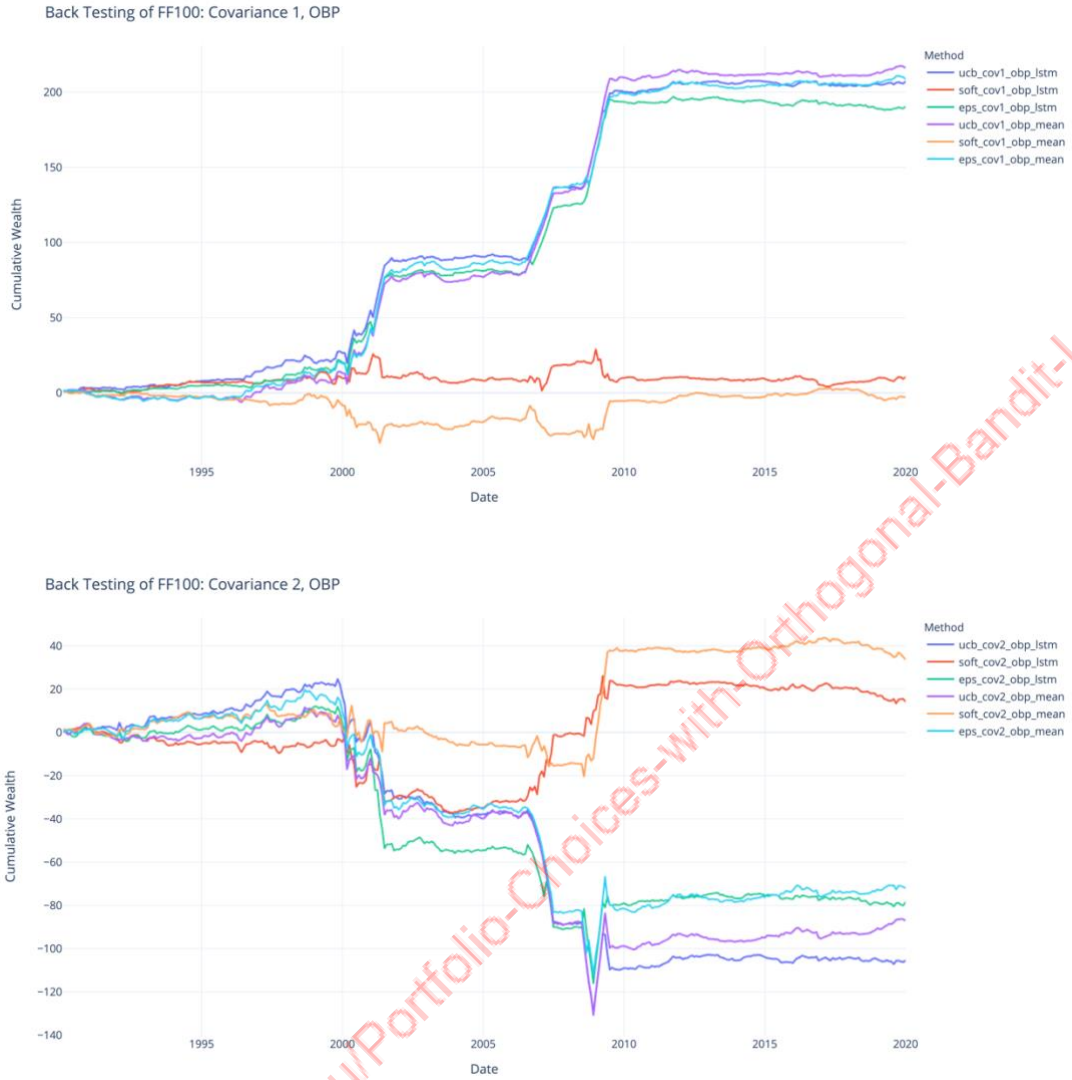
There is one interesting finding that when we apply the LSTM algorithm on predicting the expected return. The result of the Softmax is much different than the result of Annealing ϵ-greedy and UCB1 algorithms. Such a phenomenon is probably attributed to the fact that when the estimation of the return is not so robust. Softmax strategy can still have a relatively good performance because of the adaptability we discovered before. For the search methods that are not so robust, the performance will be much lower but can still make the money. A piece of advice for the investors is that when you are not confident in your prediction of return, Softmax is more recommendable, but if you are confident in your prediction algorithm, you can try to use UCB1 strategy which has potential to make more money for a long-term investment.

Figure 7: Comparison Between Proposed Strategy and Benchmark

Figure 7 visualizes the OB strategy using kernel shrinkage for covariance estimation, LSTM for return prediction and Softmax searching algorithm against Equally weighted portfolio, which is always a benchmark in portfolio performance testing. Intuitively, the proposed strategy, OB, is superior to the benchmark, especially from the year around 2001.

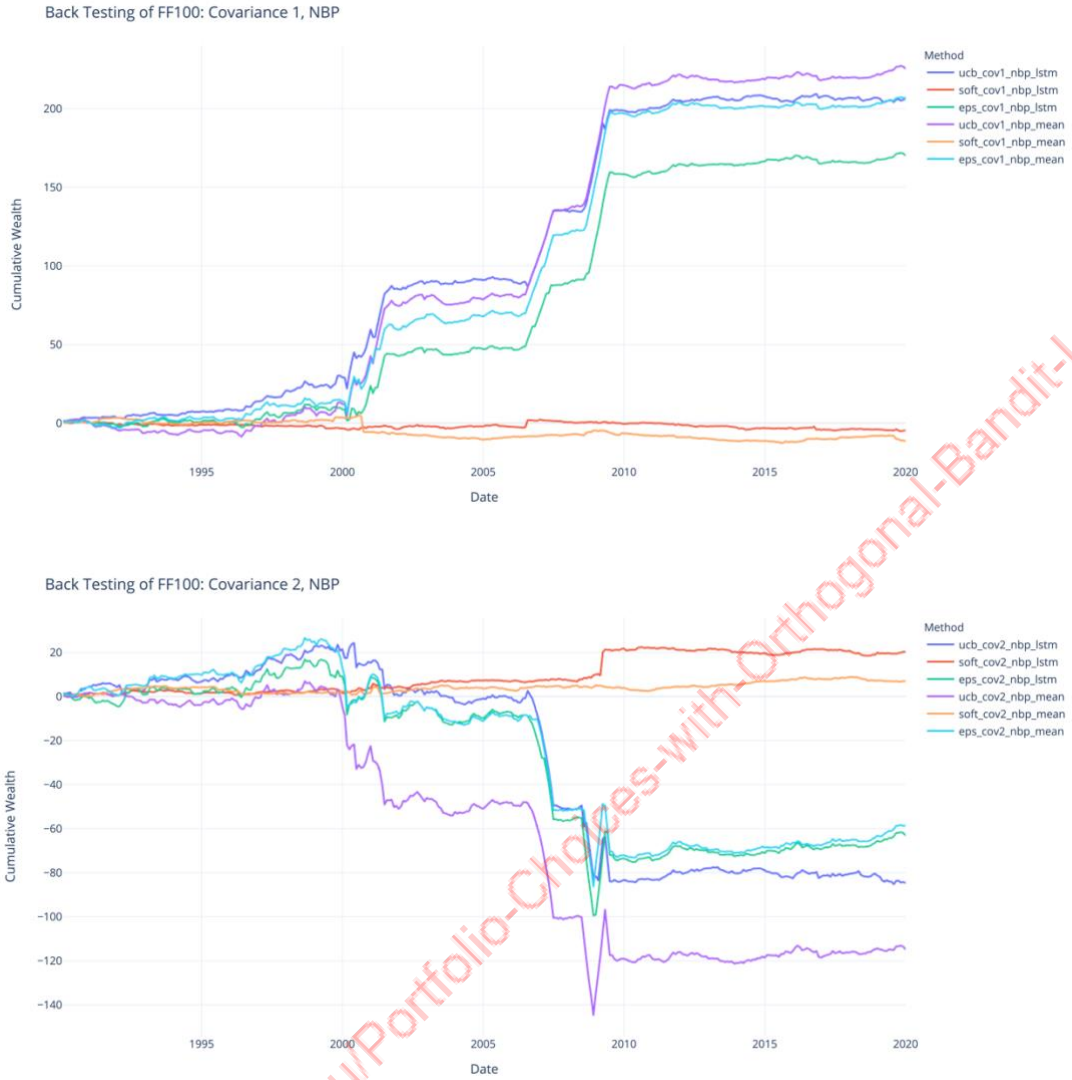
For validation purpose, a back-testing on the dataset of FF100 is conducted. However, due to the complexity of Data FF100, the three factors are not available on the Internet. We can only apply the first two methods on the covariance estimation.

Figure 8: Back-Testing for FF100 With OB



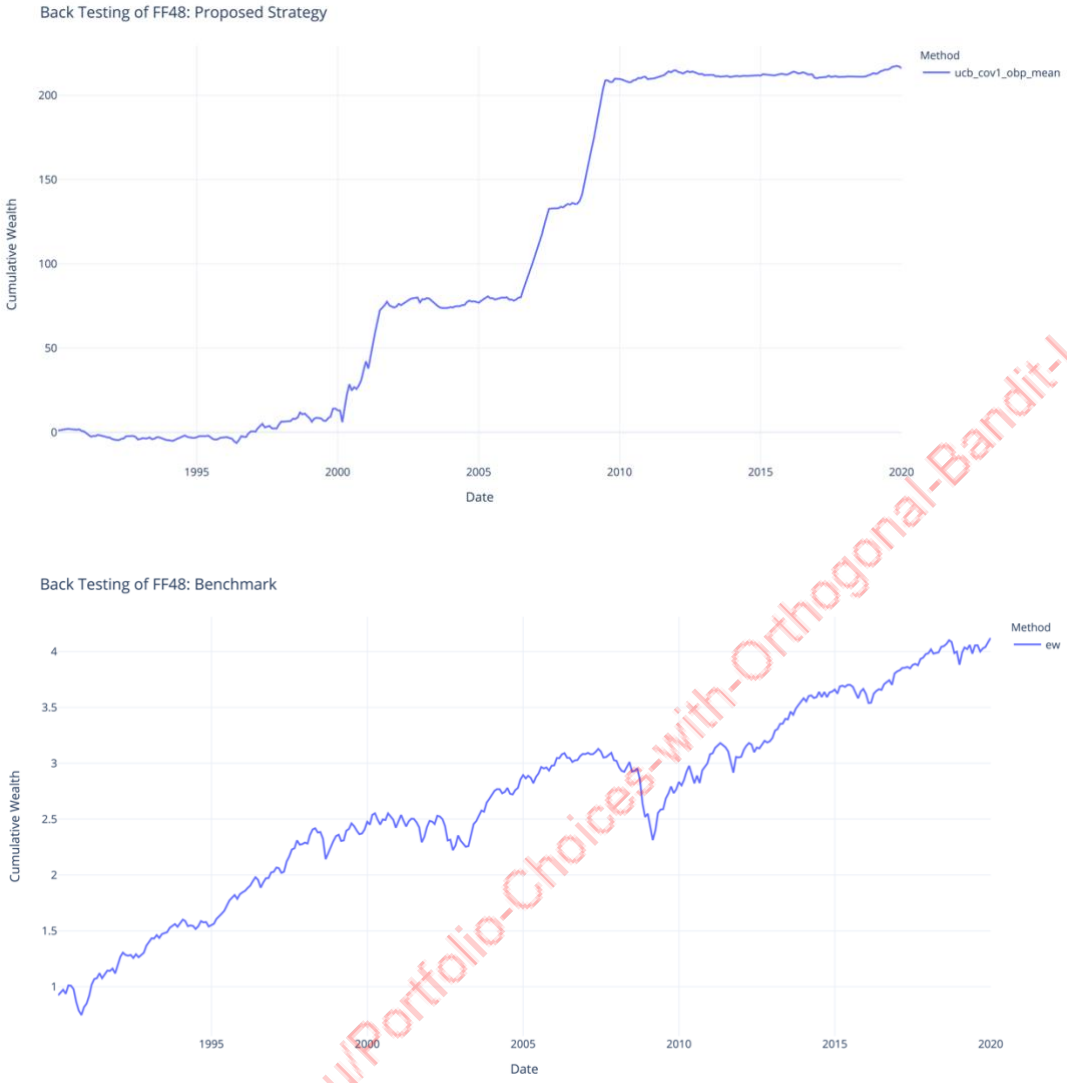
Surprisingly, the OP and NP share a very similar pattern that the cumulative wealth of UCB1 and annealing ϵ-greedy have some bottlenecks of increment. While the performance of Softmax, by comparison, is not satisfactory, but still stable.

Figure 9: Back-Testing for FF100 With NB



Like the discovery we find above, even if we changed the method on the covariance estimation, the cumulative wealth of UCB1 and Annealing ϵ-greedy also share a similar pattern, but this time they lose money. The Softmax algorithm keeps its stability and adaptability all the time.

Figure 10: Comparison Between Proposed Strategy and Benchmark



The following table summarizes the performance of the optimal OB strategy with Bi-Slot Machine Policy for the dataset FF48 and FF100. By comparing the performance between groups, it appears the FF100 dataset possess more potentials than FF48 as for the long-term investment.

Table 1: Back-Testing Results

Proposed OB Strategy	Covariance Matrix Estimation	Expected Return Prediction	Bandit Algorithm	Cumulative Wealth (Dec 2019)
FF48	Kernel Shrinkage	LSTM	Softmax	21.25
FF100	Linear Shrinkage	Mean	UCB1	216.06

6. Conclusion

In conclusion, we apply principal component decomposition method to convert correlated assets to uncorrelated portfolios, called orthogonal portfolios. During the bandit learning process, we use Sharpe

ratio as the reward proxy that considers both return and risks, and the Bi-Slot Machine Policy helps us to include both the significant and insignificant factors in the final portfolio with a low variance. From the result of back-testing, the proposed OB method has good performance in both datasets. Considering the search policy of the proposed method, the accuracy of the return prediction plays a huge role in the result. When we use very rough predictions, like the mean of the historical return, only the Softmax algorithm can attain an acceptable return, but if we have a more accurate or reliable prediction method, UCB1 has a better performance on average. However, there are some limitations in our project. For example, our method could not be used in the market that prohibits short selling, since we can not ensure every weight of the assets to be non-negative. Moreover, due to the time limit, we didn't tune the hyper-parameter of the Softmax algorithm with cross-validation. As in prescriptive analytics, every process is closely connected with each other. To construct a good portfolio, we need a more accurate estimation method and a more stable decision algorithm. Besides, there are a lot of other realistic factors that we need to consider during the process, such as the transaction cost and tax. Under this framework, we need more constraints to make it more practical.

7. Limitations and Further Studies

Our further studies can be carried out from the following perspectives.

When applying our proposed OB strategy to the stock market where short selling is not allowed, like the Chinese stock market, additional non-negative weight constraints should be added to the portfolio optimization model. Beyond that, we ought to take the transaction costs, taxes and position constraints into consideration.

In addition, we should perform back-testing of OB strategy in diversified types of datasets that will help to improve the robustness of our portfolio management, like ETF139, EQ181 and SP500. Note that FF48 and FF100 are monthly data that underscore the long-term performance, while ETF139, EQ181 and SP500 is composed of actively traded and typically accessible assets in a higher frequency.

As an alternative validation approach, comparing with more portfolio management strategy, like value-weighted portfolio or online moving average reversion. In terms of the methodologies, James Stein Estimator will be more suitable to estimate the expected return in a high-dimensional dataset.

Apart from that, an empirical study points out that Thompson sampling, a type of Bayesian inference based on β distribution to update the posterior probability based on the prior probability according to investors' risk preferences, outperforms other peer methods in the real-world cases.

In order to assess the risk-return tradeoff, Sharpe Ratio that measures the average return gained in excess of the risk-free rate per unit of volatility can be used to reflect the out-of-sample performance. Given that the distribution of the return of assets in the OB portfolio has long tails than that of the normal distribution, we should apply the studentized circular block bootstrapping method to figure out whether the differences in Sharpe ratios between two investment strategies are statistically significant or not (Ledoit & Wolf, 2008).

Note that in our experiment, we arbitrarily set τ to be 0.1, thus, to further improve the performance of our trading strategy, we can tune the hyper-paramter with cross validation, such as temperature in Softmax algorithm. Furthermore, we can even optimize the structure of LSTM, for instance, adding a dropout layer after the convolution layer to avoid over-fitting.

8. Appendix 1: Reference List

[1] [Agrawaland Goyal,2012] S.Agrawal and N.Goyal. Analysis of Thompson sampling for the multi-armed bandit problem. InProceedings of the 25th Annual Conference on Learning Theory, pages39.1–39.26, 2012.

[2] [Bai and Ng, 2002] J. Bai and S. Ng. Determining the number of factors in approximate factor models. Econometrica, 70(1):191-221, 2002.

[3] [DeMiguel et al., 2009] V. DeMiguel, L. Garlappi, and R. Uppal. Optimal versus naive diversification: How inefficient is the 1/N portfolio strategy? The Review of Financial Study, 22:1915-1953, 2009.

[4] [Fama and French, 1993] E. F. Fama and K. R. French. Common risk factors in the returns on stocks and bonds. Journal of Financial Economics, 33:3-56, 1993.

[5] [Fan et al., 2008] J. Fan, Y. Fan, and J. Lv. High dimensional covariance matrix estimation using a factor model. Journal of Econometrics, 147:186-197, 2008.

[6] [Lai and Robbins, 1985] T. L. Lai and H. Robbins. Asymptotically efficient adaptive allocation rules. Advances in applied mathematics,6(1):4-22, 1985.

[7] [Ledoit and Wolf,2004] Ledoit, O. and Wolf, M. (2004). A well-conditioned estimator for large-dimensional covariance matrices. Journal of Multivariate Analysis, 88(2):365-411.

[8] [Ledoit and Wolf,2018] Ledoit, Olivier and Wolf, Michael, Analytical Nonlinear Shrinkage of Large-Dimensional Covariance Matrices (November 2018). University of Zurich, Department of Economics, Working Paper No. 264, Revised version. Available at SSRN: https://ssrn.com/abstract=3047302.

[9] [LedoitandWolf,2008] O. Ledoit and M. Wolf. Robust performance hypothesis testing with the Sharpe ratio. Journal of EmpiricalFinance,15:850-859, 2008.

[10][Meucci, 2009] A. Meucci. Risk and asset allocation. Springer Science & Business Media, 2009.

9. Appendix 2: Code

9.1 Import Packages

```
import numpy as np
import pandas as pd

import tushare as ts
from fbprophet import Prophet
import pystan

import matplotlib.pyplot as plt
%matplotlib inline

from sklearn.decomposition import PCA
from sklearn.preprocessing import StandardScaler
from sklearn.preprocessing import MinMaxScaler
import statsmodels.api as sm
from statsmodels.stats.outliers_influence import summary_table

from tqdm.notebook import tqdm

import torch
import torch.nn as nn
from torch.autograd import Variable

import pyomo.environ as pe

from google.colab import drive
drive.mount('/content/drive')
import os
os.chdir('/content/drive/My Drive/HKU/MSBA 7021/MSBA 7021 Project')

from IPython.core.interactiveshell import InteractiveShell
InteractiveShell.ast_node_interactivity = "all"
```

9.2 Load Data

```
df = pd.read_csv('FF100.csv')
#df = pd.read_csv('FF48.csv')

start = list(df['Unnamed: 0']).index(198001)
end = list(df['Unnamed: 0']).index(201912)

data = df.iloc[start:end+1,]
data = data.reset_index(drop = True)
data = data.rename(columns = {data.columns[0]:"Date"})
```

9.3 Predict Return

9.3.1 LSTM

```
class LSTM(nn.Module):

    def __init__(self, num_classes, input_size, hidden_size, num_layers):
        super(LSTM, self).__init__()

        self.num_classes = num_classes
        self.num_layers = num_layers
        self.input_size = input_size
        self.hidden_size = hidden_size
        self.seq_length = seq_length
```

```
11.
12.         self.lstm = nn.LSTM(input_size = input_size, hidden_size = hidden_size,
13.                             num_layers = num_layers, batch_first = True)
14.         self.fc = nn.Linear(hidden_size, num_classes)
15.
16.     def forward(self, x):
17.         h_0 = Variable(torch.zeros(
18.             self.num_layers, x.size(0), self.hidden_size))
19.         c_0 = Variable(torch.zeros(
20.             self.num_layers, x.size(0), self.hidden_size))
21.
22.         # Propagate input through LSTM
23.         ula, (h_out, _) = self.lstm(x, (h_0, c_0))
24.         h_out = h_out.view(-1, self.hidden_size)
25.         out = self.fc(h_out)
26.
27.         return out
28.
29. def sliding_windows(data, seq_length):
30.     x = []
31.     y = []
32.
33.     for i in range(len(data)-seq_length):
34.         _x = data[i:(i+seq_length)]
35.         _y = data[i+seq_length]
36.         x.append(_x)
37.         y.append(_y)
38.
39.     return np.array(x),np.array(y)
40.
41. def LSTM_pred(infile, start_date, end_date, num_epochs, learning_rate,
42.                 input_size, hidden_size, num_layers, num_classes, outfile):
43.
44.     df = pd.read_csv(infile)
45.
46.     start = list(df['Unnamed: 0']).index(start_date)
47.     end = list(df['Unnamed: 0']).index(end_date)
48.
49.     data = df.iloc[start:end+1,]
50.     data = data.reset_index(drop = True)
51.     data = data.rename(columns = {ff100.columns[0]:"Date"})
52.
53.     result = {}
54.     R = {}
55.
56.     for i in range(len(data.columns)-1):
57.
58.         training_set = data.iloc[:,i+1:i+2].values
59.
60.         sc = MinMaxScaler()
61.         training_data = sc.fit_transform(training_set)
62.
63.         x, y = sliding_windows(training_data, seq_length)
64.
65.         train_size = int(len(y) * 0.67)
66.         test_size = len(y) - train_size
67.
68.         dataX = Variable(torch.Tensor(np.array(x)))
69.         dataY = Variable(torch.Tensor(np.array(y)))
70.
71.         trainX = Variable(torch.Tensor(np.array(x[0:train_size])))
72.         trainY = Variable(torch.Tensor(np.array(y[0:train_size])))
73.
74.         testX = Variable(torch.Tensor(np.array(x[train_size:len(x)])))
75.         testY = Variable(torch.Tensor(np.array(y[train_size:len(y)])))
76.
```

```
        lstm = LSTM(num_classes, input_size, hidden_size, num_layers)

        criterion = torch.nn.MSELoss()  # mean-squared error for regression
        optimizer = torch.optim.Adam(lstm.parameters(), lr = learning_rate)
        # optimizer = torch.optim.SGD(lstm.parameters(), lr = learning_rate)

        # train the model
        for epoch in range(num_epochs):
            outputs = lstm(trainX)
            optimizer.zero_grad()

            # obtain the loss function
            loss = criterion(outputs, trainY)
            loss.backward()
            optimizer.step()

            if epoch % 100 == 0:
                print("Epoch: %d, loss: %1.5f" % (epoch, loss.item()))

        lstm.eval()
        train_predict = lstm(dataX)

        data_predict = train_predict.data.numpy()
                dataY_plot = dataY.data.numpy()

                data_predict = sc.inverse_transform(data_predict)
                result[df.columns.tolist()[i+1]] = data_predict

           for idx,content in result.items():
                cont = [float(i) for i in content]
                R[idx] = cont

           Result = pd.DataFrame(R)
           Result.to_csv(outfile)
           return Result

       seq_length = 120

       LSTM_pred(infile = 'FF100.csv', start_date = 198001, end_date = 201912,
                 num_epochs = 100, learning_rate = 0.01, input_size = 1, hidden_siz
   e = 6,
                 num_layers = 1, num_classes = 1, outfile = "FF100_predictions.csv"
   )
```

9.3.2 *Prophet*

```
class Prophet_Predict:

    stock_code = ''
    tsData = pd.DataFrame()

    def __init__(self, stock_code):
        self.stock_code = stock_code

    def date_setting(self, start_date, end_date):
        self.tsData = ts.get_hist_data(code = self.stock_code, start = start_date,
   end=end_date)
        self.tsData = self.tsData.reset_index()

    def makePredictionByDay(self, node):
        new_data = pd.DataFrame(index = range(0, len(self.tsData)), columns = ['Dat
   e', 'Close'])
        for i in range(0, len(self.tsData)):
            new_data['Date'][i] = self.tsData['date'][i]
            new_data['Close'][i] = self.tsData['close'][i]
```

```
        new_data['Date'] = pd.to_datetime(new_data.Date, format = '%Y-%m-%d')
        new_data.index = new_data['Date']

        new_data = new_data.sort_index(ascending=True)
        new_data.rename(columns = {'Close': 'y', 'Date': 'ds'}, inplace = True)
        forecast_valid = []

        prediction = new_data[node:]
        for i in range(0, len(new_data) - node):
            train = new_data[:node + i]
            valid = new_data[node + i:]

            model = Prophet()
            model.fit(train)

            close_prices = model.make_future_dataframe(periods=1)
            forecast = model.predict(close_prices)
            forecast_valid.append(forecast['yhat'][len(train):len(train) + 1])
            print(forecast['yhat'][len(train):len(train) + 1])
        prediction['Predictions'] = forecast_valid
        plt.plot(train['y'])
        plt.plot(prediction[['y', 'Predictions']])
        plt.show()

    def makePrediction(self, node):
        new_data = pd.DataFrame(index = range(0, len(self.tsData)), columns  =['Dat
e', 'Close'])
        for i in range(0, len(self.tsData)):
            new_data['Date'][i] = self.tsData['date'][i]
            new_data['Close'][i] = self.tsData['close'][i]
        new_data['Date'] = pd.to_datetime(new_data.Date, format = '%Y-%m-%d')
        new_data.index = new_data['Date']

        new_data = new_data.sort_index(ascending = True)
        new_data.rename(columns = {'Close': 'y', 'Date': 'ds'}, inplace = True)
        forecast_valid = []

        train = new_data[:node]
        valid = new_data[node:]

        model = Prophet()
        model.fit(train)

        close_prices = model.make_future_dataframe(periods = len(valid))
        forecast = model.predict(close_prices)
        forecast_valid = forecast['yhat'][node:]

        valid['Predictions'] = forecast_valid.values
        plt.plot(train['y'], label = 'training set')
        plt.plot(valid[['y', 'Predictions']], label = ['truth', 'prediction'])
        plt.show()

# a = Prophet_Predict('000001')
# a.date_setting(start_date = '1980-01-01', end_date = '2019-12-31')
# a.makePrediction(100)
```


9.4 Estimate Covariance Matrix

9.4.1 Linear Shrinkage

```
def shrinkage(returns: np.array) -> Tuple[np.array, float, float]:
    t, n = returns.shape
    mean_returns = np.mean(returns, axis=0, keepdims=True)
    returns -= mean_returns
```

```
    sample_cov = returns.transpose() @ returns / t

    # sample average correlation
    var = np.diag(sample_cov).reshape(-1, 1)
    sqrt_var = var ** 0.5
    unit_cor_var = sqrt_var * sqrt_var.transpose()
    average_cor = ((sample_cov / unit_cor_var).sum() - n) / n / (n - 1)
    prior = average_cor * unit_cor_var
    np.fill_diagonal(prior, var)

    # pi-hat
    y = returns ** 2
    phi_mat = (y.transpose() @ y) / t - sample_cov ** 2
    phi = phi_mat.sum()

    # rho-hat
    theta_mat = ((returns ** 3).transpose() @ returns) / t - var * sample_cov
    np.fill_diagonal(theta_mat, 0)
    rho = (
      np.diag(phi_mat).sum()
      + average_cor * (1 / sqrt_var @ sqrt_var.transpose() * theta_mat).sum()
    )

    # gamma-hat
    gamma = np.linalg.norm(sample_cov - prior, "fro") ** 2

    # shrinkage constant
    kappa = (phi - rho) / gamma
    shrink = max(0, min(1, kappa / t))

    # estimator
    sigma = shrink * prior + (1 - shrink) * sample_cov

    return sigma, average_cor, shrink
Cov1 = []
for i in range(0,360):
    portfoilo_100_returns = data.drop(columns=['Date'])
    # change type to numpy array
    train = portfoilo_100_returns.iloc[0+i:120+i,].values
    # Using the Ledoit-
Wolf shrinkage to estimate the unreliable sample covariance.
    estimation_cov_1 = shrinkage(train)[0] # 48*48;array
    Cov1.append(estimation_cov_1)
```

9.4.2 *Kernel Shrinkage*

```
from direct_kernel import DirectKernel
Cov2 = []
for i in range(0,360):
    # change type to numpy array
    train = data.iloc[0+i:120+i,1:].values
    # Using the direct kernel shrinkage to estimate the unreliable sample covarianc
e.
    estimation_cov_2 = DirectKernel(train).estimate_cov_matrix() # 48*48;array
    Cov2.append(estimation_cov_2)
```

9.4.3 *Multi-factor Model*

```
df = pd.read_csv('data_for_factor_model.csv')
portfolios_names = df.keys()[2:50]
B = np.zeros([48,3])
res = np.zeros([48,120])
Cov3= []
for i in range(0,360):
```

```
    train = df.iloc[0+i:120+i,2:]
    f_cov = np.cov([df['mkr'][0+i:120+i],df['SMB'][0+i:120+i],df['HML'][0+i:120+i]]
)
    for j,stock in enumerate(portfolios_names):
        model = sm.OLS(train[stock], sm.add_constant(train[['mkr', 'SMB', 'HML']].v
alues))
        result = model.fit()
        B[j] = result.params.tolist()[1:4]
        res[j] = np.array(result.resid)
    temp= np.dot(np.dot(B,f_cov),B.T)
    sigma_0 = np.dot(res,res.T) / len(train)
    estimation_cov_3 = temp + np.diag(np.diag(sigma_0))
    Cov3.append(estimation_cov_3)
```

9.5 Experiment With Bandit Learning

9.5.1 Define Functions

```
# Define the Singular Value Decomposition
def SSVD(cov):
    u, s, vh = np.linalg.svd(cov, full_matrices=False)
    pc = pd.DataFrame(StandardScaler().fit_transform(u))
    lambdas = [s[i]/(sum(u[:,i]))**2 for i in range(len(s))]
    return pc, lambdas
# Define Sharp Ratio, UCB1, Softmax, Annealing Epsilon Greedy Function
# Define Orthogonal Bandit and Naive Bandit function
def sharp(pc_weight, ret, lamb):
    ratio = np.dot(pc_weight,ret)/((lamb)**0.5)
    return ratio

def UCB(name, time, pull_times):
    pc_weight = H[name]
    ret = ret_data.iloc[time,:].tolist()
    ratio = sharp(pc_weight,ret,lambdas[name])
    return ratio+((2*np.log(time+tau))/(tau + pull_times[name]))**0.5


def multi_bandit_ucb(pc, ret_data, tau, l):
    names = list(pc.columns)
    name1 = names[:l]
    name2 = names[l:]
    pull_times = dict(zip(names, np.zeros(len(names))))
    choice = []
    for time in range(len(ret_data)):
        scores = np.array([UCB(name,time, pull_times) for name in names])
        score1 = scores[:l]
        score2 = scores[l:]
        max_score_arm1 = score1.argmax()
        max_score_arm2 = score2.argmax()
        choice.append([max_score_arm1,max_score_arm2+l])
        pull_times[names[max_score_arm1]] += 1
        pull_times[names[max_score_arm2]] += 1
    return choice

def softmax(name, time, temp):
    names = H.columns.tolist()
    pc_weight = H[name]
    ret = ret_data.iloc[time,:].tolist()
    ratio = sharp(pc_weight,ret,lambdas[name])
    ratio_sum = np.array([sharp(H[name],ret,lambdas[name]) for name in names])
    return np.exp(ratio/temp)/(np.exp(ratio_sum/temp).sum())


def multi_bandit_softmax(pc, ret_data, l, temp):
```

```
    names = list(pc.columns)
    name1 = names[:l]
    name2 = names[l:]
    choice = []
    for time in range(len(ret_data)):
        scores = np.array([softmax(name, time, temp) for name in names])
        score1 = scores[:l]
        score2 = scores[l:]
        max_score_arm1 = np.random.choice(name1, 1, list(score1/score1.sum()))
        max_score_arm2 = np.random.choice(name2, 1, list(score2/score2.sum()))
        choice.append([max_score_arm1,max_score_arm2+l])
    return choice

def multi_bandit_annealing_eps_greedy(pc, ret_data, l):
    names = list(pc.columns)
    name1 = names[:l]
    name2 = names[l:]
    choice = []


    for time in range(len(ret_data)):
        ret = ret_data.iloc[time,:].tolist()
        ratio_sum = np.array([sharp(pc[name],ret,lambdas[name]) for name in names])

        eps = 1 / np.log(time+1 + 0.0001)
        u1 = np.random.rand()
    if u1 < eps:
        arm1 = np.random.randint(0,len(name1))
    else:
        arm1 = ratio_sum[:l].argmax()

    u2 = np.random.rand()
    if u2 < eps:
        arm2 = np.random.randint(len(name1), len(names))
    else:
        arm2 = ratio_sum[l:].argmax()

    choice.append([arm1, arm2])
    return choice

def naive_bandit_ucb(pc, ret_data, tau, l):
    names = list(pc.columns)
    pull_times = dict(zip(names, np.zeros(len(names))))
    choice = []
    for time in range(len(ret_data)):
        scores = np.array([UCB(name,time,pull_times) for name in names])
        max_score_arm = scores.argmax()
        choice.append([max_score_arm])
        pull_times[names[max_score_arm]] += 1
    return choice

def naive_bandit_softmax(pc, ret_data, l, temp):
    names = list(pc.columns)
    choice = []
            for time in range(len(ret_data)):
                scores = np.array([softmax(name, time, temp) for name in names])
                max_score_arm = np.random.choice(names, 1, scores.tolist())
                choice.append(max_score_arm)
            return choice

        def naive_bandit_annealing_eps_greedy(pc, ret_data, l):
            names = list(pc.columns)
            choice = []

            for time in range(len(ret_data)):
                ret = ret_data.iloc[time,:].tolist()
```

```
                ratio_sum = np.array([sharp(pc[name],ret,lambdas[name]) for name in
    names])
                eps = 1/np.log(time+1+0.0001)
                u = np.random.rand()
                if u < eps:
                    arm = np.random.randint(0,len(names))
                else:
                    arm = ratio_sum[:1].argmax()
                choice.append(arm)
            return choice
```

9.5.2 Exploration-Exploitation

```
# load data(return, covariance)
# Be careful to choose FF48 or FF 100
ret_data = pd.read_csv('FF100 Result/Mean result for FF100.csv',index_col=0)
ret_data = pd.read_csv('FF48 Result/Mean result for FF48.csv',index_col=0)
# Do SVD on covariance matrix based on different methods
# Creating the orthognal portfolio and do profolio selection based on Multi-
Arm Bandit

for i in tqdm(range(len(Cov1))):
    cov = Cov1[i]
    l = 4
    H, lambdas = SSVD(cov)
    tau = 120
    temp = 0.1
  # OBP
    choices_ucb_1_o = multi_bandit_ucb(H, ret_data, tau, l)
    choices_soft_1_o = multi_bandit_softmax(H, ret_data, l, temp)
    choices_eps_1_o = multi_bandit_annealing_eps_greedy(H, ret_data, l)
  # NBP
    choices_ucb_1_n = naive_bandit_ucb(H, ret_data, tau, l)
    choices_soft_1_n = naive_bandit_softmax(H, ret_data, l, temp)
    choices_eps_1_n = naive_bandit_annealing_eps_greedy(H, ret_data, l)


for i in tqdm(range(len(Cov2))):
    cov = Cov2[i]
    l = 4
    H, lambdas = SSVD(cov)
    tau = 120
    temp = 0.1
    # OBP
    choices_ucb_2_o = multi_bandit_ucb(H, ret_data, tau, l)
    choices_soft_2_o = multi_bandit_softmax(H, ret_data, l, temp)
    choices_eps_2_o = multi_bandit_annealing_eps_greedy(H, ret_data, l)
    # NBP
    choices_ucb_2_n = naive_bandit_ucb(H, ret_data, tau, l)
    choices_soft_2_n = naive_bandit_softmax(H, ret_data, l, temp)
    choices_eps_2_n = naive_bandit_annealing_eps_greedy(H, ret_data, l)

for i in tqdm(range(len(Cov3))):
    cov = Cov3[i]
    l = 4
    H, lambdas = SSVD(cov)
    tau = 120
    temp = 0.1
    # OBP
    choices_ucb_3_o = multi_bandit_ucb(H, ret_data, tau, l)
    choices_soft_3_o = multi_bandit_softmax(H, ret_data, l, temp)
    choices_eps_3_o = multi_bandit_annealing_eps_greedy(H, ret_data, l)
    # NBP
    choices_ucb_3_n = naive_bandit_ucb(H, ret_data, tau, l)
    choices_soft_3_n = naive_bandit_softmax(H, ret_data, l, temp)
```

```
    choices_eps_3_n = naive_bandit_annealing_eps_greedy(H, ret_data, 1)

# Calculate the Final Weight for the OBP
def calculate_final_weight(choices, Cov):
    final_weight = []
    for time, choice in enumerate(choices):
        cov = Cov[time]
        H, lambdas = SSVD(cov)
        weight1 = H[choice[0]]
        weight2 = H[choice[1]]
        theta = lambdas[choice[0]]/(lambdas[choice[0]]+lambdas[choice[1]])
        new_weight = (1-theta)*weight1 + theta*weight2
        final_weight.append(new_weight)
    return final_weight
final_weight1_ucb_o = calculate_final_weight(choices_ucb_1_o, Cov1)
final_weight2_ucb_o = calculate_final_weight(choices_ucb_2_o, Cov2)
final_weight3_ucb_o = calculate_final_weight(choices_ucb_3_o, Cov3)
final_weight1_soft_o = calculate_final_weight(choices_soft_1_o, Cov1)
final_weight2_soft_o = calculate_final_weight(choices_soft_2_o, Cov2)
final_weight3_soft_o = calculate_final_weight(choices_soft_3_o, Cov3)
final_weight1_eps_o = calculate_final_weight(choices_eps_1_o, Cov1)
final_weight2_eps_o = calculate_final_weight(choices_eps_2_o, Cov2)
final_weight3_eps_o = calculate_final_weight(choices_eps_3_o, Cov3)

# Calculate the Final Weight for NBP
def calculate_final_weight_n(choices, Cov):
    final_weight = []
    for time, choice in enumerate(choices):
        cov = Cov[time]
        H, lambdas = SSVD(cov)
        new_weight = H[choice]
        final_weight.append(new_weight)
    return final_weight
final_weight1_ucb_n = calculate_final_weight_n(choices_ucb_1_n, Cov1)
final_weight2_ucb_n = calculate_final_weight_n(choices_ucb_2_n, Cov2)
final_weight3_ucb_n = calculate_final_weight_n(choices_ucb_3_n, Cov3)
final_weight1_soft_n = calculate_final_weight_n(choices_soft_1_n, Cov1)
final_weight2_soft_n = calculate_final_weight_n(choices_soft_2_n, Cov2)
final_weight3_soft_n = calculate_final_weight_n(choices_soft_3_n, Cov3)
final_weight1_eps_n = calculate_final_weight_n(choices_eps_1_n, Cov1)
final_weight2_eps_n = calculate_final_weight_n(choices_eps_2_n, Cov2)
final_weight3_eps_n = calculate_final_weight_n(choices_eps_3_n, Cov3)

# Given the final weights, we calculate the weighted average return
return_df = data.iloc[120:,1:]
def calculate_reward(weight_lst, return_df):
    reward = []
    for i, weight in enumerate(weight_lst):
                reward.append(np.dot(return_df.iloc[i:i+1,:],weight.values)[0])
            return reward
        reward_ucb_1_n = calculate_reward(final_weight1_ucb_n,return_df)
        reward_soft_1_n = calculate_reward(final_weight1_soft_n,return_df)
        reward_eps_1_n = calculate_reward(final_weight1_eps_n,return_df)
        reward_ucb_2_n = calculate_reward(final_weight2_ucb_n,return_df)
        reward_soft_2_n = calculate_reward(final_weight2_soft_n,return_df)
        reward_eps_2_n = calculate_reward(final_weight3_eps_n,return_df)
        reward_ucb_1_o = calculate_reward(final_weight1_ucb_o,return_df)
        reward_soft_1_o = calculate_reward(final_weight1_soft_o,return_df)
        reward_eps_1_o = calculate_reward(final_weight1_eps_o,return_df)
        reward_ucb_2_o = calculate_reward(final_weight2_ucb_o,return_df)
        reward_soft_2_o = calculate_reward(final_weight2_soft_o,return_df)
        reward_eps_2_o = calculate_reward(final_weight2_eps_o,return_df)
        # For FF100, no need to run this code
        reward_ucb_3_n = calculate_reward(final_weight3_ucb_n,return_df)
        reward_soft_3_n = calculate_reward(final_weight3_soft_n,return_df)
        reward_eps_3_n = calculate_reward(final_weight3_eps_n,return_df)
```

```
        reward_ucb_3_o = calculate_reward(final_weight3_ucb_o,return_df)
        reward_soft_3_o = calculate_reward(final_weight3_soft_o,return_df)
        reward_eps_3_o = calculate_reward(final_weight3_eps_o,return_df)
```

9.5.3 Optimization With MVP

```
import pyomo.environ as pe

solver = pe.SolverFactory('gurobi')
n = len(portfolios_names)
ret_targ = np.linspace(0,2,50)
MVP = []

for i in tqdm(range(0,360)):
    train = data.iloc[0+i:120+i,1:]
    mu = train.mean(axis = 0)
    sigma = train.std(axis = 0)
    cov = Cov1[i]
    min_std = np.zeros(ret_targ.shape)
    mu_targ = 0

    model = pe.ConcreteModel()
    model.w = pe.Var(range(n),domain = pe.Reals)
    model.var = pe.Objective(expr = sum(model.w[i] * model.w[j] * cov[i,j]
                for i in range(n) for j in range(n)),sense = pe.minimize)
    model.weights = pe.Constraint(expr = sum(model.w[i] for i in range(n)) == 1)
    model.ret = pe.Constraint(expr = sum(model.w[i] * mu[i] for i in range(n)) == m
u_targ)

    for (i,mu_targ) in enumerate(ret_targ):
        model.ret.set_value(sum(model.w[i] * mu[i] for i in range(n)) == mu_targ)
        results = solver.solve(model)
        min_std[i] = np.sqrt(model.var())

    MVP_idx = np.argmin(min_std)
    MVP.append(ret_targ[MVP_idx])

MVP2 = []

for i in tqdm(range(0,360)):
    train = data.iloc[0+i:120+i,1:]
    mu = train.mean(axis = 0)
    sigma = train.std(axis = 0)
    cov = Cov2[i]
    min_std = np.zeros(ret_targ.shape)
    mu_targ = 0

    model = pe.ConcreteModel()
    model.w = pe.Var(range(n),domain = pe.Reals)
    model.var = pe.Objective(expr = sum(model.w[i] * model.w[j] * cov[i,j]
                for i in range(n) for j in range(n)),sense = pe.minimize)
    model.weights = pe.Constraint(expr = sum(model.w[i] for i in range(n)) == 1)
    model.ret = pe.Constraint(expr = sum(model.w[i] * mu[i] for i in range(n)) == m
u_targ)

    for (i,mu_targ) in enumerate(ret_targ):
        model.ret.set_value(sum(model.w[i] * mu[i] for i in range(n)) == mu_targ)
        results = solver.solve(model)
        min_std[i] = np.sqrt(model.var())

    MVP_idx = np.argmin(min_std)
    MVP2.append(ret_targ[MVP_idx])

MVP3 = []

```

```
for i in tqdm(range(0,360)):
    train = data.iloc[0+i:120+i,1:]
    mu = train.mean(axis = 0)
    sigma = train.std(axis = 0)
    cov = Cov3[i]
    min_std = np.zeros(ret_targ.shape)
    mu_targ = 0

    model = pe.ConcreteModel()
    model.w = pe.Var(range(n),domain = pe.Reals)
    model.var = pe.Objective(expr = sum(model.w[i] * model.w[j] * cov[i,j]
                for i in range(n) for j in range(n)),sense = pe.minimize)
    model.weights = pe.Constraint(expr = sum(model.w[i] for i in range(n)) == 1)
    model.ret = pe.Constraint(expr = sum(model.w[i] * mu[i] for i in range(n)) == m
u_targ)

    for (i,mu_targ) in enumerate(ret_targ):
        model.ret.set_value(sum(model.w[i] * mu[i] for i in range(n)) == mu_targ)
        results = solver.solve(model)
        min_std[i] = np.sqrt(model.var())

    MVP_idx = np.argmin(min_std)
    MVP3.append(ret_targ[MVP_idx])
```

9.5.4 Benchmark of EW Portfolio

```
start_ew_cw = list(df['Unnamed: 0']).index(199001)
end_ew_cw = list(df['Unnamed: 0']).index(201912)
ret_ew = df.iloc[start_ew_cw:end_ew_cw+1,]
ret_ew = ret_ew.reset_index(drop = True)
ret_ew = ret_ew.rename(columns = {ret_ew.columns[0]:"Date"})
stocks = ret_ew.keys()[1:]
```

9.5.5 Calculate Cumulative Wealth

```
# calculate the cumulative wealth
cw_ucb_o_cov1 =  np.cumsum(np.array(reward_ucb_1_o)) # list to array(360,)
cw_soft_o_cov1 =  np.cumsum(np.array(reward_soft_1_o))
cw_eps_o_cov1 =  np.cumsum(np.array(reward_eps_1_o))
cw_ucb_o_cov2 =  np.cumsum(np.array(reward_ucb_2_o)) # list to array(360,)
cw_soft_o_cov2 =  np.cumsum(np.array(reward_soft_2_o))
cw_eps_o_cov2 =  np.cumsum(np.array(reward_eps_2_o))
cw_ucb_o_cov3 =  np.cumsum(np.array(reward_ucb_3_o)) # list to array(360,)
cw_soft_o_cov3 = np.cumsum(np.array(reward_soft_3_o))
cw_eps_o_cov3 =  np.cumsum(np.array(reward_eps_3_o))
cw_ucb_n_cov1 =  np.cumsum(np.array(reward_ucb_1_n)) # list to array(360,)
cw_soft_n_cov1 = np.cumsum(np.array(reward_soft_1_n))
cw_eps_n_cov1 =  np.cumsum(np.array(reward_eps_1_n))
cw_ucb_n_cov2 =  np.cumsum(np.array(reward_ucb_2_n)) # list to array(360,)
cw_soft_n_cov2 =  np.cumsum(np.array(reward_soft_2_n))
cw_eps_n_cov2 =  np.cumsum(np.array(reward_eps_2_n))
cw_ucb_n_cov3 =  np.cumsum(np.array(reward_ucb_3_n)) # list to array(360,)
cw_soft_n_cov3 =  np.cumsum(np.array(reward_soft_3_n))
cw_eps_n_cov3 =  np.cumsum(np.array(reward_eps_3_n))
cw_MVP_cov1 =  np.cumsum(np.array(MVP))
cw_MVP_cov2 =  np.cumsum(np.array(MVP2))
cw_MVP_cov3 =  np.cumsum(np.array(MVP3_))

cw_ew = [0]*len(ret_ew['Date'])
cwi = 1

for date in range(len(ret_ew['Date'])):
    ret_rate_aday = (weight_ew@tmp.iloc[date])/100
    print("ret_rate_aday,",ret_rate_aday)
```

```
    cwi = cwi*(ret_rate_aday+1)
    print("cwi:",cwi)
    cw_ew[date] = cwi

# Save all the cw into csv file
cw_obp_100_lstm_cov1_arr = np.vstack([cw_ucb_o_cov1,cw_soft_o_cov1,cw_eps_o_cov1])
# array(3,360)
cw_obp_100_lstm_cov1 = pd.DataFrame(cw_obp_100_lstm_cov1_arr.T,columns = ['ucb','so
ft','eps'])
cw_obp_100_lstm_cov1.to_csv('cw_obp_100_lstm_cov1.csv',index=False)
cw_nbp_100_lstm_cov1_arr = np.vstack([cw_ucb_n_cov1,cw_soft_n_cov1,cw_eps_n_cov1])
# array(3,360)
cw_nbp_100_lstm_cov1 = pd.DataFrame(cw_obp_100_lstm_cov1_arr.T,columns = ['ucb','so
ft','eps'])
cw_nbp_100_lstm_cov1.to_csv('cw_nbp_100_lstm_cov1.csv',index=False)

cw_obp_100_lstm_cov2_arr = np.vstack([cw_ucb_o_cov2,cw_soft_o_cov2,cw_eps_o_cov2])
# array(3,360)
cw_obp_100_lstm_cov2 = pd.DataFrame(cw_obp_100_lstm_cov2_arr.T,columns = ['ucb','so
ft','eps'])
cw_obp_100_lstm_cov2.to_csv('cw_obp_100_lstm_cov2.csv',index=False)
cw_nbp_100_lstm_cov2_arr = np.vstack([cw_ucb_o_cov2,cw_soft_o_cov2,cw_eps_o_cov2])
# array(3,360)
cw_nbp_100_lstm_cov2 = pd.DataFrame(cw_obp_100_lstm_cov1_arr.T,columns = ['ucb','so
ft','eps'])
cw_nbp_100_lstm_cov2.to_csv('cw_nbp_100_lstm_cov2.csv',index=False)
# For FF 100 data set no need to run this code
cw_obp_100_lstm_cov3_arr = np.vstack([cw_ucb_o_cov3,cw_soft_o_cov3,cw_eps_o_cov3])
# array(3,360)
cw_obp_100_lstm_cov3 = pd.DataFrame(cw_obp_100_lstm_cov3_arr.T,columns = ['ucb','so
ft','eps'])
cw_obp_100_lstm_cov3.to_csv('cw_obp_100_lstm_cov3.csv',index=False)
cw_nbp_100_lstm_cov3_arr = np.vstack([cw_ucb_n_cov1,cw_soft_n_cov1,cw_eps_n_cov1])
# array(3,360)
cw_nbp_100_lstm_cov3 = pd.DataFrame(cw_nbp_100_lstm_cov1_arr.T,columns = ['ucb','so
ft','eps'])
cw_nbp_100_lstm_cov3.to_csv('cw_nbp_100_lstm_cov3.csv',index=False)

# Save file for MVP
cw_MVP_arr = np.vstack([cw_MVP_cov1,cw_MVP_cov2,cw_MVP_cov3])
cw_MVP = pd.DataFrame(cw_MVP_arr.T,columns = ['cov1','cov2','cov3'])
cw_MVP.to_csv('cw_MVP.csv',index = False)

# Save file for EW
cw_ew = pd.DataFrame(cw_ew,columns = ['ew'])
cw_ew.to_csv('cw_ew.csv',index = False)
```

9.6 Visualization

```
files = os.listdir()[1:]
files.sort()
result = pd.DataFrame()

for f in files:
    if '.csv' in f:
        temp = pd.read_csv(f)
    if f != 'ew.csv':
        temp = temp.add_suffix('_'+f[:-4])
    result = pd.concat([result, temp], axis = 1, sort = False)

tmp = result.applymap(lambda x: x/100+1)
tmp['Date'] = pd.date_range('19900101',freq = 'M',periods = 360)

tmp_melt = tmp.melt(id_vars = 'Date', var_name = 'Method', value_name = 'Cumulative
 Wealth')
```

```
fig_price = px.line(tmp_melt, x = 'Date', y = 'Cumulative Wealth', color = 'Method', height = 600)
fig_price.update_layout(xaxis_title = 'Back Testing of FF100')
```

9.7 Web Crawling

```
def save_sp500_tickers():
    resp = requests.get('https://en.wikipedia.org/wiki/List_of_S%26P_500_companies')

    soup = bs.BeautifulSoup(resp.text, "lxml")
    table = soup.find('table', {'class': 'wikitable sortable'})

    tickers = []

    for row in table.findAll('tr')[1:]:
        ticker = row.findAll('td')[0].text.strip()
        tickers.append(ticker)

    with open("sp500tickers.pickle", "wb") as f:
        pickle.dump(tickers, f)
        print(tickers)

    return tickers


save_sp500_tickers()


def get_data_from_yahoo(reload_sp500=False):
    if reload_sp500:
        tickers = save_sp500_tickers()
    else:
        with open("sp500tickers.pickle", "rb") as f:
            tickers = pickle.load(f)

    if not os.path.exists('stock_dfs'):
        os.makedirs('stock_dfs')

    start = dt.datetime(1980, 1, 1)
    end = dt.datetime(2020, 1, 1)

    for ticker in tickers:

        print(ticker)

        if not os.path.exists('stock_dfs/{}.csv'.format(ticker)):
            try:
                df = web.DataReader(ticker, 'yahoo', start, end)
                df.to_csv('stock_dfs/{}.csv'.format(ticker))
            except Exception as ex:
                print('Error:', ex)
        else:
            print('Already have {}'.format(ticker))


get_data_from_yahoo(True)
yf.pdr_override()
BRKB = web.get_data_yahoo("BRKB", start = "1980-01-01", end = "2019-12-31")
BFB = web.get_data_yahoo("BFB", start = "1980-01-01", end = "2019-12-31")
print(BRKB.head())
print(BFB.head())
```