DS-GA3001 Advanced Python For Data Science
Stock Portfolio Selection

Group 07: Yakun Wang, Ziwei Wang, Yulin Shen, Yihong Zhou, Yi Zhou

1 Introduction

A novel learning-to-trade algorithm named as CORrelation-driven Non-parametric learning strategy(*CORN*) was proposed to optimize trading strategy. In this project, we aim to put the proposed *CORN* algorithm into production and then apply the advanced techniques to improve the program efficiency.

1.1 *CORN*-Algorithm

CORN is mainly inspired by the idea of exploiting **statistical correlations** between the **market windows** in the historical stock market, and also driven by the exploration of applying powerful non-parametric learning techniques to effectively optimize the portfolio.[1]

Suppose we have m stocks, then $\boldsymbol{x_t} = (x_{(t,1)}, ..., x_{(t,m)})$ stands for the *price relative vector* for m stocks on the t-th trading day. $x_{(t,i)}$ equals the closing price of stock i on the t-th day divided by that of previous day.

$$x_{(t,i)} = \frac{P_{(t,i)}}{P_{(t-1,i)}} \tag{1}$$

Market window, denoted as **w**, needs to be pre-defined before running *CORN*. If we are interested in m stocks with market window of w from today, then the latest market movement can be represented as $\boldsymbol{(x_{t-w}, ..., x_{t-1})}$, or $\boldsymbol{X_{t-w}^{t-1}}$. We use New York Stock Exchange(NYSE) dataset[1]that contains daily price relatives of 36 stocks for a 22-year period to check the functionality of baseline model.

Output is a portfolio $b_t(w, \rho) = (b_{(t,1)}, ..., b_{(t,m)})$, specifying allocation of wealth among m stocks. Therefore, $\boldsymbol{b_t(w, \rho) \cdot x_t}$ is the daily relative return achieved by an expert that *learns* through *CORN*.

1.2 Expert's portfolio–$b_t(w, \rho)$

Expert is a major component of *CORN*, as it updates daily investment allocation to maximize up-to-day wealth return. A *correlation-similar* set, $C_t(w, \rho)$[2], is declared to store certain historical relative prices, whose correlations between $\boldsymbol{X_{i-w}^{i-1}}$, and the latest market movement $\boldsymbol{X_{t-w}^{t-1}}$, are greater than or equal to the pre-set threshold ρ:

$$C_t(w, \rho) = \{w < i < t-1, \frac{Cov(\boldsymbol{X_{i-w}^{i-1}}, \boldsymbol{X_{t-w}^{t-1}})}{std(\boldsymbol{X_{i-w}^{i-1}})std(\boldsymbol{X_{t-w}^{t-1}})} >= \rho\} \tag{2}$$

Once $C_t(w, \rho)$ is calculated at the beginning of the t-th trading day, corresponding $b_t(w, \rho)$, known as expert's portfolio for i-th day, is then computed by solving an optimization problem(Line-11 in Algorithm 1).

Algorithm 1 *CORN* Learning Procedure

1: **procedure** $CORN(t, X_1^{t-1}, w, \rho)$
2: $C_t(w, p) = \emptyset$ ▷ Initialize correlation-similar set
3: **if** t <= w + 1 **then:**
4: **return** $b_t(w, \rho) = (\frac{1}{m}, ..., \frac{1}{m})$
5: **for** i = w + 1 **to** t − 1 **:** ▷ Expert's learning progress below
6: **if corrcoef**$(X_{i-w}^{i-1}, X_{t-w}^{t-1}) >= \rho$ **then:** ▷ Compute correlation matrix
7: $C_t(w, \rho) = C_t(w, \rho) \cup \{i\}$
8: **if** $C_t(w, \rho) == \emptyset$ **then:**
9: **return** $b_t(w, \rho) = (\frac{1}{m}, ..., \frac{1}{m})$
10: **else:** ▷ Solve optimization for optimal portfolio
11: **return** $b_t(w, \rho) = \text{argmax}_{b \in \Delta m} \prod_{i \in C_t(w,\rho)} (b \cdot x_i)$ ▷ Expert's portfolio

[1]Data can be downloaded from GitHub
[2]Please see Graph 2 in appendix for visualized details

CORN allows **multiple** experts to propose independent portfolio suggestions at the same time. Experts differ depends on pre-set market window w and threshold ρ. Combining all $b_t(w, \rho)$ produces the final portfolio for i-th day $\boldsymbol{b_t}$:

$$\boldsymbol{b_t} = \frac{\sum_{w,p} q(w,\rho) s_{t-1}(w,\rho) b_t(w,\rho)}{\sum_{w,p} q(w,\rho) s_{t-1}(w,\rho)} \tag{3}$$

NOTE: $s_{t-1}(w, \rho)$ is the historical performance for an expert and $q(w, \rho)$ is a probability function.

2 General Program Structure and Benchmark

There are two versions of implementation for the *CORN* algorithm. One is *CORN*-U, which combines the experts' portfolios with even weights $q(w, \rho)$. The other is *CORN*-K, which only puts weight on the top K experts with the most wealth every day. Both implementations have their experts learning independently. Therefore, we could encapsulate all expert-related executions into a class, and make the objects sortable through the help of class methods __eq__ and __lt__. Besides, the two versions are realized simultaneously and distinguished through keyworded variable length of arguments at input. This high-level encapsulation makes our learning process easy to be paralleled later. See Figure 1 for the encapsulated general program structure.

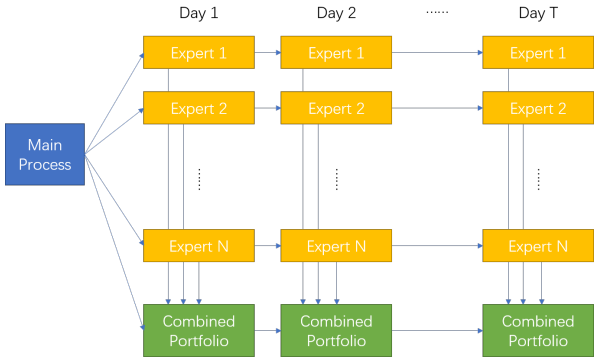


Figure 1: Program Structure Visualization

Using the above structure as a benchmark, we first tried to optimize through techniques like **cython**. It takes approximately **10 minutes** to run the entire program. However, the use of cython has nearly no effect. Then, the line profiling is used to find the real bottlenecks of our program. It shows that **99.9%** of running time is spent on experts' learning progress. Inside the function *learning*, computing correlation matrix contributes to **75.0%** of total time spent and solving optimization problem takes another **23.6%** of time, so it is clear that our optimization should focus on the three most time-consuming parts.

3 Methodology

Based on the design of *CORN* algorithm and profiling result, we have three major aspects to optimize over the benchmark.

- **Expert's Optimal Portfolio**: reformulate it into a convex optimization problem and use the c-based package to find out the optimal point.
- **Correlation Matrix**: apply Numba to speed up computation.
- **Multiple Experts' learning progress**: a *for-loop* to initiate individual expert's learning one by one can be replaced by multiprocessing, as each learning process is independent of the others.

3.1 Reformulate Optimization Problem

Each expert learns the optimal portfolio by following the similar idea of the classic BCRP strategy. This kind of optimization problem could be solved through the general solvers provided by **Scipy**, such as SLSQP. However, there exists a trade-off: SLSQP will be less fast and less robust compared to algorithms offered by specialized optimization package, which are specifically designed for convex problems.

The proposed formula is an ill-conditioned convex optimization problem because cumulative product over a large number of elements might overflow easily. To fit the problem into an existing convex

optimization framework and eliminate possible numerical computing dilemmas, we reformulate the proposed formula to a new form via logarithm transformation:

$$b_t(w,\rho) = \underset{b\in\triangle_m}{\operatorname{argmax}} \prod_{i\in C_t(w,\rho)} (b\cdot x_i)$$

$$\log(b_t(w,\rho)) = \underset{b\in\triangle_m}{\operatorname{argmax}} \sum_{i\in C_t(w,\rho)} [\log(b\cdot x_i)]$$

As daily relative income, $b \cdot x_i$ is supposed to be naturally positive. In this case, no additional constraint is needed to guarantee that $b \cdot x_i$ is positive.

More importantly, this new formula follows the disciplined convex programming (DCP) rules. Therefore, it is solvable by specially designed convex programming solvers such as CVXPY. **Maximize(concave)** and the constraint is affine. We use the C-written SCS solver from CVXPY and compare its performance with the SLSQP solver from **Scipy**.

3.2 Accelerate Correlation Matrix Computation

An intuitive practice is to use the *corrcoef()* function from numpy. However, it returns a matrix of correlation coefficients between every two elements. What we actually want is a series of correlation coefficients between moving historical time windows $\boldsymbol{X_{i-w}^{i-1}}$ and a fixed, current time window $\boldsymbol{X_{t-w}^{t-1}}$. Therefore, *corrcoef()* leads to a waste of source and slows down the entire program.

Alternatively, we write an optimized function that flattens the matrix to a long vector, computes the fixed standard error first and then iteratively computes the covariance between moving vector and fixed vector. All Numpy functions used here are supported by Numba, so we use the JIT decorator with **nopython mode** from Numba to accelerate the program. Results comparisons are summarized in later parts.

3.3 Multiprocess Experts' Learning Progress

We first parallel the step of updating experts' portfolios in each day by multiprocessing, and then collect their results at the end of the day, as each expert's portfolio is independent. While we can get the same portfolio result as the benchmark, running time actually increases. The reason might be that multiprocessing is expensive with high overhead and building pooling every day is costly over the benefit brought by multiprocessing.

However, after reconsidering the goal of optimization, we find that compared with sampling cost of waiting for closing price relatives each day, computing cost of *CORN* algorithm on online-portfolio selection is negligible. Another way of applying such algorithm is data backtesting. We need to detect the effectiveness of the algorithm and adjust model parameters by historical data. Under such situation, since historical data is available for all experts and expert learning process is also independent of the main process, we can only pool once, let expert learn through and then feedback their all historical portfolios and wealth history to the main process. Then the main process incorporates exports' portfolios and re-adjusts its final portfolio from time 1 to T entirely. When T gets large, we can separate T by several chunks, do parallel computing for each chunk, collect data from experts and update its own portfolio, then turn to the next iteration.

We also implemented MPI to optimize, and it lessens running time. But optimization solver is stable when we use MPI, and the result cannot compete with the pool.

4 Results and Discussion

4.1 Optimization

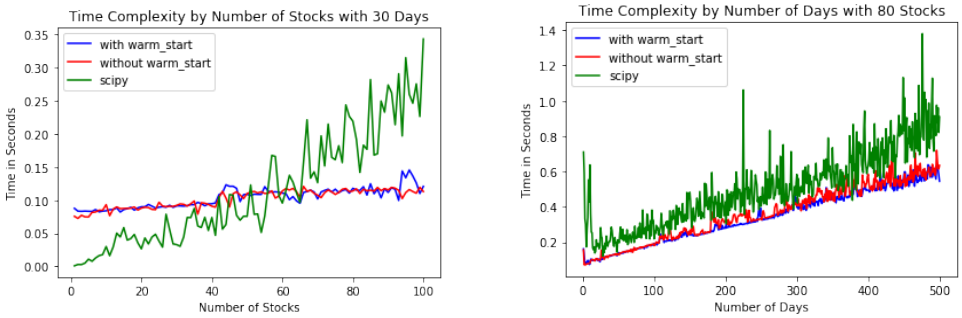
The two line graphs in 4.1 shows the time performance of Scipy and Cvxpy solvers when increasing the number of stocks or the number of days in optimization. The green line represents the time consumed by Scipy package, the blue and red lines are for Cvxpy package with or without warm start. The warm start option, a parameter in the function, might use the previous solution as an initial point or reuse cached matrix factorizations for later calculations. So from that point, we could assume the warm start should increase the calculation speed.

The left graph is the time performance of different packages based on the changing of the number of stocks. We stabilize the number of days as 30 in the problem, and gradually increases the number of stocks from 1 to 100 to see how the solver efficiency varies. The graph result shows the time rise for Cvxpy is smooth and little, but the time consumption for Scipy grows almost linearly. Therefore, in this case, the time complexity for Cvxpy is close to O(1), and for Spicy is O(n). Additionally, Scipy performance fluctuates very much. This package's solver is not as stable as the ones of Cvxpy. On the other hand, when looking carefully into the Cvxpy performance lines, we figured out the warm

start option shows no significant improvements. This is because the data we used for performance testing is independent every day so that the KKT matrix used by warm start alters. Then, there are no reusable previous solutions could speed up the calculation.

The right graph is the time performance based on the changing of the number of days. In this case, we stabilize the number of stocks as 80 and make the number of days from 1 to 500. The time duration for both two packages increase proportionally, and they are both in O(n). Nonetheless, Cvxpy overwhelmingly beats Scipy this time.

Overall, the specific convex optimization solver is much better than a general optimization solver. Cvxpy is fast and reliable, and particularly in our problem, it is a good solution.



4.2 Correlation Matrix

Experiment uses the first 500 days from the NYSE dataset as input and sets time window $w = 5$, correlation threshold $\rho = 0.5$. Average time spent is calculated on 7 runs, 100 loops each. See Table 1 for a summary of performance comparison.

Approach Used	Average Time	Std
numpy corrcoef()	30.1 ms	260 μs
scipy.stats pearsonr()	23.1 ms	511 μs
vectorized, scipy.stats pearsonr() with @jit decorator (object mode)	19.9 ms	393 μs
vectorized, self-written code with @jit decorator (nopython mode)	1.08 ms	137 μs
vectorized, self-written code with @jit decorator (object mode) and function signature	1.08 ms	137 μs

Table 1: Results Comparison of Different Correlation Matrix Computing Approaches

4.3 Multiprocessing & Parallel Processing

With the same dataset, we ran the *CORN*U algorithm with $w = 5, \rho = 0.4$. See Table 2 for a summary.Apparently, with one-time multiprocessing has the best performance.

Approach Used	Time Spent
without acceleration	82.2 s
with expensive multiprocessing	1737.3 s
with MPI	49.8 s
with one-time multiprocessing	20.0 s

Table 2: Results Comparison of Different Parallelizing Approaches

4.4 Conclusion

Finally, we run the optimized model and the benchmark separately on a larger dataset, which includes the first 1000 days of the NYSE relative prices, and use a larger time window $w = 7, \rho = 0.4$. The optimized one takes only 55.8 seconds, which is more than 10 times shorter compared with the 636.0 seconds spent by the original implementation.

We also visualize the performance of *CORN* compared with other individual stocks in appendix for reference.[3]

[3]Figure 3 for the individual stock values and our algorithm performance.

References

[1] Bin Li, Dingjiang Huang, and Steven CH Hoi. Corn: Correlation-driven nonparametric learning approach for portfolio selection–an online appendix. *arXiv preprint arXiv:1306.1378*, 2013.

Appendix

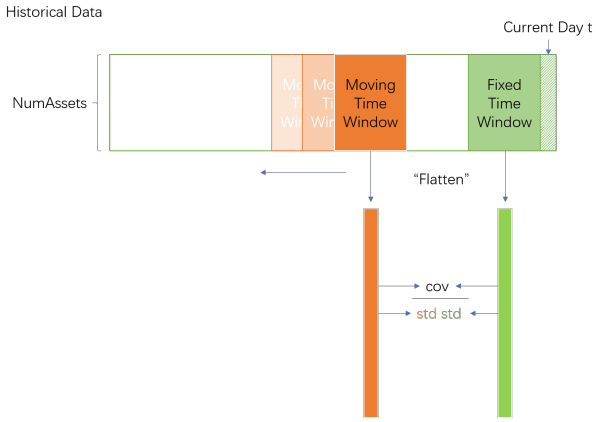


Figure 2: Correlation Coefficients between moving Historical Time Window $\boldsymbol{X_{i-w}^{i-1}}$ and a Fixed, Current Time Window $\boldsymbol{X_{t-w}^{t-1}}$

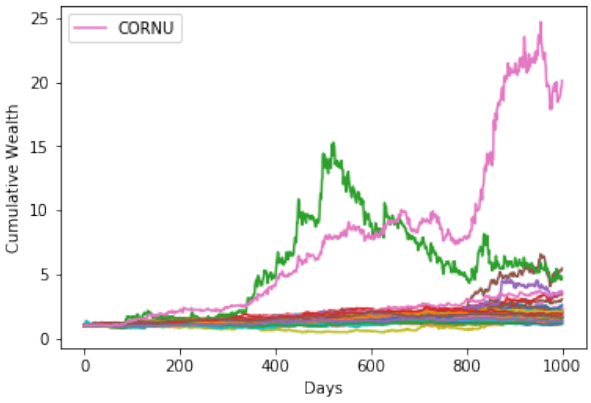


Figure 3: Algorithm Performance vs Individual Stocks