CSC 490 Directed Studies: Online Portfolio Problem

Jasper Halvorson

Abstract

This report is a comprehensive summary of the ideas explored during a directed studies course focused on the online portfolio selection problem. A brief background on the problem is presented, immediately followed by the actions taken to build the necessary theoretical knowledge. The remaining sections cover algorithm implementations and discoveries made through simulation of various existing and novel online portfolio algorithms on historical stock data.

1 Introduction

Online learning is a branch of machine learning concerned with making predictions on sequential data, and updating decision mechanisms in response to observed outcomes with the goal of minimizing the regret relative to the optimal choice in hindsight. Naturally, this applies to financial markets as we can invest in stocks and evaluate our decision by comparing our total return to that of a benchmark portfolio. In the general online portfolio selection problem, we choose an allocation of our total wealth across n assets in each round, observe the outcome of our decision, and update our distribution for the next round.

In this report, we implement and empirically evaluate several online portfolio algorithms including Online Gradient Descent (OGD), Online Newton Step (ONS), and Exponentiated Gradient (EG). We first study the mechanics of each algorithm, their application to financial market data, and their performance relative to one another and standard benchmarks on an S&P 500 derived dataset. We then develop modifications to these baseline methods, running simulations on similar stock data with the goal of improving performance. Finally, we attempt to create an algorithm to handle an extremely volatile "penny stock" dataset, a setting in which the existing strategies break down.

2 Related Works

The two key objectives of online portfolio selection can be summarized as minimizing the regret and computational expense of the underlying selection algorithms. *Prediction Learning and Games* [CBL06] examines regret-focused prediction, providing a source for understanding theoretical methods for deriving regret bounds and analyzing algorithms. One repeated topic is online convex optimization, a field of study directly applicable to online portfolio selection and comprehensively covered in

Elad Hazan's *Introduction to Online Convex Optimization* [Haz19]. Hazan's work further formalizes foundational theory referenced in this report including definitions and regret bound proofs of the OGD and ONS algorithms which we apply to the online portfolio problem.

Cover's 1991 *Universal Portfolios* [Cov91] provides a solution concept to the online portfolio problem, achieving asymptotically optimal performance (meets the performance of the Optimal CRP 3.1.3 over the long run). The algorithm, however, requires an expensive integration over the simplex in each round, rendering it unfeasible in practice. Van Erven's 2020 paper [vEvdHKK20] proposes an open problem to find an online portfolio selection algorithm which meets the theoretical regret guarantees of Universal Portfolios with reduced computational complexity. Just this year, an algorithm meeting these specifications, *Volumetric-Barrier enhanced Follow-The-Regularized-Leader*, was presented in *Efficient and near-optimal online portfolio selection* [JOG25].

3 Algorithms

3.1 Benchmark Strategies

Several baseline investment strategies exist in the space of online portfolio selection. In this report, we will compare more advanced algorithms with these baselines to evaluate their performance. In particular, we focus on the Best Stock, Uniform Buy-and-Hold, Uniform/Optimal CRPs, and Follow-The-Leader benchmark strategies. A more comprehensive overview of existing online portfolio algorithms can be found in Li and Hoi's survey [LH14].

3.1.1 Best Stock

This strategy consists of finding the single stock that had the best performance over the entire studied time period, and allocating all of the investment to it at time 0. In particular, let the cumulative return of stock i over T days be

$$R_{T,i} \;=\; \prod_{t=1}^{T} r_{t,i},$$

and define

$$i^\star \;=\; \arg\max_{1\le i\le n} R_{T,i}.$$

Here, we invest all wealth in stock $i^\star$ at time 0, and hold it until time T. Note that since the selection process relies on future information, *Best Stock* can only be studied in hindsight.

3.1.2 Uniform Buy and Hold

Similar to *Best Stock*, we buy fixed quantities of stock at time 0 and hold them until time T. If we have n stocks and our starting wealth is W, we invest $\frac{W}{n}$ in each stock.

3.1.3 Uniform/Optimal CRP

A Constantly Rebalanced Portfolio (CRP) is a portfolio that reallocates its total wealth at each investment round (1 day for our purposes) according to fixed proportions (that sum to 1) associated with each stock. A Uniform CRP commits $\frac{W}{n}$ to each stock in each round, t, regardless of how much stock $\frac{W}{n}$ is able to buy according to the prices in round t. An Optimal CRP chooses the most profitable set of proportions in hindsight and uses them for its rebalancing policy. Concretely, let $\mathbf{x}_t$ be the (fixed) weight vector used in every round, with

$$\mathbf{x}_t = \mathbf{x} \in \mathcal{K} \quad \text{and} \quad \mathcal{K} = \Big\{ \mathbf{x} \in \mathbb{R}^n_{\geq 0} : \sum_{i=1}^{n} x_i = 1 \Big\}.$$

Letting $\mathbf{r}_t$ be the vector of returns for each stock in round t (as used in *Best Stock*), the Uniform CRP simply takes

$$\mathbf{x} = \Big(\frac{1}{n}, \ldots, \frac{1}{n}\Big)$$

in every round. The Optimal CRP, however, chooses the weights

$$\mathbf{x}^\star = \arg\max_{\mathbf{x} \in \mathcal{K}} \prod_{t=1}^{T} \langle \mathbf{x}, \mathbf{r}_t \rangle,$$

which maximize the total return over the studied time period.

3.1.4 Follow-The-Leader

Follow-The-Leader (FTL) is an online strategy which allocates all the available wealth to, in our setting, the stock that has performed the best up until the current round. Below, we describe the algorithm using notation consistent with the other baseline strategies presented.
At round t, the FTL strategy chooses

$$i_t = \arg\max_{i \in \{1,\ldots,n\}} R_{t-1,i},$$

and invests all wealth in that stock:

$$\mathbf{x}_t = e_{i_t},$$

where e_{i_t} is the standard basis vector with a 1 in the index i_t and zeros elsewhere.
Note that a more general description of the FTL algorithm can be found in Section 3.2 of *Prediction Learning and Games* [CBL06].

3.2 Online Gradient Descent

Hazan [Haz19] describes online gradient descent (OGD) as the simplest generalized algorithm that applies to online convex optimization. Algorithm 1 shows the pseudocode for the general form of OGD, taken directly from Hazan [Haz19]. At each time step $\mathbf{t}$, the algorithm chooses a point $\mathbf{x}_t$ on the convex set $\mathcal{K}$ and updates it using the scaled gradient of the cost function at $\mathbf{x}_t$. The final operation at each time step is a projection to ensure $\mathbf{x}_t$ remains on $\mathcal{K}$.

Algorithm 1: online gradient descent

Input: convex set $\mathcal{K}$, T, $x_1 \in \mathcal{K}$, step sizes $\{\eta_t\}$

for $t = 1$ **to** T **do**

  Play x_t and observe cost $f_t(x_t)$.

  Update and project:

$$y_{t+1} = x_t - \eta_t \nabla f_t(x_t)$$
$$x_{t+1} = \Pi_{\mathcal{K}}(y_{t+1})$$

end

Applying this to financial market data, we consider a scenario in which we have $\mathbf{n}$ stocks to distribute our wealth across, and at the beginning of each trading day we chose some such distribution. We can describe $\mathcal{K}$ as the convex set of weights associated with the proportion of our investment allocated to each stock and $\mathbf{T}$ as the total number of trading periods. To be concrete, we define the following:

- The feasible set is
$$\mathcal{K} = \Big\{ \mathbf{x} \in \mathbb{R}^n_{\geq 0} : \sum_{i=1}^{n} x_i = 1 \Big\}.$$
- At round $t \in \{1, \ldots, T\}$, choose a portfolio $\mathbf{x}_t = (x_{t,1}, \ldots, x_{t,n}) \in \mathcal{K}$.
- Let $S = \{s_1, \ldots, s_n\}$ denote the n stocks, and let W_t be current wealth. We invest $x_{t,i} W_t$ in s_i during round t, so the allocation is proportional to $\mathbf{x}_t$.
- Let the *price relatives* for round t be $\mathbf{r}_t = (r_{t,1}, \ldots, r_{t,n})$, where $r_{t,i} := \frac{p_{t,i}}{p_{t-1,i}}$. Each $r_{t,i}$ is today's price divided by yesterday's for stock i at time t.
- Log-loss is used as the cost function:
$$f_t(\mathbf{x}_t) \;=\; -\log\big(\langle \mathbf{x}_t, \mathbf{r}_t \rangle\big).$$
  Note that minimizing the loss corresponds to a larger wealth value since $W_{t+1} = W_t \langle \mathbf{x}_t, \mathbf{r}_t \rangle$.

Using the specifications defined on the previous page, OGD can be written in Python and applied to stock data from the Stooq API. Rather than using the raw prices, price relatives can be used to capture returns as described above.
Following Hazan's learning-rate schedule, we set

$$\eta_t = \frac{D}{G\sqrt{t+1}},$$

where G is an upper bound on the Euclidean norm of the gradients observed so far and D is an upper bound on the Euclidean diameter of $\mathcal{K}$:

$$\forall \mathbf{x}, \mathbf{y} \in \mathcal{K},\ \|\mathbf{x} - \mathbf{y}\| \leq D.$$

Since $\mathcal{K}$ is the standard simplex, its Euclidean diameter is $\sqrt{2}$; therefore, we set

$$D = \sqrt{2}.$$

Note that η_t is chosen after playing $\mathbf{x}_t$ in the current round, so no future information is used to decide the weights.

The gradient computation of OGD determines the gradient of the loss with respect to the chosen weights, $\mathbf{x}_t$. We determine this to be $-\frac{r_t}{x_t^\top r_t}$ by defining

$$u = \langle x_t, r_t \rangle = x_t^\top r_t,$$

and applying the chain rule:

$$\frac{\partial}{\partial x_t}\big(-\log u\big) = -\frac{1}{u} \cdot \frac{\partial u}{\partial x_t}.$$

Since

$$\frac{\partial}{\partial x_t}(x_t^\top r_t) = r_t,$$

we obtain

$$\nabla f_t(x_t) = -\frac{1}{\langle x_t, r_t \rangle}\, r_t = -\frac{r_t}{x_t^\top r_t}.$$

Using the learning rate η_t and the gradient computed in round t, we then take a gradient step to update the weights. This update may cause y_{t+1} (Line 21 of 1) to leave the feasible set $\mathcal{K}$, meaning that a projection back to the nearest point on the set is necessary. Concretely, we want to find a set x in $\mathcal{K}$ to minimize the Euclidean distance between x and y_{t+1} as stated below:

$$\min_{x \in \mathcal{K}} \ \|x - y_{t+1}\|_2^2 \quad .$$

We can use the CVXPY library to handle this optimization problem, resulting in a feasible set of weights x_t. Listing 7 shows the corresponding projection function which is called in Line 22 of Listing 1.

3.3 Online Newton Step

By keeping track of all past gradient information in a matrix, A_t, the *Online Newton Step* (ONS) algorithm can modify the weights in each round based on second-order information. A_t acts as a Hessian approximate through rescaling according to curvature implied by the history of previously observed gradients; therefore, ONS takes different size steps in each direction according to the consistency of their past gradients. Intuitively, if gradients have been large in a particular direction, ONS takes smaller steps as this may indicate instability. In contrast, when gradients are more consistent (smaller second-order derivative), ONS considers this information to be more reliable and will take larger steps.

Algorithm 2 taken from Hazan [Haz19] provides pseudocode which we used as a base for our Python implementation, Listing 2 of the Appendix.

Algorithm 2: online Newton step

Input: convex set $\mathcal{K}$, T, $x_1 \in \mathcal{K} \subseteq \mathbb{R}^n$, parameters $\gamma, \varepsilon > 0$, $A_0 = \varepsilon I_n$

for $t = 1$ **to** T **do**

Play x_t and observe cost $f_t(x_t)$.

Rank-1 update: $A_t = A_{t-1} + \nabla_t \nabla_t^\top$

Newton step and generalized projection:

$$y_{t+1} = x_t - \frac{1}{\gamma} A_t^{-1} \nabla_t$$

$$x_{t+1} = \Pi_{\mathcal{K}}^{A_t}(y_{t+1}) = \arg\min_{x \in \mathcal{K}} \left\{ \|y_{t+1} - x\|_{A_t}^2 \right\}$$

end

A_t is initialized as an $n \times n$ identity matrix scaled by a constant ϵ,

$$A_0 \;=\; \epsilon I_n \;=\; \begin{pmatrix} \epsilon & 0 & \cdots & 0 \\ 0 & \epsilon & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & \epsilon \end{pmatrix},$$

and is updated in each round by adding the current gradient's outer product. The weight update in each round takes a curvature-aware step in the direction of the scaled product of the inverse A_t matrix and the current gradient, followed by an A_t based projection back to the convex set, $\mathcal{K}$.
In the portfolio selection setting, we retain the same notation as described in the *Online Gradient Descent* section. Additionally, since our choice of cost function remains as log-loss, our gradient computation is also the same. For our realization of ONS, however, we must consider choices for our new parameters γ and ϵ, as well as the optimization problem presented by our A_t-norm projection

onto $\mathcal{K}$. Following the theoretically optimal choice from Hazan [Haz19], we select γ as:

$$\gamma = \tfrac{1}{2}\min\left\{\frac{1}{GD}, \alpha\right\},$$

where α is the exp-concavity constant of the log-loss cost function $f_t(x)$ which is known to be 1. Since ϵ is dependent on information from all rounds and is only used once, we choose

$$\epsilon = 0$$

rather than its theoretically optimal choice of

$$\epsilon = \frac{1}{\gamma^2 D^2}.$$

The A_t-norm projection becomes a minimization, similar to the Euclidean projection used for OGD. In this case, we solve for the weight vector $x \in \mathcal{K}$ that minimizes $(x-y)^\top A_t (x-y)$:

$$\min_{x \in \mathcal{K}} (x-y)^\top A_t (x-y).$$

We present our implementation of the projection using the cvxpy Python library in Listing 8 in the Appendix. Note that if we replace A_t with the identity matrix, I, the minimization would match that of the Euclidean projection.

3.4 Exponentiated Gradient

As mentioned in Section 2, Universal Portfolios is an online portfolio algorithm that achieves optimal regret bounds, but is computationally unfeasible due to its expensive integration over the n-dimensional simplex (where n is the number of stocks). Cesa-Bianchi and Gàbor Lugosi [CBL06] present *Exponentiated Gradient* as an efficient and simpler alternative to Universal Portfolios that achieves similar regret bounds. Reproducing their formulae in terms of our defined variables so far in Section 3, we get the following:

$$x_{i,t} = \frac{x_{i,t-1}\,\exp\!\big(\eta\,(\nabla f_t(\mathbf{x}_t))_i\big)}{\sum_{j=1}^{n} x_{j,t-1}\,\exp\!\big(\eta\,(\nabla f_t(\mathbf{x}_t))_j\big)}$$

and

$$\eta = \frac{c}{C}\sqrt{\frac{8\ln n}{T}}.$$

where we assume that c and C are positive constants such that all price relatives r_t are in the interval $[c, C]$ for $t \in (0, T)$.

We can implement the Exponentiated Gradient algorithm based on the update rule described on the previous page and by using the same loss and gradient computations as done for OGD and ONS. Since a normalization is already included in the update formula, no projection back to the simplex is needed. Further, the learning rate η is computed as

$$\eta_t = \frac{c_t}{C_t}\sqrt{\frac{8\ln n}{T}},$$

where we define c_t and C_t as the smallest and largest *previously observed* price relatives. At time t, price relatives from 0 to $t-1$ are used in η_t's computation. This definition makes the assumption that price relatives at t fall into $[c_t, C_t]$ in order to avoid using future information for predictions. With these assumptions on the price relatives, however, we cannot guarantee the same theoretical regret bounds and price relatives that fall outside $[c_t, C_t]$ may cause unstable weight updates. Listing 3 of the Appendix is our Python implementation of the Exponentiated Gradient algorithm.

3.5 Algorithm Comparison

To understand how these algorithms compare to each other and to baselines, we use 10 years (Nov 1, 2015 - Nov 1, 2025) of historical data on a group of 20 arbitrarily chosen S&P 500 stocks. First, consider online gradient descent applied to this dataset in Listing 1. We notice in Figure 1 that the theoretical learning rate choice proves to be too slow to impact the decisions; this is in part due to the gradient bound selection being chosen based only on past information (future information would compromise the integrity of the simulation). Testing on data prior to Nov 1, 2015, we determine that a learning rate scalar of 20 helps accelerate the learning rate while maintaining stable performance. While the learning rate adjustment invalidates the theoretical regret bounds on our algorithm, it delivers far better practical results on our 2015-2025 data as shown in Figure 2.

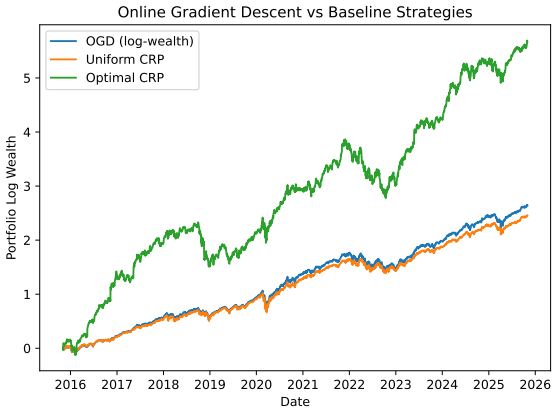

Figure 1: OGD vs Baseline CRPs

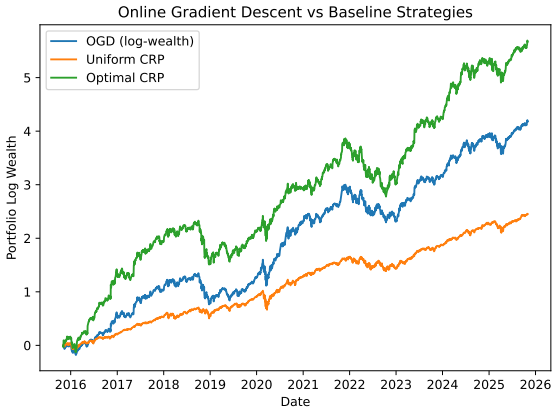

Figure 2: OGD Scaled Eta vs Baseline CRPs

Testing Online Newton Step on the same data, we observed reasonable performance and learning

speed as shown in Figure 3. No significant improvement was observed in backtesting by increasing the learning rate on ONS.

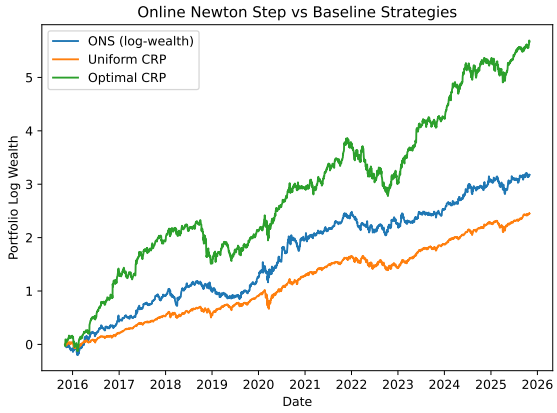

Figure 3: ONS vs Baseline CRPs

Exponentiated Gradient had the best performance of the three on this dataset, but also required scaling the learning rate as done for OGD (scalar of 100 used in this case).

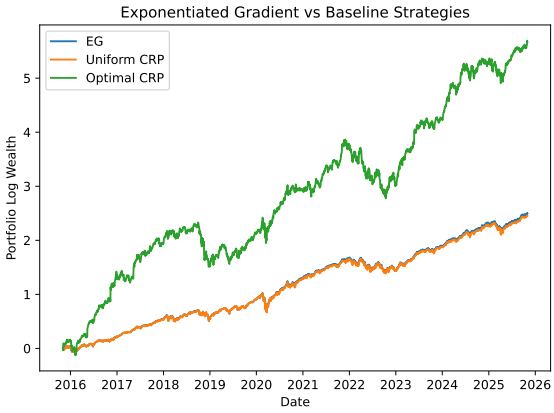

Figure 4: EG vs Baseline CRPs

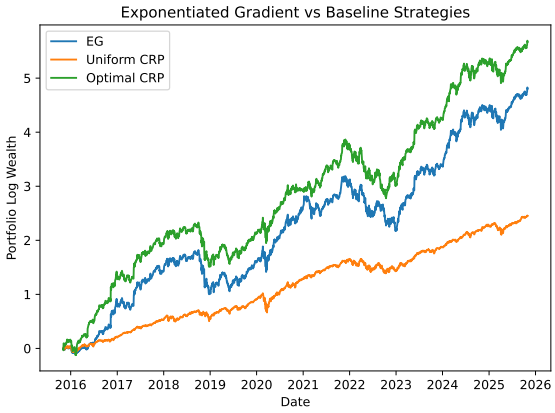

Figure 5: EG Scaled Eta vs Baseline CRPs

Next, consider plots on the same dataset, directly comparing the three algorithms (using scaled versions of EG and OGD) in Figure 6 and comparing their regret against the Optimal CRP performance in Figure 7.

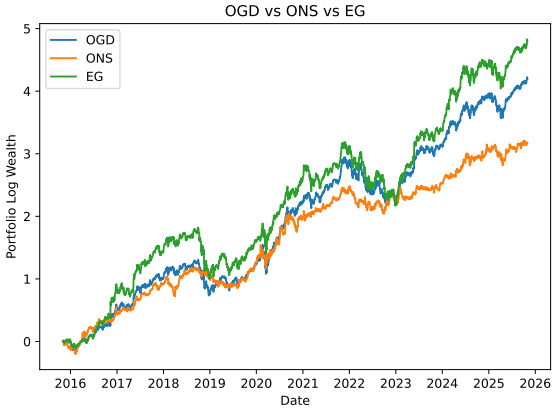

Figure 6: OGD vs ONS vs EG

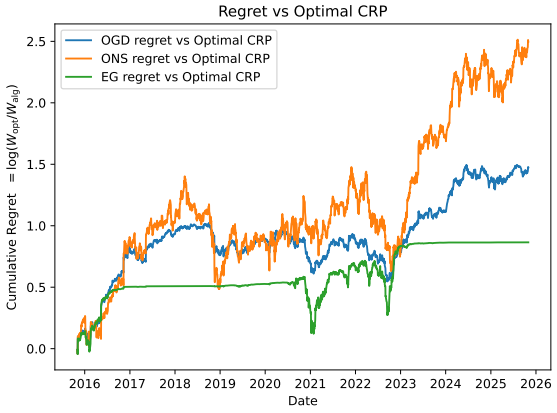

Figure 7: Algorithm Regret against Optimal CRP

In this particular scenario, EG performed the best, followed by OGD and then ONS. This ordering will, of course, not always be the case; in fact, ONS has the best theoretical regret guarantees of the three and we would usually expect it to outperform the others in sufficiently long or adversarial scenarios.

4 Exploring New Solutions

4.1 Datasets

Section 3.5 uses a 20-stock dataset we call "SP20" of high market cap stocks on the S&P 500 over the period Nov 1, 2015 - Oct 31, 2025; this dataset is again again studied in Section 4.3. Testing a specific modification concerned with grouping stocks together, Section 4.2 uses a different 20 stock dataset and an additional 40 stock dataset of S&P 500 stocks which feature more sectors (Energy, Industrial, etc.) over the same time period; we call these "Group SP20" and "Group SP40", respectively. Finally, we create a dataset with 30 penny stocks called "PENNY" and study it over the period Nov 1, 2020 - Oct 31, 2025 in Section 4.4. Note that in each case, algorithm modifications were determined through testing on time frames prior to the studied period.

4.2 Additional Experts

In this section, we aim to evaluate how the algorithms perform when learning from groups of related stocks as well as from the individual stocks. This is accomplished by grouping together stocks into pseudo-ETFs and presenting them as investable assets to the algorithms. We hypothesize that the algorithms from Sections 3.2-3.4 will improve performance with extra information available to base their weight decisions off of.

To test the potential effectiveness of this strategy, I arbitrarily select 20 S&P 500 (as of November 4th, 2025) stocks from a range of sectors. These stocks are then grouped by sector into 6 bundles, which serve as investable assets. When the algorithm allocates weight to one of the bundles, however, the weight is redistributed across the stocks of that bundle. For example, if we have stocks $s_1, \ldots, s_{20}$ and bundles $b_1, \ldots, b_6$, then an investment in bundle b_i is implemented by assigning an equal fraction of that bundle's weight to each stock contained in b_i. An additional 40-stock group was used in the same way.

4.2.1 Application to Online Gradient Descent

For the online gradient descent algorithm, as described in Section 3.2, adding pseudo-ETFs entails computing the loss, gradients, and η values using the set $\{s_1, \ldots, s_{20}, b_1, \ldots, b_6\}$, and then redistributing bundle weights to obtain the decision vector x_t for round t.

This strategy had similar results to what was observed without the bundles. Figure 8 shows how the two compare on price relative data from our studied time period.

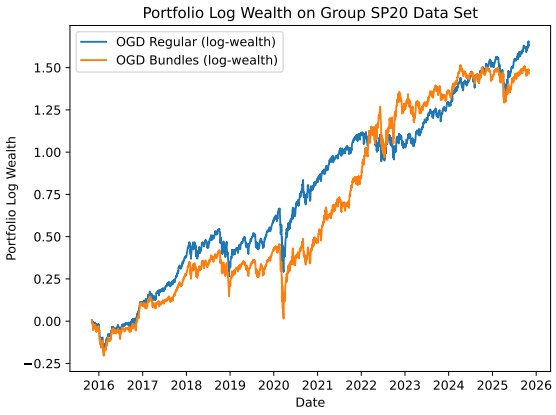

Figure 8: OGD Basic Bundling on Group SP20 Dataset

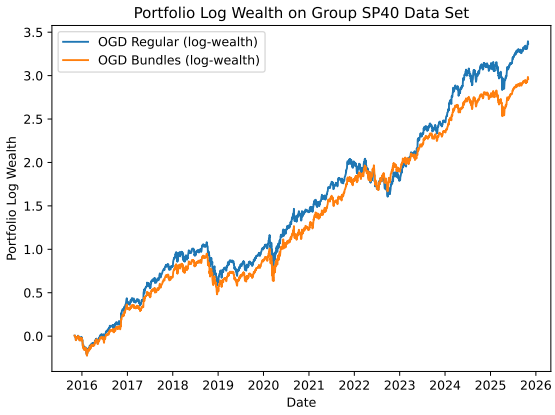

Figure 9: OGD Basic Bundling on Group SP40 Dataset

To attempt to gain more variance between the new strategy and regular OGD, the redistibution rule was then changed to allocate the entire weight assigned to the group to the best performing individual stock (at time t) of that group. This resulted in a considerably worse performance for each dataset. Since the OGD update rule expects the weight it assigns to a pseudo-ETF to be distributed according to the group averages, modifying this gives OGD inconsistent feedback which degrades its ability to stably improve its predictions. Figures 10 and 11 show the results of this algorithm compared to plain OGD on the Group SP20 and Group SP40 datasets, respectively.

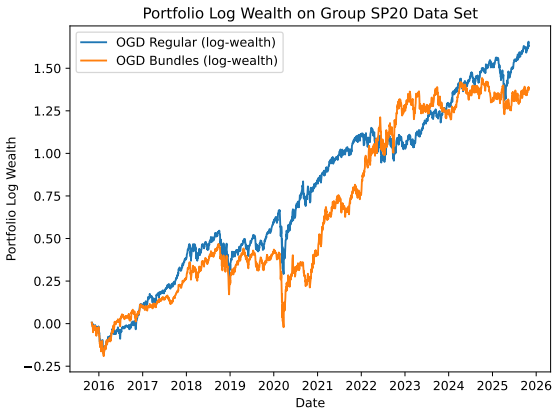

Figure 10: OGD Single-Allocation Bundling on Group SP20 Dataset

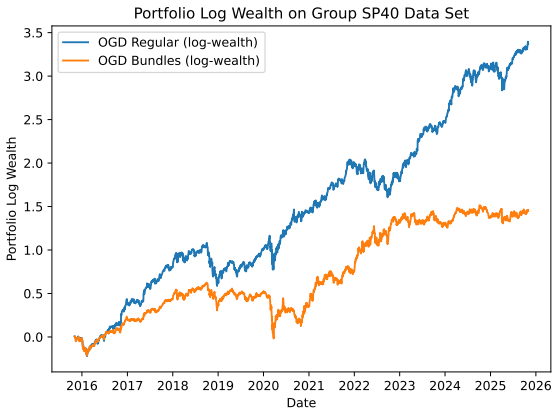

Figure 11: OGD Single-Allocation Bundling on Group SP40 Dataset

Next, rather than creating bundles by sector, we simulate 20 randomized bundling choices on data prior to Nov 1, 2015 (for Group SP20 and Group SP40, respectively) and use the best performing choices for the 2015-2025 data. The constraints on each bundling choice were that they must have a minimum of 3 elements per bundle and all bundles together span 50% to 75% of the set. We observed a great improvement with this strategy, presented in Figures 12, 13 and Table 1. This strategy preserves OGD's learning mechanics while incorporating a modification that proved successful on past data, making it perform well in cases where the future market behaviour generally follows learned patterns from the past. Note that Group SP40 performed better in general over this period, accounting for the much larger wealth increase when compared to Group SP20.

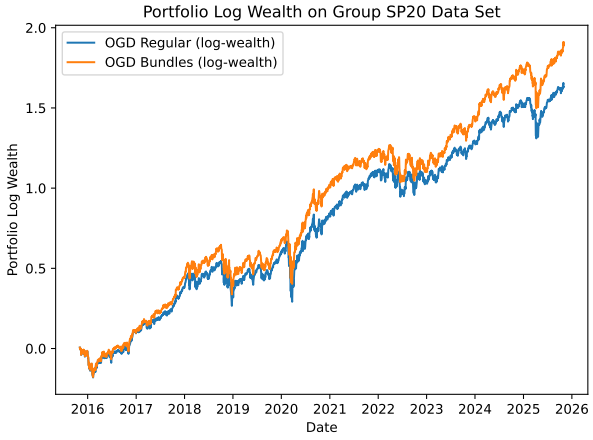

Figure 12: OGD with Good Bundling Choice on Group SP40 Dataset

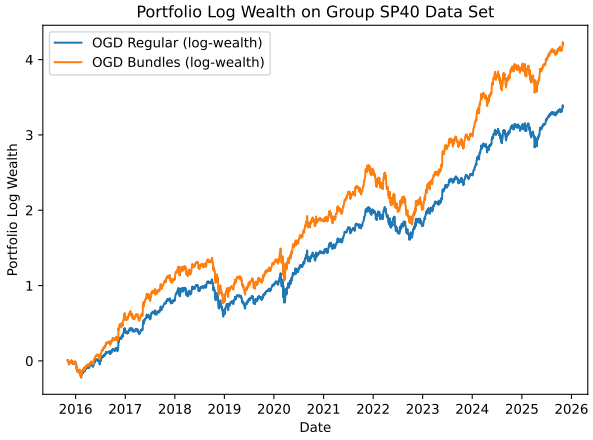

Figure 13: OGD with Good Bundling Choice on Group SP40 Dataset

Dataset	OGD Regular	OGD w/ Good Choice Bundling
Group SP20	411%	574%
Group SP40	2 800%	6 560%

Table 1: OGD Total Return Comparison with Good Bundle Choices

Overall, these experiments suggest that the addition of pseudo-ETFs to OGD can provide a significant benefit when the pseudo-ETF choices are determined from historical data patterns. This may suggest that there exists an effective bundling-based algorithm which dynamically creates and deletes pseudo-ETFs as part of its update rule. A strategy of this sort could then continuously improve its bundling choices as it observes more outcomes.

We also found that a Follow-The-Leader variant for bundle redistribution among its assets resulted in performance deterioration. This behaviour aligns with OGD's update rule struggling to consistently improve since it allocates wealth to a pseudo-ETF based on its average price change. Concretely, OGD expects the wealth to be evenly distributed to all assets contributing to the average price change of the pseudo-ETF which is not the case with this strategy.

4.3 AdaGrad Update

AdaGrad is similar to Online Newton Step with the key difference being that in uses $A_t^{-1/2}$ rather than A_t^{-1} in its update step [Cor21]. We hypothesize that there is some choice of A_t's exponent in the interval [-1, -1/2] that achieves better results on our historical stock dataset.

First, we experiment on past data to find an optimal choice of exponent, p, in [-1, -1/2] by taking 0.05 sized steps and finding which value results in the highest final wealth. Finding that -1/2 was the optimal value among these choices and that the wealth increased as p approached -1/2, we expanded p's interval to [-1, 0). Based on the returns on the past data graphed in Figure 14, p was chosen to be -0.3. Even though the global maximum appeared at -0.15, the local maximum of -0.3 was more stable and is closer to the choices of p used for AdaGrad and ONS updates. With our choice set, we then ran the same experiment on the data of interest (2015-2025) and found that -0.3 was a reasonable choice, outperforming both AdaGrad (p=-0.5) and ONS (p=-1) as shown in Figure 15

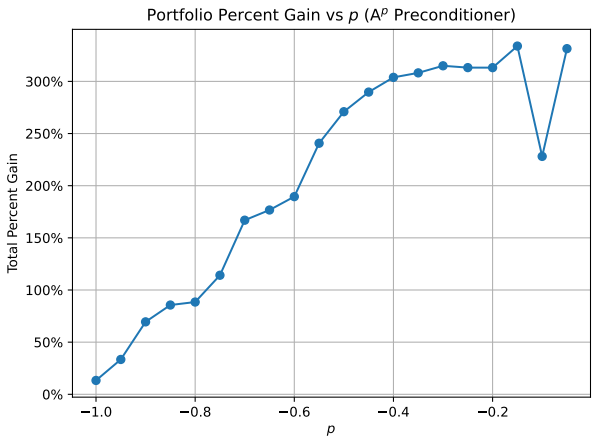

Figure 14: ONS Final Percent Gain on Past Data with Varying p

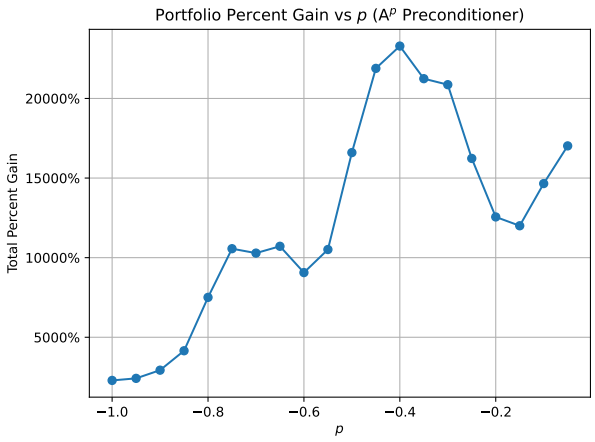

Figure 15: ONS Final Percent Gain on Study Period with Varying p

Finally, comparing the choice of p=-0.3 to regular ONS on a log-scale plot over the study period again shows a phenomenally better performance. Figure 16 shows the portfolio log wealth over time comparing these algorithms. To put this difference into perspective, Figure 2 summarizes the total percent gain and what 1000$ would become if one invested following each strategy on SP20 over the studied time period.

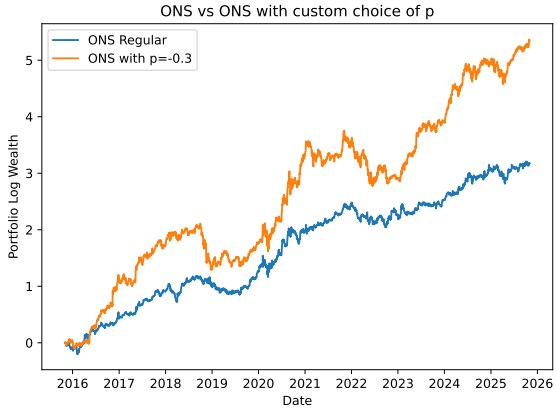

Figure 16: ONS vs ONS w/ p=-3 on SP20

Algorithm	**Percent Gain**	**Result of 1000$ Invested**
ONS Regular	2 289%	$23 890
ONS with p=-0.3	16 600%	$167 000

Table 2: ONS vs ONS w/ p=-3 on SP20

4.4 Effective Algorithm for Volatile Stocks

ONS, OGD, and EG all seem to perform poorly given volatile stocks. To attempt to mitigate these issues, we devise a variant of EG which incorporates a volatility scalar for each stock, along with different FTL algorithms to test on the volatile data.

4.4.1 Volatile Dataset Selection

We start by searching for penny stocks (stocks with prices under $5) on finviz.com, adding those with a monthly volatility percentage of 12% or higher to our dataset. Note that the monthly volatility percentage can be thought of as how much the stock fluctuates over a month. Here, 12% indicates that the stock tends to change by around 12% in a month, measured by taking daily standard deviations and converting them to a monthly change percentage. We identify 30 such stocks that also have price data available through the Stooq API. As for the time period, a 2 year (2018-2020) study period and a 5 year (2020-2025) testing period is chosen. As described in Section 4.1, we refer to this dataset as "PENNY". Figure 17 shows the performance of baseline algorithms, ONS, EG, and OGD, demonstrating the challenge of this new dataset.

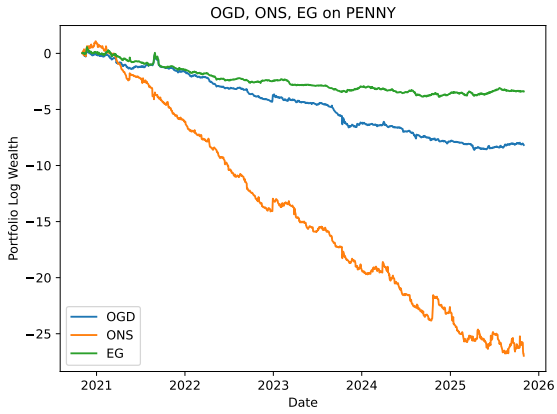

Figure 17: Standard Algorithms on PENNY

4.4.2 Volatility-Aware Exponentiated Gradient

In this section, we experiment with the exponentiated gradient algorithm by adding a way of tracking the volatility of each stock and integrating it into the update rule. To accomplish this, we define a volatility vector v of length n which holds a "current" volatility score for each stock. Exponentially weighted moving average (EWMA) is used for each entry as

$$v_{t,i} = \sqrt{(1-\lambda)\, v_{t-1,i}^2 + \lambda\, r_{t,i}^2 + \varepsilon},$$

where $0 < \lambda < 1$ and a larger λ results in heavier weight on recent prices.
Adding this to EG's update, we get the following:

$$x_{i,t} = \frac{x_{i,t-1}\ \exp\Big(\eta\ \frac{(\nabla f_t(\mathbf{x}_t))_i}{v_{i,t}+\varepsilon}\Big)}{\sum_{j=1}^{n} x_{j,t-1}\ \exp\Big(\eta\ \frac{(\nabla f_t(\mathbf{x}_t))_j}{v_{j,t}+\varepsilon}\Big)}.$$

When running the algorithm against our dataset, we observe worse performance when considering past volatility for a variety of choices of λ compared against vanilla EG as shown in Figure 18. This signifies that being cautious on stocks with relatively more volatility is ineffective on highly volatile datasets. Note that this aligns well with ONS having particularly poor performance since past trends are directly used by the A_t matrix.

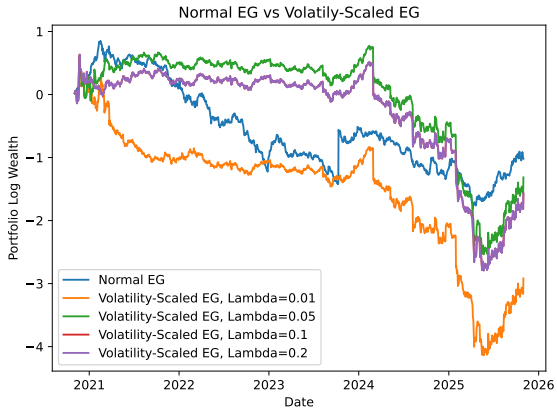

Figure 18: EG vs Volatility-Aware EG

4.4.3 Follow-The-Leader Variant

Attempting again to perform well against the volatile dataset, we experiment with three modified *Follow-The-Leader* algorithms summarized in the following list:

- ***1-Day Window:*** Allocate all wealth to the best stock of the previous day.
- ***3-Day Window:*** Allocate all wealth the best-performing stock over the past 3 days.
- ***Follow Positives:*** Evenly distribute wealth across all stocks with a positive return in the previous day. If no stock has a positive return, keep the current distribution.

We show in Figure 19 that none of these algorithms were able to handle the extreme nature of the penny stock data, all of which falling short of the uniform CRP baseline. In particular, in each case, over 98% of the portfolio value is lost (uniform CRP is also poor, losing around 62%). Since we are only accessing price relatives over full trading days, we are not letting our online algorithms change their distributions fast enough to account for large price swings. With access to price relatives for shorter periods of time, we hypothesize that the algorithms presented in Section 4.4 along with the vanilla algorithms would have more success. Additionally, more advanced algorithms such as ADA-

BARRONS [LWZ18] and VB-FTRL [JOG25] may be worth attempting here. Both of which, however, would also benefit from shorter trading periods.

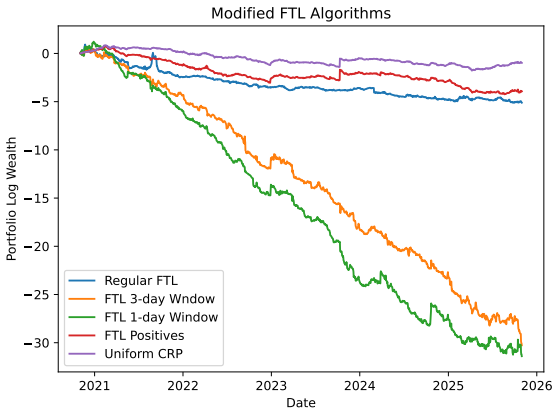

Figure 19: Modified FTL Algorithms

5 Summary and Results

This report explored different methods of online portfolio selection and their application to real stock market data. We implemented three core algorithms in Online Gradient Descent (OGD), Online Newton Step (ONS), and Exponentiated Gradient (EG) along with baseline strategies for comparison. Building from those, we experimented with adding bundles of stocks from our studied dataset as additional investable assets and observed the results on OGD. Next, we modified Online Newton Step based on historical data simulations with varying exponent choices for the A_t matrix's update. Our last group of experiments involved creating an EG algorithm with individualized volatility scalars applied to each stock along with three different FTL algorithms to test their effectiveness on a highly volatile dataset of penny stocks.

We found the most success with algorithms that used past data to drive modifications. In particular, the bundling strategies with backtested bundling choices and choosing a new ONS A_t update exponent proved to have the most favourable outcomes compared to the strategies they were based on. Another key observation was that some algorithm learning difficulties likely came from large round sizes. In every case, we considered price relatives between trading days, but each algorithm would have more opportunity to detect price movements if instead given price relatives from every hour, minute, or second.

6 Future Works

Given the understanding of online portfolio selection developed through this project, it would be interesting to delve into more theoretical and unexplored related topics, possibly considering im-

provements to the regret bounds or time complexity of complex and highly optimized algorithms like VB-FTRL [JOG25]. Another direction of interest would be setting up a trading account with access to more sophisticated and real time data, putting theoretical strategies into practice and trying to beat various benchmarks. This may introduce new challenges such as transaction fees and price latency which should be considered in algorithm development.

7 Appendix

7.1 Algorithms Implementations

Listing 1: Online Gradient Descent Implementation (Python)

```python
def ogd(self):
    # Initial weight spread is uniform distribution
    xt = self.weights.copy()

    # Init Decisions (X), Gradients (Grad), Losses (L)
    X = np.zeros((self.T, self.n))
    Grad = np.zeros((self.T, self.n))
    L = np.zeros(self.T)

    # Go through each time step and update the weight distribution according to OGD
    for t in range(self.T):
        # "Play" xt (observe loss - ft(xt) in textbook. Line 2 of Algorithm 1)
        X[t] = xt
        L[t] = self.loss(xt, t)

        # Line 3 in Algorithm 1
        Grad[t] = self.gradient(xt, t)
        self.computeEta(Grad[t], t)

        # Get xt for next round
        yNext = X[t] - self.etaScale * self.eta[t] * Grad[t]
        xt = cvxpyOgdProjectToK(yNext)

    growth = (X * self.priceRelatives).sum(axis=1)
    wealth = growth.cumprod()
    return X, wealth, L
```

Listing 2: Online Newton Step Implementation (Python)

```
def ons(self, alpha):
    # Initial weight spread is uniform distribution
    xt = self.weights.copy()

    At = 0

    X = np.zeros((self.T, self.n))
    Grad = np.zeros((self.T, self.n))
    L = np.zeros((self.T))

    for t in range(self.T):
        # "Play" xt and observe cost (line 2 in Algorithm 2)
        X[t] = xt
        L[t] = self.loss(xt, t)
        gradt = self.gradient(xt, t)
        Grad[t] = gradt

        self.computeGammaEpsilon(gradt, alpha)

        # Rank-1 update (line 3 in Algorithm 2)
        if t == 0:
            # 'A' starts as an epsilon scaled Identity matrix
            At = self.epsilon * np.eye(self.n)

        At  = At + np.outer(gradt, gradt) # does gt @ gtTranspose

        # Newton step
        # np.linalg.solve(a, b) computes x = a^(-1)b from ax = b
        invAg = np.linalg.solve(At, gradt)
        yt = xt - (1.0 / self.gamma) * invAg

        # cvx optimized projection:
        xt  = cvxpyOnsProjectToK(yt, At)

    growth = (X * self.data).sum(axis=1)
    wealth = growth.cumprod()
    return X, wealth, L
```

Listing 3: Exponentiated Gradient Implementation (Python)

```
def eg(self):
    # Initial weight spread is uniform distribution
    xt = self.weights.copy()

```

```
    X = np.zeros((self.T, self.n))
    L = np.zeros((self.T))

    for t in range(self.T):
        X[t] = xt
        L[t] = self.loss(xt, t)
        gradt = self.gradient(xt, t)

        eta = self.learnScalar * self.computeEta(self.data[t])

        # Use the previous time step's xt (xt not updated yet here)
        yt = xt * np.exp(-eta * gradt)

        # Normalize back to simplex by dividing by sum from PLG
        xt = yt / sum(yt)

    growth = (X * self.data).sum(axis=1)
    wealth = growth.cumprod()
    return X, wealth, L
```

Listing 4: Optimal CRP (Python)

```
def optimalCrpWeightsCvx(rt):
    T, n = rt.shape

    xt = cp.Variable(n)
    constraints = [
        xt >= 0,
        cp.sum(xt) == 1
    ]

    # Objective: maximize final wealth which is sumt log(xt^T rt)
    eps = 1e-12
    obj = cp.Maximize(cp.sum(cp.log(rt @ xt + eps)))
    prob = cp.Problem(obj, constraints)
    prob.solve()

    # Weights and cumulative wealth for optimal CRP
    return xt.value, np.cumprod(rt @ xt.value)
```

Listing 5: Uniform CRP (Python)

```
def uniformCRP(rt):
    n = rt.shape[1]
    uniform_weights = np.ones(n) / n  # 1/n allocation to each stock
```

```
    daily_growth = rt @ uniform_weights
    cumulative_wealth = np.cumprod(daily_growth)

    return cumulative_wealth
```

Listing 6: Follow-The-Leader (Python)

```
def followLeader(self):
    xt = self.weights.copy()
    X = np.zeros((self.T, self.n))
    L = np.zeros((self.T))

    cummulativeRet = np.zeros(self.n)

    for t in range(self.T):
        X[t] = xt
        rt = self.data[t]

        cummulativeRet += np.log(rt)
        bestIdx = np.argmax(cummulativeRet)

        xt = np.zeros_like(xt)
        xt[bestIdx] = 1.0

    growth = (X * self.data).sum(axis=1)
    wealth = growth.cumprod()
    return X, wealth, L
```

7.2 Projections

Listing 7: Projection to the Simplex (CVXPY)

```
def cvxpyOgdProjectToK(yt, At):
    n = yt.shape[0] # vector length
    x = cp.Variable(n)

    objective = cp.Minimize(cp.sum_squares(x-yt))
    constraints = [x >= 0, cp.sum(x) == 1]
    prob = cp.Problem(objective, constraints)
    prob.solve()

    return x.value
```

Listing 8: Online Newton Step Projection (CVXPY)

```
def cvxpyOnsProjectToK(yt, At):
    n = yt.shape[0]
    x = cp.Variable(n)

    objective = cp.Minimize(cp.quad_form(x - yt, At))
    constraints = [x >= 0, cp.sum(x) == 1]
    prob = cp.Problem(objective, constraints)
    prob.solve()

    return x.value
```

References

[CBL06] Nicolò Cesa-Bianchi and Gábor Lugosi. *Prediction, Learning, and Games*. Cambridge University Press, Cambridge, U.K., 2006.

[Con16] Laurent Condat. Fast projection onto the simplex and the ℓ_1 ball. *Mathematical Programming*, 158(1–2):575–585, 2016.

[Cor21] Cornell University Computational Optimization Open Textbook. Adagrad. `https://optimization.cbe.cornell.edu/index.php?title=AdaGrad`, 2021.

[Cov91] Thomas M. Cover. Universal portfolios. *Mathematical Finance*, 1(1):1–29, January 1991.

[Haz19] Elad Hazan. Introduction to online convex optimization, 2019. arXiv:1909.05207.

[JOG25] Rémi Jézéquel, Dmitrii M. Ostrovskii, and Pierre Gaillard. Efficient and near-optimal online portfolio selection. *Mathematics of Operations Research*, 2025.

[LH14] Bin Li and Steven C. H. Hoi. Online portfolio selection: A survey. *ACM Computing Surveys*, 46(3):1–36, January 2014.

[LWZ18] Haipeng Luo, Chen-Yu Wei, and Kai Zheng. Efficient online portfolio with logarithmic regret. In *Advances in Neural Information Processing Systems 31 (NeurIPS 2018)*, 2018. Also available as arXiv:1805.07430.

[vEvdHKK20] Tim van Erven, Dirk van der Hoeven, Wojciech Kotłowski, and Wouter M. Koolen. Open problem: Fast and optimal online portfolio selection. In Jacob Abernethy and Shivani Agarwal, editors, *Proceedings of the 33rd Conference on Learning Theory (COLT)*, volume 125 of *Proceedings of Machine Learning Research*, pages 3864–3869. PMLR, 2020.