# Portfolio policy parametrization

Master thesis submitted in accordance with the requirements of HEC Paris for the degree of Master in Management
by

**Charles Cros**

April 2022

# Contents

# Chapter 1

# Introduction

Finding a systematic, data-centric method for investing and allocating one's assets has long been a topic in the investment management industry.

Traditionally, investing has been dominated by a more "fundamental" approach. This latter simply consisted in acquiring securities one happened to know about and believed to have an interesting potential for generating returns. Within this framework, the investment decision is mostly based on qualitative criteria.

As opposed to this approach, systematic investing tries to find constant policies according to which one could allocate his/her capital. It can thus be seen as "meta-investing" since decisions are not taken at the security level, but at the investment universe level. The main advantage of this framework are manifold: better diversification, risk control, theoretical basis.

Since it requires to clearly set the investment objective, systematic investors usually used a "mean-variance" utility function (see appendix A.1, equation A.2) to derive optimal weights for the various assets in the investment universe. The main issue was of practical order. This range of methods requires one to estimate the distribution of future returns, or at least its first two moments (the vector of expected excess returns and the covariance matrix of returns) (see appendix A.1, equation A.4). To that end, investors usually use parametric models. However, this can be extremely difficult: accuracy can become hard to find as returns show a low level of predictability and the problem has a significantly high dimensionality.

Against this backdrop, Brandt, Santa-Clara and Valkanov (2009) [1] have suggested a new approach. Instead of estimating the moments of returns using parametric models, they directly try to (parametrically) estimate optimal weights: they bypass the theoretical optimal weight formula A.4. The main idea behind this method is to bypass the estimation of the return moments (expected returns and covariance matrix of returns mainly).

However, although seemingly simple, this method can meet several implementation hurdles. Depending on the computation and optimization approach one picks, estimation time can become significant, and an overfitting issue might come into play as well.

This present master thesis aims at replicating Brandt, Santa-Clara and Valkanov's results essentially using Python as a coding language, covering a base case where a simple, linear portfolio policy is adopted, a 'long-only' case and a final case where transaction costs are factored in the optimization process.

The motivation essentially stems from that fact that on one side, Python has become a widely used tool in the financial industry when it comes to data analysis and quantitative studies, and on the other side Brandt, Santa-Clara and Valkanov's paper provide well-performing results that could be of interest for many investors. Finding a good way to implement their method under Python could therefore be of use. Consequently, the final objective of this master thesis is to shed light on what could be the best (in terms of performance and estimation time) approach to such an implementation. In addition to their base, unconstrained case, it covers the case where portfolio weights cannot be negative ('long-only constraint') and transaction costs. The main reason for this is many investor are subject to a positivity constraint ('long-only' asset managers) and that transaction costs can materially affect the net performance of a portfolio.

# Chapter 2

# Methodology

## 2.1 Definitions

Let us consider in our present case that notations will be the same for linear applications from $\mathbf{R}^{a\times b}$ to $\mathbf{R}^b$ (for any $a, b \in \mathbf{N}^2$) and matrices of corresponding shapes in $\mathbf{R}^{a\times b}$.

Also, for any unidimensional function $a$, let us write its first order and second order derivatives $a'$ and $a''$.

For multidimensional functions $b : c_1, c_2, \ldots, c_m \mapsto b_1(c_1, c_2, \ldots, c_m), \ldots, b_n(c_1, c_2, \ldots, c_m)$, let us note its Jacobian matrix $J_b \equiv \begin{bmatrix} \frac{\partial b_1}{\partial c_1} & \frac{\partial b_1}{\partial c_2} & \cdots & \frac{\partial b_1}{\partial c_m} \\ \frac{\partial b_2}{\partial c_1} & \frac{\partial b_2}{\partial c_2} & & \frac{\partial b_2}{\partial c_m} \\ \vdots & & \ddots & \vdots \\ \frac{\partial b_n}{\partial c_1} & \cdots & \cdots & \frac{\partial b_n}{c_m} \end{bmatrix}$ (note that $J_b : \mathbf{R}^m \to \mathbf{R}^{m\times n}$). Finally, for functions $d : \mathbf{R}^m \mapsto \mathbf{R}$, let us note $\nabla_d$ its gradient and $\nabla_d^2$ its Hessian matrix when they exist.

Let us consider a period of $T \in \mathbf{N}$ months, a utility function $u : \mathbf{R} \to \mathbf{R}$. At each period $t \in [\![0; T-1]\!]$, let there be $N_t$ investable securities in our universe, with a return $r_{i,t}, \forall i \in [\![1; N_t]\!]$.

At each period $t$ and for each security $i$, let a set of exogeneous, $K$-dimensional data be noted $x_{i,t} \equiv (x_{1,i,t}, x_{2,i,t}, \ldots, x_{K,i,t})$. The method is to apply a weighting scheme $w_{i,t} : \mathbf{R}^{K+L} \to \mathbf{R}$ that depends on this data and on a set of $L$-dimensional parameters $\theta \equiv (\theta_1, \theta_2, \ldots, \theta_L)$. Weights should add up to 1 at any point in time: $\forall t \in [\![0; T-1]\!], \sum_{i=0}^{N_t} w_{i,t}(x_{i,t}, \theta) = 1$

Let us also define equivalents at the scale of the investment universe (and not at the scale of securities): $\forall t \in [\![0; T-1]\!], r_{t+1} \equiv (r_{1,t+1}, \ldots, r_{N_t,t+1})$, $x_t \equiv (x_{1,t}, \ldots, x_{N_t,t})$, $w_t \equiv$

$(w_{1,t}, \ldots, w_{N_t,t})$. Note that $r_{t+1}$ is of size $N_t \times 1$ and not $N_{t+1} \times 1$: only securities that are investable in $t$ are taken into account.

A first thing to stress would be that in our case, $\theta$ is not time dependent; however, it could be in some extensions of our problem.

## 2.2 Problem formulation

Following the approach of Brandt, Santa-Clara and Valkanov (2009) [1], the intertemporal utility should be maximised:

$$\max_{\theta} \Big[ \mathbf{E}\big[ u\big( w_t(x_t, \theta)^\intercal r_{t+1} \big) \big] \Big] \tag{2.1}$$

$$s.t.\ \ w_t(x_t, \theta)^\intercal \mathbf{1} = 1, \ \ \forall t \in [\![ 0; T-1 ]\!] \tag{2.2}$$

with $\mathbf{1} \equiv (1, \ldots, 1) \in \mathbf{R}^{N_t}$.

However, in practice, the optimization programme therefore becomes the following (empirical equivalent):

$$\max_{\theta} \Big[ \frac{1}{T} \sum_{t=0}^{T-1} u\big( w_t(x_t, \theta)^\intercal r_{t+1} \big) \Big] \tag{2.3}$$

subject to the same feasibility constraint (weights adding up to one). Our aim is to find the corresponding

$$\theta^* \equiv \arg\max_{\theta} \Big[ \frac{1}{T} \sum_{t=0}^{T-1} u\big( w_t(x_t, \theta)^\intercal r_{t+1} \big) \Big] \tag{2.4}$$

under the feasibility constraint.

Since in our present case, our utility function and weighting scheme are likely to be continuous and differentiable (at least twice), one can find the optimum using a first order condition.

Let the objective function be:

$$H : \theta \mapsto \frac{1}{T} \sum_{t=0}^{T-1} u\big( w_t(x_t, \theta)^\intercal r_{t+1} \big) \tag{2.5}$$

and the constraint functions $g \equiv (g_1, \ldots, g_T)$:

$$\forall t \in [\![ 0, T-1 ]\!], \ \ g_t : \theta \mapsto w_t(x_t, \theta)^\intercal \mathbf{1} - 1 \tag{2.6}$$

## 2.3 Approach

Finding the parameters $\theta^*$ that maximize the intertemporal utility $H$ (see 2.5) is generally done numerically. Although one can always simulate a number of different parameters and pick the one that maximizes the objective function ('naive' approach, cf. infra), this method can become very costly when computation costs and dimensions are high. This is why the present thesis explores other approaches to inferring the parameters.

The approaches that will be tested that are explained and tested in what follows rely on 'first order conditions'. These latter are equations of the form $m(\theta^*) = \mathbf{0}$. The function $m$ (and at times its Jacobian matrix $J_m$) are essential components used in different optimization algorithms. One thus need to find them, or at least approximate them, and then run an optimization algorithm.

In what follows, the first subsection dwells on what $m$ functions and $J_m$ matrix can be used in different cases (i.e. ways of handling the feasibility constraint). The second subsection dwells on what optimization algorithms are to be tested in the present master thesis.

### 2.3.1 First order condition

According to the saddle-point theorem, parameters that maximize the (convex) objective function should also be saddle-points of a Lagrangian function (linear combination of the objective function and the constraints). In what follows, the derived $m$ functions and $J_m$ matrices are thus the gradient and Hessian matrix of this Lagrangian function, respectively.

To compute $m$ and $J_m$, one thus first needs to set a Lagrangian function $\mathcal{L}$ that depends on the objective function and the feasibility constraint. However there are several ways of specifying this function, depending on how one chooses to handle the feasibility constraint. For each, analytical formulas for $m$ and $J_m$ will be provided in this section.

The first solution is to 'brute force' the constraint and directly run our optimization over a Lagrangian function that directly considers it as a set of equality constraints. While this may be a standard approach, it can be computationally costly (the number of constraints and thus multipliers to estimate grows with $T$). A second solution is to specify within the weighting scheme (at the investment universe scale) that the last weight should be 1 minus the sum of all others. However, it is hard to say if in practice, this method is less costly than the first one. Also, it holds less generalization power as it may be harder to implement extensions such as new constraints on weights. A third solution can be to embed an normalization form within the weighting scheme:

$$\forall t \in [\![0; T-1]\!], \forall i \in [\![1, N_t]\!], w_{i,t} \equiv \frac{\tilde{w}_{i,t}}{\sum_{j=1}^{N_t} \tilde{w}_{j,t}} \tag{2.7}$$

where $\tilde{w}_{i,t}$ denotes the weighting scheme before the normalization. Alternatively,

$$\forall t \in [\![0; T-1]\!], w_t \equiv (\tilde{w}_t^\intercal \mathbf{1}_{N_t})^{-1} \tilde{w}_t \tag{2.8}$$

One last solution is to bypass this constraint by normalizing (centering and reducing) the features to take into account in the model (cf. infra). While it is the most attractive approach in terms of computation costs, it is applicable only in specific cases (and restrictive) cases.

**Lagrangian method** The Lagrange multiplier theorem states that there exists a unique $\lambda^* \equiv (\lambda_0, \dots, \lambda_{T-1}) \in \mathbf{R}^T$ such that $\nabla_\theta H(\theta^*) = \lambda^* \nabla_\theta g(\theta^*)$. To solve the problem one can thus use a first order condition on our Lagrangian function $\mathcal{L} : \theta, \lambda \mapsto H(\theta) + \lambda g(\theta)$. It can be written as follows:

$$\frac{1}{T} \sum_{t=0}^{T-1} u'\Big(w_t(x_t, \theta^*)^\intercal r_{t+1}\Big)\Big(r_{t+1}^\intercal J_{w_t}(x_t, \theta^*)\Big) + T\lambda_t^* \mathbf{1}^\intercal J_{w_t}(x_t, \theta^*) = \mathbf{0} \tag{2.9}$$

One can adjust the $\lambda_t$ coefficients by a scale of $\frac{1}{T}$ to have a little bit simpler equations, hence:

$$m(\theta^*, \lambda^*) \equiv \frac{1}{T} \sum_{t=0}^{T-1} u'\Big(w_t(x_t, \theta^*)^\intercal r_{t+1}\Big) r_{t+1}^\intercal J_{w_t}(x_t, \theta^*) + \lambda_t^* J_{w_t}(x_t, \theta^*)^\intercal \mathbf{1} = \mathbf{0} \tag{2.10}$$

Here $\mathbf{0} = (0, \dots, 0) \in \mathbf{R}^L$

The Jacobian matrix of $m$ has then an $L \times (L + T)$ shape:

$$J_m = \frac{1}{T} \sum_{t=0}^{T-1} \Big[ \underbrace{J_{w_t}^\intercal r_{t+1} u''(w_t^\intercal r_{t+1}) r_{t+1}^\intercal J_{w_t} + u'(w_t^\intercal r_{t+1}) \sum_{i=1}^{N_t} r_{i,t+1} \nabla_{w_{i,t}}^2 + \sum_{i=1}^{N_t} \nabla_{w_{i,t}}^2}_{L \times L \text{ matrix}}, \underbrace{J_{w_t}^\intercal \mathbf{1}, \dots, J_{w_t}^\intercal \mathbf{1}}_{T \text{ matrices of shape } L \times 1} \Big] \tag{2.11}$$

As underlined earlier (cf. supra), in this case the complexity of the computation (size of $J_m$) to perform grows with the sample size, which is why this method will not to be privileged in the present case.

**1-minus method** The procedure is roughly the same, except the weighting scheme $w$ factors in the constraint. As a result, there is no apparent constraint in our programme. The first order condition becomes:

$$m(\theta^*) = \frac{1}{T} \sum_{t=0}^{T-1} u'\Big(w_t(x_t, \theta^*)^\intercal r_{t+1}\Big) r_{t+1}^\intercal J_{w_t}(x_t, \theta^*) = \mathbf{0} \tag{2.12}$$

Therefore estimating $\lambda^*$ is not needed. The Jacobian matrix of $m$ has then an $L \times L$ shape:

$$J_m = \frac{1}{T}\sum_{t=0}^{T-1}\Big[\underbrace{J_{w_t}^\intercal r_{t+1}u''(w_t^\intercal r_{t+1})r_{t+1}^\intercal J_{w_t} + u'(w_t^\intercal r_{t+1})\sum_{i=1}^{N_t} r_{i,t+1}\nabla^2_{w_{i,t}}}_{L\times L \text{ matrix}}\Big] \tag{2.13}$$

**Normalization method** The Jacobian matrix $J_m$ has essentially the same form as in the previous case (see equation 2.13). However, one still needs to express $J_{w_t}$ and $\nabla^2_{w_{i,t}}$ as a function of $J_{\tilde{w}_t}$, $\nabla_{\tilde{w}_{i,t}}$ and $\nabla^2_{\tilde{w}_{i,t}}$. Using the notations in equation 2.8:

$$\forall t \in [\![0; T-1]\!], J_{w_t} = (\mathbf{1}_{N_t}^\intercal \tilde{w}_t)^{-1} J_{\tilde{w}_t} - \tilde{w}_t\big((\mathbf{1}_{N_t}^\intercal \tilde{w}_t)^{-2}\mathbf{1}_{N_t}^\intercal J_{\tilde{w}_t}\big) \tag{2.14}$$

$$\forall t \in [\![0; T-1]\!], \forall i \in [\![1; N_t]\!], \nabla^2_{w_{i,t}} = (\mathbf{1}_{N_t}^\intercal \tilde{w}_t)^{-4}\left[\nabla^\intercal_{\tilde{w}_{i,t}}\big(\mathbf{1}_{N_t}^\intercal J_{\tilde{w}_t}\big) + \tilde{w}_{i,t}\Big(\sum_{j=1}^{N_t}\nabla^2_{w_{j,t}} - 2J_{\tilde{w}_t}^\intercal \mathbf{1}_{N_t}\mathbf{1}_{N_t}^\intercal J_{\tilde{w}_t}\Big)\right] \tag{2.15}$$

## 2.4 Optimization algorithms

The aim of this subsection is to give more detail on the different optimization algorithms that are to be tested.

### 2.4.1 Naive estimation

A first method for finding the optimal parameters of the model is a 'naive' one: one simply simulates a high number of parameters over a given interval (or hypercube), and then designates as estimator the one that optimizes the objective function (or minimize the norm of its gradient when the constraints cannot be internalized in the formulation of the objective function):

$$\hat{\theta} = \arg\max_{\theta\in\tilde{\Theta}} H(\theta) \tag{2.16}$$

where $\tilde{\Theta} = (\theta_1, \ldots, \theta_n)$ is a sample drawn from a possible space for parameters.

### 2.4.2 Gradient descent

A second method is a widely used approach in machine learning. The gradient descent consists in iteratively adding / substracting the gradient of the objective function / Lagrangian function to a previous starting parameter (depending on if one wants to maximize / minimize), up to a multiplicative factor $\eta$ (a 'step').

Several improvements of this method that provide more optimal steps have already been put in place, however the present master thesis will only cover the case where $\eta$ is constant. More specifically (given one wants to minimize $-H$):

1. Pick a starting vector of estimates $\hat{\theta}_0$
2. Successively update its value using the following equation:

$$\hat{\theta}_{n+1} = \hat{\theta}_n - \eta m(\hat{\theta}_n)^\intercal \tag{2.17}$$

3. Stop iterating when $m(\hat{\theta}_n)$ is close enough to $\mathbf{0}$, i.e. for a given $\epsilon \in \mathbf{R}_+^*$:

$$\|m(\hat{\theta}_n)\|_{\mathbf{I}}^2 \equiv m(\hat{\theta}_n)m(\hat{\theta}_n)^\intercal < \epsilon \tag{2.18}$$

Alternatively, one can also choose to stop iterations (also referred to as 'epochs') when a certain number of iterations is reached. In practice, it equates to replacing a 'while' loop by a 'for' loop, and helps control overfitting.

### 2.4.3 Newton-Ralphson

One method that generally converges more quickly towards optima than a simple gradient descent is the Newton-Ralphson algorithm. It consists in finding the roots of the objective / Langrangian function gradient through the Newton method. In practice, it is the same as the gradient descent, except the constant step $\eta$ is replaced by the inverse (or pseudo-inverse in a non-square case, i. e. when constraints are not internalized) of the Jacobian matrix of the gradient. More specifically:

1. Pick a starting vector of estimates $\hat{\theta}_0$
2. Successively update its value using the following equation:

$$\hat{\theta}_{n+1} = \hat{\theta}_n - (J_m^\intercal(\hat{\theta}_n)J_m(\hat{\theta}_n))^{-1}J_m^\intercal(\hat{\theta}_n)m(\hat{\theta}_n)^\intercal \tag{2.19}$$

3. Stop iterating when $m(\hat{\theta}_n)$ is close enough to $\mathbf{0}$, i.e. for a given $\epsilon \in \mathbf{R}_+^*$:

$$\|m(\hat{\theta}_n)\|_{\mathbf{I}}^2 \equiv m(\hat{\theta}_n)m(\hat{\theta}_n)^\intercal < \epsilon \tag{2.20}$$

**Numerical v. analytical gradient** The previous section details the analytical formulas for the gradients of our problem that are used in the optimization algorithms. However, deriving the precise formula for each use case can quickly become burdensome. This is why, in practice some may resort to the following approximation of the gradient, that one may define as 'numerical gradient':

$$\nabla_m \approx \begin{pmatrix} \frac{m(\theta_1+h,\theta_2,...,\theta_L)-m(\theta_1,\theta_2,...,\theta_L)}{h} \\ \frac{m(\theta_1,\theta_2+h,...,\theta_L)-m(\theta_1,\theta_2,...,\theta_L)}{h} \\ \vdots \\ \frac{m(\theta_1,\theta_2,...,\theta_L+h)-m(\theta_1,\theta_2,...,\theta_L)}{h} \end{pmatrix} \tag{2.21}$$

Testing the different estimation times for each of these methods is detailed in the 'Results' section.

In addition, PyTorch has an 'autograd' feature that automatically computes the analytical gradient of a model. This is another way of saving the handmade derivation of $m$ (see 3.2.3).

## 2.5 Evaluating the precision of estimators

Beyond estimating the parameters of a given weighting scheme, one may need to evaluate the significance of our estimators.

### 2.5.1 Generalized method of moments

Brandt, Santa-Clara and Valkanov [1] choose to interpret the first order condition 2.12 as a case of a generalized method of moments, so that one may derive an analytical formula for the variance (and therefore standard deviation) of our estimators (see appendix A.2 for details on the GMM). From Hansen (1982) [2], after having estimated the covariance matrix $V$ of our first order condition function this way (in the following a normalization approach is assumed):

$$V = \frac{1}{T}\sum_{t=0}^{T-1} u'\Big(w_t(x_t,\hat{\theta})^\intercal r_{t+1}\Big)^2 r_{t+1}^\intercal J_{w_t}(x_t,\hat{\theta}) J_{w_t}(x_t,\hat{\theta})^\intercal r_{t+1} \tag{2.22}$$

one can compute an asymptotic covariance matrix for our estimator $\hat{\theta}$:

$$\hat{\Sigma}_{\hat{\theta}} = \frac{1}{T}\big[G^\intercal V^{-1} G\big] \tag{2.23}$$

where $G \equiv \frac{1}{T}\sum_{t=0}^{T-1} u''(w_t^\intercal r_{t+1})\big(r_{t+1}^\intercal J_{w_t}\big)^\intercal (r_{t+1}^\intercal J_{w_t})$.

### 2.5.2 Bootstrapping

Brandt, Santa-Clara and Valkanov [1] also suggest a 'bootstrapping' method for the estimation of the covariance matrix of our estimator. The idea is to simulate the estimation of the target parameters $\theta^*$ $M$ times on $M$ subsamples of our dataset $((x_t, r_{t+1}))_{t\in[[0;T-1]]}$ that are drawn with replacement. On this population of size $M$, one simply computes the covariance

matrix.

One may note that while this method might appear simple to implement and robust, it has a significant drawback, which is the computation cost.

# Chapter 3

# Implementation

## 3.1 Starting point: linear model for weights, CRRA utility function, and 3 well-known factors

Following the example given by Brandt, Santa-Clara and Valkanov [1], we first choose to apply our approach to a basic framework: a linear specification for weights, a CRRA utility function and the 3 well-known factors from Fama, French, Booth and Sinquefield (1993) [3] and Carhart (1997) [4].

### 3.1.1 A linear model for a portfolio policy

A convenient choice the function $w_t, t \in [\![0, T-1]\!]$ can be a linear specification:

$$\forall t \in [\![0; T-1]\!], \forall i \in [\![1, N_t]\!], w_{i,t} = \bar{w}_{i,t} + \frac{1}{N_t}\theta^\intercal x_{i,t} \tag{3.1}$$

where $\bar{w}_{i,t}$ denotes the weight of asset $i$ in a benchmark portfolio at time $t$, and the features $x_{i,t}$ are standardized. The advantages of this specification are manifold:

- It simplifies the optimization problem: on top of simpler analytical forms in our optimization program, one does not need to factor in the feasibility constraint. As for all $i, t$, $\mathbb{E}[\theta^\intercal x_{i,t}] = \theta^\intercal \mathbb{E}[x_{i,t}] = 0$, the weights $w_{i,t}$ do add up to one. Note that this is specific to our linear weight function with standardized features;
- It can easily be interpreted;
- It captures the idea of active portfolio management as deviations from a benchmark.

### 3.1.2 A constant relative risk aversion utility

A constant relative risk aversion utility function (CRRA) is to be used:

$$\forall r \in \mathbf{R} : u(r) \equiv \frac{(1+r)^{1-\gamma}}{1-\gamma} \tag{3.2}$$

The interest of this function is namely the fact it is twice continuously differentiable, commonly used, reflects a decreasing marginal utility of returns and does not depend on the investor's initial wealth.

### 3.1.3 Features directly drawn from the Fama-French-Carhart 4-factor model

We choose to only use 4 exogenous inputs. For all $t$ in $[\![0; T-1]\!]$, for $i$ in $[\![1, N_t]\!]$, the weight of a stock in the benchmark portfolio $\bar{w}_{i,t}$ (interpreted as intercept), the log of its market capitalization $me_{i,t}$, the log of its book-to-market ratio $btm_{i,t}$ and its lagged one-year return $mom_{i,t}$. The main advantages of picking these factors as features for our problem is that we can check the coherence of our method with empirical results that can be found in the literature around risk factors impacting stock returns. As a consequence, one must infer $L = 3$ parameters. As for notations, $\bar{w}_{i,t}$ in $x_{i,t}$ is not included and as such $x_{i,t} = (me_{i,t}, btm_{i,t}, mom_{i,t})$ and $K = L = 3$, and $\bar{w}_t = (\bar{w}_{1,t}, \ldots, \bar{w}_{N_t,t})$.

### 3.1.4 Simpler formulas and problem

Following the previous specifications:

- A simple utility function first derivative:

$$\forall r \in \mathbf{R}, u'(r) = (1+r)^{-\gamma} \tag{3.3}$$

- A simple utility function second derivative:

$$\forall r \in \mathbf{R}, u''(r) = -\gamma(1+r)^{-1-\gamma} \tag{3.4}$$

- A simple weighting function 3.1;

- A simple Jacobian matrix for the weighting function ($K \times N_t$ shape):

$$\forall t \in [\![0; T-1]\!], J_{w_t}(x_t, \theta) = \frac{1}{N_t} x_t \tag{3.5}$$

- Even simpler Hessian matrices:

$$\forall t \in [[0; T-1]], \nabla^2_{w_{i,t}} = \mathbf{0} \tag{3.6}$$

Consequently:

- A simple first order condition (no constraint):

$$\frac{1}{T} \sum_{t=0}^{T-1} \left(1 + (\bar{w}_t + \frac{1}{N_t} x_t \theta)^\intercal r_{t+1}\right)^{-\gamma} \frac{1}{N_t} r_{t+1}^\intercal x_t = \mathbf{0} \tag{3.7}$$

- A simple Jacobian matrix for our $m$ function:

$$J_m(\theta) = \frac{-\gamma}{T}\sum_{t=0}^{T-1}\left(1+(\bar{w}_t+\frac{1}{N_t}x_t\theta)^\intercal r_{t+1}\right)^{-1-\gamma}\frac{1}{{N_t}^2}x_t^\intercal r_{t+1}r_{t+1}^\intercal x_t \tag{3.8}$$

By stating $\tilde{x}_t \equiv \frac{1}{N_t}x_t$ for all $t$:

$$m(\theta) = \frac{1}{T}\sum_{t=0}^{T-1}\left(1+(\bar{w}_t+\tilde{x}_t\theta)^\intercal r_{t+1}\right)^{-\gamma} r_{t+1}^\intercal \tilde{x}_t = \mathbf{0} \tag{3.9}$$

$$J_m(\theta) = \frac{-\gamma}{T}\sum_{t=0}^{T-1}\left(1+(\bar{w}_t+\tilde{x}_t\theta)^\intercal r_{t+1}\right)^{-1-\gamma} \tilde{x}_t^\intercal r_{t+1}r_{t+1}^\intercal \tilde{x}_t \tag{3.10}$$

Note that in our case, the Jacobian matrix is $K \times K$ and pseudo-inverse in equation 2.19 can be replaced by the simple inverse: the optimization algorithm becomes as follows:

1. Pick a starting vector of estimates $\hat{\theta}_0$
2. Successively update its value using the following equation:

$$\hat{\theta}_{n+1} = \hat{\theta}_n - J_m(\hat{\theta}_n)^{-1}m(\hat{\theta}_n)^\intercal \tag{3.11}$$

3. Stop iterating when $m(\hat{\theta}_n)$ is close enough to $\mathbf{0}$, i.e. for a given $\epsilon \in \mathbf{R}_+^*$:

$$\|m(\hat{\theta}_n)\|_{\mathbf{I}}^2 \equiv m(\hat{\theta}_n)m(\hat{\theta}_n)^\intercal < \epsilon \tag{3.12}$$

## 3.2 A first extension: long-only portfolios

In many cases, investors cannot short sell securities: they are under a constraint of keeping weights positive. In the previous approach, this is not accounted for and the parameters then found might not be optimal for a 'long-only' strategy. The next section aims at developing the method to effectively estimate optimal parameters in that case.

### 3.2.1 A non-linear model

Being subject to a 'long-only' constraint slightly tweaks the previous formulation of our portfolio policy 3.1. The weights are now given by the following formula:

$$\forall t \in [\![0; T-1]\!], \forall i \in [\![1, N_t]\!], w_{i,t} = \max\left(\bar{w}_{i,t} + \frac{1}{N_t}\theta^\intercal x_{i,t}, 0\right) \tag{3.13}$$

At this point weights do not 'naturally' sum up to one as they did before. This implies one needs to adopt one of the methods highlighted in Chapter 1 to handle the feasibility constraints. In what follows, the normalization approach 2.14 will be used.

### 3.2.2 Implications on optimization

One significant issue is that our objective function is no longer differentiable at any point in:

$$\Theta_{ND} \equiv \{\theta \in \mathbf{R}^L : \exists t \in [\![0; T-1]\!], i \in [\![1; N_t]\!] : \bar{w}_{i,t} + \frac{1}{N_t}\theta^\intercal x_{i,t} = 0\} \tag{3.14}$$

For this kind of issue, although one more optimal choice would be to use a 'subgradient' method, in practice machine-learning practitioners resort primarily to 'stochastic gradient descent' ('SGD'). This latter approach is very similar to gradient descent, excepts it does not use all of the dataset points to compute the gradient, but rather one (or a small number of data points in the case of mini-batch gradient descent).

In the present case, SGD can be equivalent to 'replacing' $H$ (see definition 2.5) with a simpler function $h_t : \theta \mapsto u(w_t(x_t, \theta)^\intercal r_{t+1})$ for a given $t$ and iterating the optimization steps of a traditional gradient descent algorithm for all $t$ in $[\![0; T-1]\!]$, and then over again until a satisfactory precision has been obtained. The $m$ function and its Jacobian matrix become respectively (using the normalization approach):

$$m : \theta \mapsto u'\Big(w_t(x_t, \theta)^\intercal r_{t+1}\Big) r_{t+1}^\intercal J_{w_t}(x_t, \theta) \tag{3.15}$$

$$J_m = J_{w_t}^\intercal r_{t+1} u''(w_t^\intercal r_{t+1}) r_{t+1}^\intercal J_{w_t} + u'(w_t^\intercal r_{t+1}) \sum_{i=1}^{N_t} r_{i,t+1} \nabla^2_{w_{i,t}} \tag{3.16}$$

Note that $m$ and $J_m$ are only defined over $\Theta_{ND}^c$. While in essence it does not solve the problem, in practice using this approach allows to only have extremely small probabilities to end on a point of non-differentiability.

### 3.2.3 Reinterpretation of the problem as a graph

In practice, implementing these model and optimization can be demanding, especially if each time one needs to analytically derive the gradient of our objective function. That is one can represent the previous methodology under a graph form. This way, implementation using specialized frameworks (Torch/Pytorch, Tensorflow) can be easier, notably when it comes to gradient computation through backpropagation (see appendix A.3) that can be done automatically.

One may note that it becomes equivalent to implementing a simple convolutional neural network that is simply composed of one convolutional layer, with one output channel, and with a kernel of shape $1 \times L$.

Considering a 'negative' utility ($-H$ or $-h$ depending on if one performs SGD or not, see objective function definition 2.5) as a loss function, the optimization problem becomes implementable under machine learning frameworks more easily.

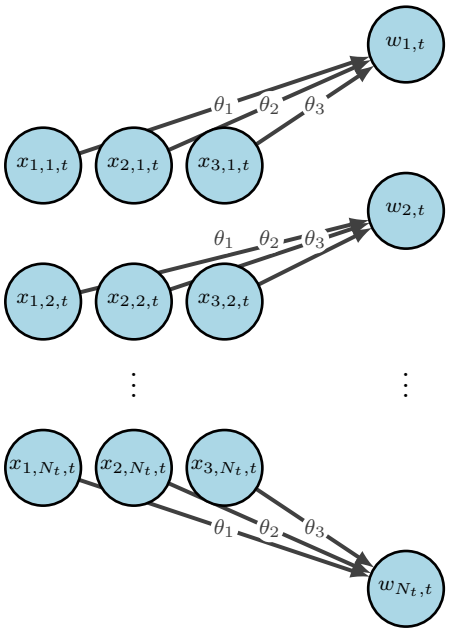


Figure 3.1: A graph representation of our simple, unconstrained linear model.

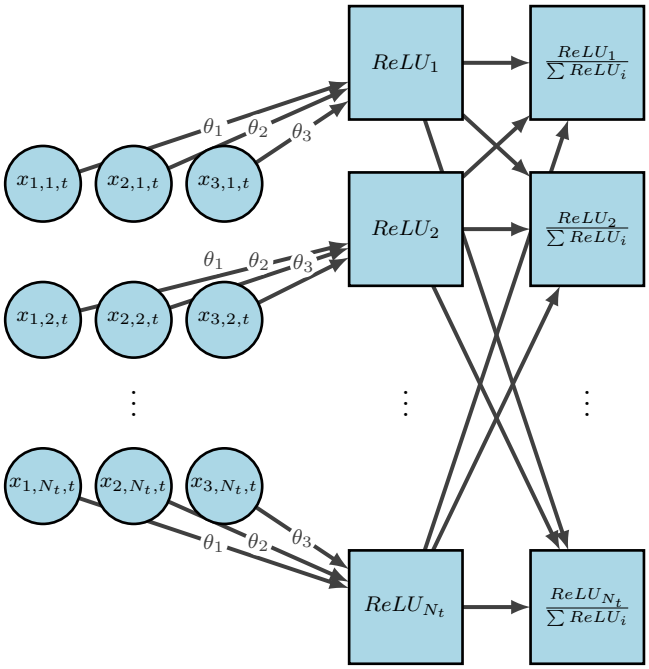


Figure 3.2: A graph representation of our long-only, normalized linear model.

## 3.3 A second extension: transaction costs

When dynamically implementing an investment policy in practice, investor face transaction costs. Investments based on previous approaches may prove too demanding in terms of portfolio turnover, and as a result the real, net financial performance of the investor can become very bad.

Consequently it appears important to include a way to factor in transaction costs in the optimization process.

### 3.3.1 Including transaction costs in the objective function

To account for transaction costs, Brandt, Santa-Clara and Valkanov (2009) [1] suggest to make it proportional to the turnover of each stock. More specifically, the total transaction costs at time $t$ are $\sum_{i=1}^{N_t} c_{i,t}|w_{i,t} - w_{i,t-1}|$. This implies that one can modify the objective function by 'regularizing' it using these costs:

$$H_{reg} : \theta \mapsto \frac{1}{T}\sum_{t=0}^{T-1} u\big(w_t(x_t,\theta)^\intercal r_{t+1} - \sum_{i=1}^{N_t} c_{i,t}|w_{i,t} - w_{i,t-1}|\big) \tag{3.17}$$

Let us define $h_{reg,t}\theta \mapsto u\big(w_t(x_t,\theta)^\intercal r_{t+1} - \sum_{i=1}^{N_t} c_{i,t}|w_{i,t} - w_{i,t-1}|\big)$ for the purpose of using SGD.

Although one could simply take a constant transaction cost across securities and time $c_{i,t} = 0.5\% \quad \forall t \in [\![0; T-1]\!], i \in [\![1; N_t]\!]$, this would fail to take into account the fact that these costs depend on the size of the companies and that they showcase a diminution over time.

Here, based on Keim and Madhavan (2009) [5], Domowitz, Glen and Madhavan (2001) [6], Hasbrouck (2006) [7], Brandt, Santa-Clara and Valkanov (2009) [1] suggest a linear explanation for the variations in transaction costs:

$$\forall t \in [\![0; T-1]\!], i \in [\![1; N_t]\!], c_{i,t} = (0.006 - 0.0025 me_{i,t})T_t \tag{3.18}$$

where $me_{i,t}$ denotes the size factor and is standardized between 0 and 1, and where $T_t$ is a 'trend such that costs in 1974 are four times larger than in 2002'. In what follows, $T_t$ is assumed linear in $t$, and to have average value of 0.5% over the period spanning from $t_1$ =01/01/1974 to $t_2$ =31/12/2002, and that $\sum_{i=0}^{N_{t_1}} c_{i,t_1} = 4 \times \sum_{i=0}^{N_{t_2}} c_{i,t_2}$.

The approach is to compute $\frac{T_{t_2}}{T_{t_1}} = \frac{N_{t_2}\sum_{i=0}^{N_{t_1}} z_{i,t_1}}{4N_{t_1}\sum_{i=0}^{N_{t_2}} z_{i,t_2}}$ and $T_{t_1} = \frac{1}{T}\sum_{t=0}^{T-1} \frac{1+(t-t_1)\frac{\frac{T_{t_2}}{T_{t_1}}-1}{t_2-t_1}}{0.005}\sum_{i=1}^{N_t}\frac{z_{i,t}}{N_t}$ and to plug them in:

$$T_t = T_{t_1} + (t - t_1)\frac{\frac{T_{t_2}}{T_{t_1}} - 1}{t_2 - t_1} T_{t_1} \tag{3.19}$$

## 3.4 Dataset adjustments

In practice, performing security-wise operations (at the '$i, t$' level) can become computationally costly since it requires a lot of Python operations. As Python is a high-level language, it can be pretty slow, and as a result estimation time can become very long.

On the contrary, time-period-wise operations (at the $t$ level) can be done much faster. The main reason is that most matricial operations in Python are done using libraries such as NumPy or PyTorch, which internally resort to a lower level language (such as C) and thus showcase much lower computation times.

This is why here we would like to speed up the computation of transaction costs by doing matricial operations such as $w_t - w_{t-1}$.

Now, let us consider the case where at a time $t$, one wants to invest in a security $i$ that just did not exist in $t-1$. In this case, $w_{i,t} - w_{i,t-1}$ can be problematic to compute in practice. Although one can set that $w_{i,t-1} = 0$ in this case, implementing this idea into code, since it may cause input size mismatches when one tries to compute $w_t - w_{t-1}$. The same goes for a security $j$ that existed in $t-1$ but does no longer exist in $t$: $w_{j,t} = 0$ but in practice this might cause mismatches as one performs matricial computations.

One solution to this could be to do a 'padding' of our dataset, meaning that for each stock one adds to the dataset lines where the stock does not exist and fill them with null values. However there are two major drawbacks to this approach. The first one is that one cannot guarantee weights of zero for formerly non-existing securities. The second one is the implied great sparsity of the dataset that may cause computation hurdles.

Instead, another solution (that will be tested) is to build state-transition matrices $M_t$ for each time $t$ that will handle the 'matching' of weights in $t-1$ and $t$. For instance, if at $t-1$ there were only 4 securities, that the first, second and fourth have 'survived' but not the third, and that two new securities appeared, the matrix $M_t$ would be the following:

$$M_t = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix} \tag{3.20}$$

The idea is to start from the identity matrix of size $N_{t-1}$, then remove the lines where securities disappeared, and finally add $N_t - N_{t-1}$ lines filled with zeros at the bottom of the

matrix. One may note that $M_t$ is of size $N_t \times N_{t-1}$.

This way, if one replaces $w_t - w_{t-1}$ by $w_t - M_t w_{t-1}$, one can quickly compute our transaction cost in a matricial fashion. Since no initial holdings are assumed, $M_0 w_{-1} \equiv \mathbf{0}_{\mathbf{R}^{N_0}}$ is assumed. Of course, for this to work one first needs to sort the securities by 'birth' dates.

## 3.5 Data

### 3.5.1 The CRSP-Compustat merged dataset

The present study is based on the CRSP-Compustat merged datasets. Our timeframe spans from 1964 to 2002. This dataset provides monthly security data and yearly financial information stocks (share code: 10, 11) trading in the following exchanges: NYSE, NASDAQ, AMEX (exchange code: 1, 2, 3). Since the focus is mainly on monthly total returns, size, value and momentum, only the following features are retained: $ret, dlret, at, lt, txditc, pstk, csho, prc$ (respectively the monthly total security return, the monthly return in case of delisting, the total assets, total liabilities, deferred taxes and investment credits, preferred shares, current number of shares outstanding, current share price). The accounting information that is available at time $t$ is ensured to be at least 6-month old.

Market capitalization is computed as $ME \equiv csho \times prc$, and the book value as $BV \equiv at - lt + txditc - pstk$, then size feature as $me = \ln(ME)$ and the book-to-market feature as $btm = \ln(1 + \frac{BV}{ME})$. Finally, the momentum feature at time $t$ is computed as the cumulative product of total gross returns (i.e. $1 + ret + dlret$) from months $t - 13$ to $t - 2$ included.

### 3.5.2 Practical considerations

In practice, searching through the dataset at each iteration in our optimization process can become very long. That is why, for speed considerations, $r_{t+1}$ are also added as a column, as well as $c_{i,t}$. Having these columns ready before the start of the optimization routine saves considerable time and possible, since they do not change with the model. On the contrary, the effective transaction costs $c_{i,t}|w_{i,t} - w_{i,t-1}|$ can make the whole process much longer for the same reason, but $w_{t-1}$ cannot be computed ex ante since it precisely depends on the optimized values of $\theta$.

Also, when it comes to PyTorch implementation and SGD, a 'data loader' needs to be built on top of the dataset, so that for each month $t$ data points are segregated. This way, only one month of data is passed at each computation of $h$. The dataloader is essentially a list of slices (according to the month $t$) of the whole dataset.

Finally, regarding the implementation of the transaction cost 'regularization', computing the state-transition matrices is relatively fast, however storing them can demand a lot of

memory space. This is why nesting the construction of these matrices within the data-loader allows not to save them.

# Chapter 4

# Results

## 4.1 Base case: unconstrained weights, no transaction costs

Tables 4.1 and 4.2 present parameters obtained for different optimization methods when using a simple linear portfolio policy as in our base case (respectively in-sample and out-of-sample). It also shows the standard deviations estimates obtained through bootstrapping, as well as different evaluation metrics for the implied portfolios. These are (in descending order): average absolute weight on one security ($\times 100$), maximum weight on one security ($\times 100$), minimum weight on one security ($\times 100$), average proportion of securities sold short, average weight of the short-selling part of the portfolio, portfolio turnover, certainty equivalent, average return of the portfolio, standard deviation of the portfolio returns, Sharpe ratio, $\alpha$ with regards to the market portfolio (intercept of the regression of excess returns over market excess returns), $\beta$ with regards to the market portfolio (coefficient of the regression of excess returns over market excess returns), the standard deviation of the residuals of the latter regression, Information Ratio, size, value and momentum characteristics of the portfolio, computed as $N_t \sum_{i=1}^{N_t} w_{i,t} x_{i,t}$. The 'BSCV' and 'Market' columns present the results obtained when weights are respectively computed using the parameters from Brandt, Santa-Clara and Valkanov (2009) [1] and (0 , 0, 0). The format is the same for all the results tables. In-sample estimates are computed using the whole 1964-2002 interval, whereas the out-of-sample estimates only using the 1964-1974 interval. Portfolio metrics are shown for the period 1974-2002.

The first thing to notice is that our first estimations remain somewhat consistent in signs with the literature. At least over this period of time, investors would want to be overweight value stocks showcasing momentum, and significantly positive parameters estimates for btm and mom are found. However, the estimate of $\theta_{me}$ remains disappointing as it appears not to be significant in two out of three approaches (naive and Newton-Ralphson). What is more, it appears significantly positive for the gradient descent method (GD). This could be

explained by the fact that as well underlined by Brandt, Santa-Clara and Valkanov, the sign of this parameter does change according to the slope of the yield curve over time. If one averages the estimation over time, statistical significance can be lost.

A second point would be to underline the similarities and differences between the different optimization approaches. In terms of computation time, to compute in-sample parameters, achieving results presented in 4.1 took the naive algorithm 120 seconds, the simple gradient descent algorithm 6 seconds and the Newton-Ralphson algorithm 2 seconds. However, Newton-Ralphson appears much more prone to overfitting the data than the other two methods. Indeed, it features much higher standard deviations in estimates and out-of-sample estimates become unsignificant. This could be attributed to the great speed at which it may converge towards local minima of our objective function (and stays stuck in it). As a consequence, the simple gradient descent might appear as a rather good choice (or at least better) for the optimization problem (see 4.3 for what kind of GD algorithm one could choose).

A third point would be to compare the magnitude of our results: in absolute value, they are almost always higher than the results of Brandt, Santa-Clara and Valkanov (2009) [1]. Again, this could be attributed to some extent to overfitting, as out-of-sample results appear to be less stable (higher standard deviation).

A fourth point would be about the respective portfolio performance. Our portfolios do show in- and out-of-sample outperformance compared to the market portfolio and the parameters from Brandt, Santa-Clara and Valkanov (2009) [1] in terms of Sharpe ratio, $\alpha$ and information ratio. However, they show underperformance in terms of certainty equivalent. Considering the previous point, one may simply explain this by the fact that our optimization appears less sensitive to risk-aversion, and allows itself to 'boost' deviations through greater parameters at the cost of increased volatility and $\beta$.

As with what follows, one may also explain a divergence in results stemming from differences between the present dataset, and the dataset used by Brandt, Santa-Clara and Valkanov's.

### 4.1.1 Numerical v. analytical gradient

In table 4.3, estimates found using numerically approximated gradients and analytically deduced ones are compared when using a gradent descent algorithm. Here the number of epochs is fixed to 100 and no precision threshold $\epsilon$ has been set, hence different results than for GD than in table 4.1. This way one can assess at the same time how 'far' are the parameters from the previously found minimum (and thus how quickly the method converges), and how much time it takes for one method to perform a given amount of epochs.

One can see that between a traditional analytical implementation (using Pandas and

In-sample

| Variable | Naive | GD (simple) | NR | BSCV | Market |
|---|---|---|---|---|---|
| $\hat{\theta}_{me}$ | 0.586 | 0.613 | 0.672 | -1.451 | - |
| $\sigma_{\hat{\theta}_{me}}$ | (2.253) | (0.321) | (1.557) | (0.548) | - |
| $\hat{\theta}_{btm}$ | 9.000 | 7.799 | 8.668 | 3.606 | - |
| $\sigma_{\hat{\theta}_{btm}}$ | (0.715) | (0.461) | (3.992) | (0.921) | - |
| $\hat{\theta}_{mom}$ | 7.566 | 7.415 | 7.992 | 1.772 | - |
| $\sigma_{\hat{\theta}_{mom}}$ | (0.351) | (0.450) | (3.522) | (0.743) | - |
| $\lvert w_{i,t}\rvert$ | 0.182 | 0.167 | 0.181 | 0.074 | 0.026 |
| $\max w_{i,t}$ | 8.699 | 8.589 | 9.241 | 5.098 | 5.487 |
| $\min w_{i,t}$ | -1.140 | -1.053 | -1.152 | -0.324 | - |
| $\Sigma \mathbf{I}(w_{i,t} < 0)/N_t$ | 0.473 | 0.471 | 0.473 | 0.442 | - |
| $\Sigma w_{i,t} \mathbf{I}(w_{i,t} < 0)$ | -2.782 | -2.517 | -2.780 | -0.880 | - |
| $\Sigma \lvert w_{i,t+1} - w_{i,t}\rvert$ | 2.129 | 2.027 | 2.199 | 0.657 | 0.078 |
| CE | 0.029 | 0.029 | 0.029 | 0.014 | 0.007 |
| $\bar{r}$ | 0.900 | 0.839 | 0.918 | 0.330 | 0.152 |
| $\sigma(r)$ | 0.711 | 0.640 | 0.717 | 0.299 | 0.171 |
| SR | 0.486 | 0.494 | 0.499 | 0.286 | 0.142 |
| $\alpha$ | 0.040 | 0.038 | 0.041 | 0.012 | - |
| $\beta$ | 1.381 | 1.353 | 1.386 | 1.065 | 1.000 |
| $\sigma(\epsilon)$ | 0.080 | 0.072 | 0.079 | 0.046 | - |
| IR | 0.504 | 0.526 | 0.522 | 0.264 | 0.131 |
| me | 1.719 | 1.913 | 1.867 | 0.334 | 2.141 |
| btm | 6.249 | 5.185 | 5.843 | 2.877 | -0.237 |
| mom | 5.043 | 5.162 | 5.507 | 0.804 | 0.019 |

Table 4.1: Simple linear portfolio policy - IS results

| Out-of-sample | | | | | |
|---|---|---|---|---|---|
| Variable | Naive | GD (simple) | NR | BSCV | Market |
| $\hat{\theta}_{me}$ | -0.175 | -0.029 | 0.129 | -1.124 | - |
| $\sigma_{\hat{\theta}_{me}}$ | (2.075) | (0.731) | (3.451) | (0.709) | - |
| $\hat{\theta}_{btm}$ | 6.388 | 4.944 | 5.608 | 3.611 | - |
| $\sigma_{\hat{\theta}_{btm}}$ | (1.857) | (0.882) | (6.592) | (1.110) | - |
| $\hat{\theta}_{mom}$ | 5.969 | 5.083 | 5.376 | 3.057 | - |
| $\sigma_{\hat{\theta}_{mom}}$ | (1.033) | (1.156) | (4.684) | (0.914) | - |
| $\lvert w_{i,t}\rvert$ | 0.136 | 0.113 | 0.124 | 0.081 | 0.026 |
| $\max w_{i,t}$ | 6.916 | 5.920 | 6.235 | 5.129 | 5.487 |
| $\min w_{i,t}$ | -0.839 | -0.686 | -0.753 | -0.427 | - |
| $\Sigma \mathbf{I}(w_{i,t} < 0)/N_t$ | 0.466 | 0.457 | 0.460 | 0.443 | - |
| $\Sigma w_{i,t} \mathbf{I}(w_{i,t} < 0)$ | -1.987 | -1.573 | -1.753 | -0.995 | - |
| $\Sigma \lvert w_{i,t+1} - w_{i,t}\rvert$ | 1.655 | 1.382 | 1.478 | 0.901 | 0.078 |
| CE | 0.026 | 0.024 | 0.025 | 0.018 | 0.007 |
| $\bar{r}$ | 0.680 | 0.572 | 0.614 | 0.404 | 0.152 |
| $\sigma(r)$ | 0.507 | 0.411 | 0.446 | 0.315 | 0.171 |
| SR | 0.466 | 0.452 | 0.460 | 0.366 | 0.142 |
| $\alpha$ | 0.031 | 0.025 | 0.027 | 0.016 | - |
| $\beta$ | 1.262 | 1.217 | 1.244 | 1.105 | 1.000 |
| $\sigma(\epsilon)$ | 0.061 | 0.049 | 0.053 | 0.041 | - |
| IR | 0.503 | 0.517 | 0.517 | 0.401 | 0.131 |
| me | 1.314 | 1.622 | 1.686 | 0.698 | 2.141 |
| btm | 4.325 | 3.179 | 3.697 | 2.559 | -0.237 |
| mom | 4.106 | 3.611 | 3.743 | 1.993 | 0.019 |

Table 4.2: Simple linear portfolio policy - OOS results

| Variable | Numerical GD | Analytical GD (pandas/numpy) | Analytical SGD (PyTorch) |
|---|---|---|---|
| me | 0.611 | 0.669 | -1.762 |
| btm | 8.413 | 8.595 | 2.285 |
| mom | 7.800 | 7.944 | 4.130 |
| Time elapsed | 7.575 | 23.961 | 7.445 |

Table 4.3: Comparison of GD methods - estimates and speed - IS results

NumPy) and a numerical one, the difference in convergence speed seems very thin. However, epochs with a numerical approach are performed almost three times faster. However, an analytical SGD implementation using PyTorch seems almost as fast as the numerical one.

## 4.2 Long-only portfolio constraint, no transaction costs

Tables 4.4 and 4.5 present parameters obtained when using a stochastic gradient descent and a PyTorch implementation for a long-only linear portfolio policy (respectively in-sample and out-of-sample).

Our estimated parameters are now almost consistent in sign with the literature. Also, one may notice the considerably reduced magnitude of the estimators (again, consistent with the literature). One may interpret this as the 'positivity constraint' acting de facto as a shrinkage method.

## 4.3 Unconstrained weights, transaction costs

Tables 4.6 and 4.7 present parameters obtained when using a stochastic gradient descent and a PyTorch implementation for a simple linear portfolio policy penalized with variable transaction costs (respectively in-sample and out-of-sample). Contrary to the estimates in the base case, a fixed number of epochs is used (rather than a precision threshold), for computation time purposes. The bootstrap estimates of standard deviations are done in a different way than in previous cases: instead of randomly drawing samples from the initial dataset, time 'windows' are drawn (i.e. samples with no holes along the $t$ axis) so that the time consistency of the method for computing transaction costs remains valid. Using this method we obtain considerably less samples and our estimates for standard deviations may be less reliable.

Looking at in-sample performance, one effectively reduces the overall turnover of the portfolio.

Penalizing the turnover may also act as shrinkage. Indeed, implying less turnover means being less 'sensitive' to signals (i.e. one would need much bigger values of $me, btm, mom$ to

In-sample

| Variable | SGD | BSCV | Market |
|---|---|---|---|
| $\hat{\theta}_{me}$ | -0.692 | -1.277 | - |
| $\sigma_{\hat{\theta}_{me}}$ | (0.399) | (1.217) | - |
| $\hat{\theta}_{btm}$ | 1.206 | 3.215 | - |
| $\sigma_{\hat{\theta}_{btm}}$ | (0.477) | (1.131) | - |
| $\hat{\theta}_{mom}$ | 1.710 | 1.416 | - |
| $\sigma_{\hat{\theta}_{mom}}$ | (0.438) | (1.213) | - |
| $\lvert w_{i,t} \rvert$ | 0.026 | 0.026 | 0.026 |
| $\max w_{i,t}$ | 36.998 | 31.250 | 32.578 |
| $\min w_{i,t}$ | - | - | - |
| $\Sigma \mathbf{I}(w_{i,t} < 0)/N_t$ | - | - | - |
| $\Sigma w_{i,t} \mathbf{I}(w_{i,t} < 0)$ | - | - | - |
| $\Sigma \lvert w_{i,t+1} - w_{i,t} \rvert$ | 0.261 | 0.208 | 0.083 |
| CE | 0.008 | 0.007 | 0.006 |
| $\bar{r}$ | 0.208 | 0.202 | 0.155 |
| $\sigma(r)$ | 0.213 | 0.228 | 0.175 |
| SR | 0.195 | 0.176 | 0.140 |
| $\alpha$ | 0.003 | 0.003 | - |
| $\beta$ | 1.096 | 1.092 | 1.020 |
| $\sigma(\epsilon)$ | 0.020 | 0.030 | 0.003 |
| IR | 0.169 | 0.105 | -0.024 |
| me | 1.023 | 0.376 | 2.193 |
| btm | 0.283 | 1.079 | -0.242 |
| mom | 0.794 | 0.203 | 0.019 |

Table 4.4: Long-only linear portfolio policy - IS results

| Out-of-sample | | | |
|---|---|---|---|
| Variable | GD | BSCV | Market |
| $\hat{\theta}_{me}$ | -0.439 | 0.651 | - |
| $\sigma_{\hat{\theta}_{me}}$ | (0.070) | (1.510) | - |
| $\hat{\theta}_{btm}$ | 0.808 | 2.679 | - |
| $\sigma_{\hat{\theta}_{btm}}$ | (0.107) | (1.417) | - |
| $\hat{\theta}_{mom}$ | 1.430 | 3.780 | - |
| $\sigma_{\hat{\theta}_{mom}}$ | (0.132) | (1.505) | - |
| $\lvert w_{i,t} \rvert$ | 0.026 | 0.026 | 0.026 |
| $\max w_{i,t}$ | 37.164 | 31.162 | 32.578 |
| $\min w_{i,t}$ | - | - | - |
| $\Sigma \mathbf{I}(w_{i,t} < 0)/N_t$ | - | - | - |
| $\Sigma w_{i,t} \mathbf{I}(w_{i,t} < 0)$ | - | - | - |
| $\Sigma \lvert w_{i,t+1} - w_{i,t} \rvert$ | 0.242 | 0.296 | 0.083 |
| CE | 0.008 | 0.008 | 0.006 |
| $\bar{r}$ | 0.200 | 0.214 | 0.155 |
| $\sigma(r)$ | 0.205 | 0.224 | 0.175 |
| SR | 0.190 | 0.194 | 0.140 |
| $\alpha$ | 0.003 | 0.004 | - |
| $\beta$ | 1.087 | 1.137 | 1.020 |
| $\sigma(\epsilon)$ | 0.016 | 0.022 | 0.003 |
| IR | 0.174 | 0.159 | -0.024 |
| me | 1.308 | 0.962 | 2.193 |
| btm | 0.094 | 0.342 | -0.242 |
| mom | 0.761 | 1.048 | 0.019 |

Table 4.5: Long-only linear portfolio policy - OOS results

act on it and change weights, otherwise the trading costs would exceed the gross outperformance). This is directly translated into lower coefficients in absolute value.

Also, our new estimates lead to an underperforming portfolio compared to previous estimates in terms of certainty equivalent, which can be puzzling. One may explain this result by the fact that the convergence speed of the optimization algorithm when trading costs are factored in becomes very slow and the optimization actually did not approach well enough the 'true' parameters.

Factoring in transaction costs considerably lengthens the training time. Achieving the present results took 250 seconds (without transaction costs, the training time is 6 seconds). To this long training time, one can add that computing the state-transition matrices takes an additional minute.

Therefore, in practice one may consider not to factor in transaction costs when it comes to fitting a model, but instead apply less computationally costly regularization methods, that should themselves effectively reduce the turnover and transaction costs.

In-sample

| Variable | SGD (penalty) | GD (simple, no penalty) | BSCV | Market |
|---|---|---|---|---|
| $\hat{\theta}_{me}$ | -0.681 | 0.613 | -1.167 | - |
| $\sigma_{\hat{\theta}_{me}}$ | (0.944) | (0.321) | (0.550) | - |
| $\hat{\theta}_{btm}$ | 1.457 | 7.799 | 3.160 | - |
| $\sigma_{\hat{\theta}_{btm}}$ | (0.805) | (0.461) | (0.924) | - |
| $\hat{\theta}_{mom}$ | 2.203 | 7.415 | 1.307 | - |
| $\sigma_{\hat{\theta}_{mom}}$ | (0.641) | (0.450) | (0.745) | - |
| $\lvert w_{i,t}\rvert$ | 0.051 | 0.135 | 0.064 | 0.026 |
| $\max w_{i,t}$ | 84.817 | 20.535 | 77.524 | 32.578 |
| $\min w_{i,t}$ | -7.540 | -61.722 | -9.933 | - |
| $\Sigma\mathbf{I}(w_{i,t} < 0)/N_t$ | 0.402 | 0.470 | 0.430 | - |
| $\Sigma w_{i,t}\mathbf{I}(w_{i,t} < 0)$ | -0.464 | -2.044 | -0.700 | - |
| $\Sigma\lvert w_{i,t+1} - w_{i,t}\rvert$ | 0.606 | 1.679 | 0.525 | 0.083 |
| CE | 0.013 | 0.023 | 0.014 | 0.006 |
| $\bar{r}$ | 0.280 | 0.545 | 0.292 | 0.152 |
| $\sigma(r)$ | 0.218 | 0.330 | 0.224 | 0.174 |
| SR | 0.296 | 0.419 | 0.281 | 0.143 |
| $\alpha$ | 0.009 | 0.026 | 0.011 | - |
| $\beta$ | 1.046 | 1.085 | 1.004 | 1.013 |
| $\sigma(\epsilon)$ | 0.023 | 0.061 | 0.041 | 0.003 |
| IR | 0.376 | 0.421 | 0.261 | 0.031 |
| me | 1.401 | 1.477 | 0.626 | 2.193 |
| btm | 0.715 | 4.159 | 2.435 | -0.242 |
| mom | 1.690 | 4.153 | 0.469 | 0.019 |

Table 4.6: Simple linear policy with transaction costs - IS results

| Out-of-sample | | | | |
|---|---|---|---|---|
| Variable | SGD (penalty) | GD (simple, no penalty) | BSCV | Market |
| $\hat{\theta}_{me}$ | -0.497 | -0.029 | -0.925 | - |
| $\sigma_{\hat{\theta}_{me}}$ | (0.665) | (0.731) | (0.780) | - |
| $\hat{\theta}_{btm}$ | 0.404 | 4.944 | 3.468 | - |
| $\sigma_{\hat{\theta}_{btm}}$ | (0.284) | (0.882) | (1.305) | - |
| $\hat{\theta}_{mom}$ | 0.782 | 5.083 | 2.497 | - |
| $\sigma_{\hat{\theta}_{mom}}$ | (0.233) | (1.156) | (0.961) | - |
| $\lvert w_{i,t} \rvert$ | 0.031 | 0.101 | 0.072 | 0.026 |
| $\max w_{i,t}$ | 43.748 | 49.768 | 155.935 | 32.578 |
| $\min w_{i,t}$ | -1.159 | -217.609 | -23.902 | - |
| $\Sigma \mathbf{I}(w_{i,t} < 0)/N_t$ | 0.267 | 0.456 | 0.435 | - |
| $\Sigma w_{i,t} \mathbf{I}(w_{i,t} < 0)$ | -0.091 | -1.412 | -0.854 | - |
| $\Sigma \lvert w_{i,t+1} - w_{i,t} \rvert$ | 0.252 | 1.279 | 0.761 | 0.083 |
| CE | 0.009 | 0.021 | 0.017 | 0.006 |
| $\bar{r}$ | 0.199 | 0.437 | 0.338 | 0.152 |
| $\sigma(r)$ | 0.188 | 0.270 | 0.230 | 0.174 |
| SR | 0.211 | 0.403 | 0.332 | 0.143 |
| $\alpha$ | 0.003 | 0.020 | 0.013 | - |
| $\beta$ | 1.026 | 1.066 | 1.029 | 1.013 |
| $\sigma(\epsilon)$ | 0.011 | 0.045 | 0.038 | 0.003 |
| IR | 0.308 | 0.435 | 0.350 | 0.031 |
| me | 1.703 | 1.373 | 0.848 | 2.193 |
| btm | 0.037 | 2.767 | 2.391 | -0.242 |
| mom | 0.642 | 3.157 | 1.444 | 0.019 |

Table 4.7: Simple linear policy with transaction costs - OOS results

# Chapter 5

# Conclusions

Implementing Brandt, Santa-Clara and Valkanov's strategy can meet several hurdles, especially when it comes to factoring in cases that depart from the basic (yet unrealistic) scenario where there are no transaction costs and constraints to investing.

When it comes to optimization, it appears that gradient descent may be the best compromise in terms of computation time (less costly than a naive method) and overfitting potential (less prone to this issue than the Newton-Ralphson method).

With regards to what method one may use to compute gradients, actually using a PyTorch framework that automatically derives them seem to be an interesting choice. First, it might appear as theoretically more robust than approximating the gradient between two close points, and second, the PyTorch framework opens the path for weighting scheme specifications beyond simple, linear cases, that can be more complex than what is traditionally used. Third, PyTorch may easily allow to test other optimization methods (such as SGD; however, while not covered in this master thesis, it could be interesting to see how Adagrad, Adam or RMSprop perform).

Finally, factoring in transaction costs proved extremely costly in terms of computation time, and this yielded disappointing results. Consequently, it might be more interesting to use other, not time-dependent regularization methods and to control for turnover ex-post.

# Appendix A

# Appendix

## A.1 The mean-variance framework

For a utility function $U$ for wealth (a stochastic random variable $W_t$) of a given investor, his/her investment decision is based on "expected utility": the objective is to maximize $\mathbb{E}[U(W_{t+1})]$ at any time $t$.

A common way of approaching this is to approximate this number by using a Taylor expansion to the second order:

$$\mathbb{E}[U(W_{t+1})] \approx U(W_t) + U'(W_t)\mathbb{E}W_t[r_p] + \frac{U''(W_t)W_t^2}{2}\mathbb{E}[r_p^2] \tag{A.1}$$

where $r_p \equiv \frac{W_{t+1}-W_t}{W_t}$. Rearranging this approximation, we define a "mean-variance" utility for returns $u$, meaning that the investor only cares about expected returns and their covariance matrix, and not about higher moments:

$$u(r_p) \equiv \mathbb{E}[r_p] + \frac{A}{2}\mathbb{V}[r_p] \tag{A.2}$$

where $A \equiv -\frac{U''(W_t)W_t}{U'(W_t)}$ is a level of risk aversion. As $A$ is deemed to be constant, we usually assume to have constant relative risk aversion preferences (CRRA).

Since one can write its portfolio return as follows:

$$r_p = r_f + w^\intercal \mu \tag{A.3}$$

where $r_f$ denotes the risk-free rate, $w$ the vector containing the weight on each asset, and $\mu$ the vector containing the excess return (over the risk-free rate) of each asset, maximizing $u(r_p$ over $w$ yields an optimal theoretical weight vector $w^*$ (analytical solution of the first-order condition):

$$w^* = \frac{1}{A}\Sigma^{-1}\mu \tag{A.4}$$

where $\Sigma$ denotes the covariance matrix of the asset returns.

## A.2 The generalized method of moments

Let us remind the general principle of this method (GMM). Since we want our estimate of $\theta^*$ and $\lambda^*$, $\hat{\theta}$ and $\hat{\lambda}$, to make $m(\hat{\theta}, \hat{\lambda})$ as close to $(0, \dots, 0) \in \mathbf{R}^{L+T}$, we consider a (weighted) norm over $\mathbf{R}^L$, $\|.\|_W$ ($W$ being the weighting matrix). We now aim at minimizing $\theta, \lambda \mapsto \|m(\theta, \lambda)\|_W$, or rather, for a simpler problem, $\theta, \lambda \mapsto \|m(\theta, \lambda)\|_W^2$.

Here, a tough choice is to find $W$ (note that it should be symmetric, positive-definite). Optimally, it would be $(m(\theta^*, \lambda^*)\, m(\theta^*, \lambda^*)^\intercal)^{-1}$, but of course we do not know $\theta^*$, which we want to estimate. There are several approaches to this issue. The first one is to start with a trivial $W$, then compute a first (temporary) estimate of $\theta^*$ and $\lambda^*$, and then recompute $W$ using this latter estimate, and then finally compute the final estimate for $\theta^*$ and $\lambda^*$. This is the 'two-step' approach. Note that it can be iterated several times. A second approach is to estimate $\theta^*$, $\lambda^*$ and $W$ jointly. A last approach, a bit specific to our case, is

A common starting matrix for the two-step approach is $W = \mathbf{I}_L$. Our procedure is then as follows:

1. We numerically estimate $\hat{\theta}, \hat{\lambda} \approx \arg\min_{\theta,\lambda} m(\theta, \lambda)^\intercal\, m(\theta, \lambda)$
2. We compute an estimate of $W$, $\hat{W} = (m(\hat{\theta}, \hat{\lambda})\, m(\hat{\theta}, \hat{\lambda})^\intercal)^{-1}$
3. We numerically estimate a final version $\bar{\theta}, \bar{\lambda} \approx \arg\min_{\theta,\lambda} m(\theta, \lambda)^\intercal\, \hat{W}\, m(\theta, \lambda)$

In our present case, we do not really need to implement this method to estimate our parameters. Instead, we already find them thanks to our previous optimization routine. However, we can leverage the properties of GMM estimators to find an analytical formula for their variances (Hansen, 1982 [2]). Here, we only plug-in our estimator of the parameters and directly approach $W$ by $(m(\hat{\theta}, \hat{\lambda})\, m(\hat{\theta}, \hat{\lambda})^\intercal)^{-1}$.

## A.3 Neural networks backpropagation and chain rule

Backpropagation is an approach to computing the gradient of a multivariate, differentiable function at a certain data point. It mainly relies on the chain rule. Let us remind that for differentiable functions $f : \mathbf{R}^{d_1} \longrightarrow \mathbf{R}^{d_2}$ and $g : \mathbf{R}^{d_2} \longrightarrow \mathbf{R}$, the chain rule allows to write the following:

$$\nabla_{f \circ g} = J_f^\intercal \nabla_g \tag{A.5}$$

If we consider $f_1, \dots, f_k$ to be layers of a neural network and $g$ a loss function, the gradient to use in our optimization algorithm can be computed as $J_{f_1}^\intercal \dots J_{f_k}^\intercal \nabla_g$. This is called 'backpropagation' of the gradient since we 'start' from $\nabla_g$ to successively (and in 'backward' fashion) update it with Jacobian matrix to get the final value of the gradient we want to compute.

## A.4 Code

The following script aims at handling the data so that it becomes exploitable.

```
import pandas as pd
import numpy as np
from tqdm import tqdm # For progress bars, not necessary
tqdm.pandas() # For progress bars, not necessary

# Step I: downloading data
# We start by downloading all the data we need from 3 datasets:
# 1/ ccm (Compustat-CRSP merged), that contains annual accruals
# 2/ crsp (CRSP), that contains monthly security market data
ccm = pd.read_csv('ccm.csv')
crsp = pd.read_table('crsp.txt', sep="\s+")
crsp['datadate'] = crsp.date
crsp = crsp.loc[crsp.datadate != 'date'] # We also drop all the lines that
↪  do not actually hold data

# Step II: building reliable indices (mapping of companies i and periods
↪  t)
# Constructing a t variable that handles the period count to simplify
↪  things (and not to have to use datadate, or any other datetime
↪  package)
# It is the 't' variable that goes from 0 to T-1

# There might be some mismatches in dates between crsp and ccm. For
↪  instance, the last day of a same month could be different. Here we
↪  index every month so that we bypass this problem.

datelist_crsp = pd.to_datetime(crsp['datadate'], format='%Y%m%d',
↪  errors='ignore').unique() # We take one occurrence of every date, and
↪  convert strings to dates at the same time
datelist_crsp.sort() # We sort our newly created dataframe (by date order)
datelist_crsp = pd.DataFrame(datelist_crsp,  columns=['date'])
datelist_crsp['date'] = datelist_crsp['date'].astype(str).apply(lambda x :
↪  x.replace('-',''))
datelist_crsp = datelist_crsp.reset_index().set_index('date')
datelist_crsp.columns = ['t'] # For readability

datelist_ccm = pd.to_datetime(ccm['datadate'], format='%Y%m%d',
↪  errors='ignore').unique() # We take one occurrence of every date, and
↪  convert strings to dates at the same time
```

```
datelist_ccm.sort() # We sort our newly created dataframe (by date order)
datelist_ccm = pd.DataFrame(datelist_ccm,  columns=['date'])
datelist_ccm['date'] = datelist_ccm['date'].astype(str).apply(lambda x :
↪  x.replace('-',''))
datelist_ccm = datelist_ccm.reset_index().set_index('date')
datelist_ccm.columns = ['t'] # For readability

crsp['t'] = crsp['datadate'].progress_apply(lambda date :
↪  datelist_crsp.loc[str(date)]) # We build the 't' column for crsp
ccm['t'] = ccm['datadate'].progress_apply(lambda date :
↪  datelist_ccm.loc[str(date)]) # Same for ccm

datelist_crsp.to_csv('datelist_crsp.csv') # saving, it will be of use later
datelist_ccm.to_csv('datelist_ccm.csv')

# We do not need to modify the 'LPERMCO' and 'PERMCO column that identify
↪  companies (acts as 'i' index)
# (i variable that goes from 1 to N_t)

crsp['i'] = crsp['PERMCO'].astype(str) # We make sure that every component
↪  has the same format, and use 'LPERMNO' as 'i'
ccm['i'] = ccm['LPERMCO'].astype(str)

# Step III: cleaning
# We replace '.' and other bothering letters by 0, and convert to float
↪  format the concerned columns
# We first need to define a function that will replace '.' by '0' (but
↪  only for data points that are '.', not for those which are 'X.XX')

def string_replace(x):
    if x in ('.','C','S','A','T','P'):
        return '0'
    else:
        return x

crsp['RET'] = crsp['RET'].apply(string_replace).astype(float)
crsp['DLRET'] = crsp['DLRET'].apply(string_replace).astype(float)
crsp['r'] = crsp['RET'] + crsp['DLRET']

crsp['PRC'] = crsp['PRC'].apply(string_replace).astype(float)
crsp['PRC'] = abs(crsp['PRC'])
```

```
ccm['at'] = ccm['at'].fillna(0) # We replace NaN by 0 for the accruals
ccm['lt'] = ccm['lt'].fillna(0)
ccm['pstk'] = ccm.pstk.fillna(0)
ccm['txditc'] = ccm.txditc.fillna(0)
ccm['csho'] = ccm.csho.fillna(0)

# Step IV: retaining only interesting columns

ccm = ccm[['at','csho','lt','pstk','txditc','t','i']]
crsp = crsp[['PRC','r','SHROUT','DLRET','vwretd','t','i']]

# Step V: merging crsp and ccm (the hard part)
# Here the main problem is that we need to match yearly data (ccm) to
↪  monthly data (crsp), according to a very specific rule (6-month
↪  information lag)
# Our approach is the following:
#   1/ For every month t, we extract all the available information in ccm.
↪  We make sure to respect a given information lag (here, 6 months).
#      By making a list while we iterate over t, we thus build a
↪  filtration (list of dataframes)
#   2/ For every month t, we extract from the filtration the latest data
↪  available
#   3/ We add back the current month t to each 'latest' dataframe, because
↪  the old 't' index becomes outdated (corresponds to the past months!)
#   4/ We concatenate everything and merge it with crsp
# All available information at time t

information_lag = 6
filtration_ccm = [ccm[ccm['t'] <= t-information_lag] for t in
↪  tqdm(datelist_ccm['t'])] # filtration

latest_ccm = [elt.groupby('i').max('t') for elt in tqdm(filtration_ccm)] #
↪  List of latest information available (list of dataframes)
latest_ccm2 = [] # At this point, the 't' column in the dataframes of
↪  latest_ccm point towards past dates, and not the current month.


# The following loop allows to add the current month and make it the new
↪  't' column. The latest_ccm2 will hold the completed dataframes.
for t, elt in tqdm(enumerate(latest_ccm)):
    elt = elt.reset_index() # For conveniency
```

```
    if elt.empty: # If there is no data, we create a new dataframe filled
    ↪  with np.nan
        elt = pd.DataFrame([np.nan] * len(elt.columns)).T
        elt.columns = ccm.columns
    time = pd.DataFrame([t] * len(elt)) # We create a new column filled
    ↪  with t
    time.columns = ['new_t']
    new_elt = pd.concat([elt, time], axis=1) # We add it to our data
    new_elt = new_elt.rename(columns={'t':'old_t', 'new_t':'t'}) # For
    ↪  conveniency
    latest_ccm2.append(new_elt)
latest_ccm2 = pd.concat(latest_ccm2) # Concatenation

data = pd.merge(crsp, latest_ccm2, on=['i','t']).drop(['old_t'], axis=1) #
↪  SUCCESSFUL MERGER!

# Step VI: building r_next
# r_next is the next month returns (very useful to avoid time-consuming
↪  queries in our optimization routine)
# We split by i, and for each we compute the next period returns

dfs = [] # Destined to be the new data
for i in tqdm(data.i.unique()): # For each company
    df = data[data.i == i].sort_values('t') # We extract a sorted dataframe
    ↪  containing all the data that concerns it
    df['r_next'] = df.shift(-1)['r'] # We append to it the shifted (towards
    ↪  1 month ahead) returns
    dfs.append(df) # We add this dataframe to the list dfs
data = pd.concat(dfs) # We concatenate everything to get a proper big
↪  dataframe

# Step VII: building factors
# Building intermediate features necessary in the computation of factors
↪  (using the same methodology as in the paper)

data['market_cap'] = data.csho * data.PRC # Number of shares outstanding
↪  times the price per share
data['book_value'] = data['at'] - data['lt'] + data.txditc - data.pstk #
↪  Assets - liabilities + deferred taxes and investment credits -
↪  preferred shares
# Building factors, according to the formulas in the paper
data['btm'] = np.log( 1 + ( data.book_value / data.market_cap ) )
```

```
data['me'] = np.log( data.market_cap )
dfs = [] # mom is a bit specific since we need past returns; dfs is
↪  destined to be the new data
for i in tqdm(data.i.unique()): # For each company
    df = data[data.i == i].sort_values('t') # We extract a sorted dataframe
    ↪  containing all the data that concerns it
    df['mom'] = ( ( 1 + df.r ).shift(2).rolling( window=12 ).apply(np.prod,
    ↪  raw=True) - 1) # We compute mom on this sub dataset
    dfs.append(df) # We add this dataframe to the list dfs
data = pd.concat(dfs) # We concatenate everything to get a proper big
↪  dataframe

# Step VIII: removing the 20% smallest firms (just like in the paper) for
↪  each period t
dfs = []
for t in tqdm(data.t.unique()):
    df = data[data.t == t]
    df = df[df.me > df.me.quantile(.2)]
    dfs.append(df)
data = pd.concat(dfs)

# Step IX: removing negative btm securities

data = data[data.btm >= 0]

# Step X: standardization (month by month)
# We start by computing averages and standard deviations of factors for
↪  each month
averages = data.groupby('t')[['btm','me','mom']].mean()
std_devs = data.groupby('t')[['btm','me','mom']].std()
# Then we make a column for each in the main dataset
data[['btm_avg','me_avg','mom_avg']] = data.t.progress_apply( lambda date :
↪  averages.loc[ date , : ] )
data[['btm_std','me_std','mom_std']] = data.t.progress_apply( lambda date :
↪  std_devs.loc[ date , : ] )
# Then we standardize the factors
data['btm_standardized'] = ( data.btm - data.btm_avg ) / data.btm_std
data['me_standardized'] = ( data.me - data.me_avg ) / data.me_std
data['mom_standardized'] = ( data.mom - data.mom_avg ) / data.mom_std

# Step XI: building w_hat, the weight of each stock within the market
↪  portfolio
```

```
total_market_value = data.groupby('t')['market_cap'].sum() # First we
↪  compute the market portfolio at each date
data['total_market_value'] = data.t.apply( lambda date : total_market_value[
↪  date ] ) # Then we assign to each line the corresponding value of the
↪  market at each date
data['w_hat'] = data.market_cap / data.total_market_value # Finally, we
↪  divide the market capitalization by the total market value to get
↪  w_hat

# Step XII: building w_equal, the weight of each stock within the
↪  equal-weighted portfolio
data['ones'] = np.ones(len(data)) # First we create a column only composed
↪  of ones (it therefore counts one for each company that exists at each
↪  date)
number_of_firms = data.groupby('t')['ones'].sum() # We then count the
↪  number of existing firms at each date
data['N'] = data.t.apply( lambda date : number_of_firms[ date ] ) # We
↪  assign it to each line of the main dataframe (just like before)
data['w_equal'] = data.ones / data.N # We divide the 'ones' column by this
↪  newly created feature to get the desired equal-weighting

# Step XIII: we create Tilde{x}, which simplies the computations (but not
↪  by much)
data['me_tilde'] = data['me_standardized'] / data['N']
data['btm_tilde'] = data['btm_standardized'] / data['N']
data['mom_tilde'] = data['mom_standardized'] / data['N']


# Step XIII: we create Tilde{x}, which simplies the computations (but not
↪  by much)
data['me_tilde'] = data['me_standardized'] / data['N']
data['btm_tilde'] = data['btm_standardized'] / data['N']
data['mom_tilde'] = data['mom_standardized'] / data['N']

data_standard_market_benchmark =
↪  data[['r','r_next','me_standardized','btm_standardized','mom_standardized','w_hat','
data_standard_market_benchmark =
↪  data_standard_market_benchmark.rename(columns={
    'me_standardized':'me',
    'btm_standardized':'btm',
    'mom_standardized':'mom',
    'w_hat':'w_bar'
```

```
})
data_tilde_market_benchmark =
↪  data[['r','r_next','me_tilde','btm_tilde','mom_tilde','w_hat','N','i','t']]
data_tilde_market_benchmark = data_tilde_market_benchmark.rename(columns={
    'me_tilde':'me',
    'btm_tilde':'btm',
    'mom_tilde':'mom',
    'w_hat':'w_bar'
})

# Step XIV: saving our tweaked dataset, at last!
data.to_csv('data_viz.csv') # We keep the big dataframe if we ever want to
↪  do some visualization
data_standard_market_benchmark.to_csv('data_standard_market_benchmark.csv')
data_tilde_market_benchmark.to_csv('data_tilde_market_benchmark.csv')
```

The following script aims at sorting the whole dataset according to the 'birth date' (date of appearance) of each security.

```
import pandas as pd
from tqdm import tqdm
df_name = 'data_tilde_market_benchmark.csv'
data = pd.read_csv(df_name, index_col=0).dropna()
birth_dates = dict({(i,data[data.i == i].t.min()) for i in
↪  tqdm(data.i.unique())})
data['birth_date'] = data.i.apply(lambda x : birth_dates[x])
data = data.sort_values(by=['birth_date'])
data.to_csv(df_name)
```

The following script computes the variable (and constant) transaction costs for each security at each point in time (when it exists).

```
"""This script aims at computing transaction costs for each stock at each
↪  month."""

import pandas as pd

#Picking the right timeframe
df_name = 'data_tilde_market_benchmark.csv'
data = pd.read_csv(df_name, index_col=0).dropna()

# Normalized me computation
data['normalized_me'] = (data.me - data.me.min()) / (data.me.max() -
↪  data.me.min())
```

```
# z computation
data['z'] = 0.006 - (0.0025 * data['normalized_me'])

# Defining t1 and t2
datelist = pd.read_csv('datelist_ccm.csv', index_col=0)
t1 = datelist.loc[19740131].values[0]
t2 = datelist.iloc[-2].values[0]

# b computation
T_ratio = data[data.t == t1].z.mean() / data[data.t == t2].z.mean() / 4
b = (T_ratio - 1) / (t2 - t1)

# T_t1 computation
temp1 = pd.Series([(1 + ((t - t1) * b))/0.005 for t in data.t.unique()])
temp2 = data.groupby('t')['z'].mean()
T_t1 = temp1.values.T @ temp2.values / len(data.t.unique())

# T_t computation
data['T_t'] = data.t.apply(lambda t: T_t1 + ((t-t1) * b * T_t1))

# Variable transaction cost computation
data['variable_c'] = data['T_t'] * data['z']

# Constant transaction cost assignment
data['constant_c'] = 0.005

data.to_csv(df_name)
```

The following script defines the $m$ function, the Jacobian matrix and the Hessian matrix of our base case.

```
# For tilde dataset
import pandas as pd
import numpy as np
import copy

data = pd.read_csv('data_tilde_market_benchmark.csv', index_col=0).dropna()

# Defining u, u_der, u_hes

def u(theta, data=data, gamma=5):
    df =  pd.DataFrame(
```

```
        data[['w_bar']].values + data[['me','btm','mom']].values @ theta,
        columns = ['r_next'] * theta.shape[1],
        index = data.index

        ) * data[['r_next']]

    df['t'] = data['t']

    return ( ( 1 + df.groupby('t').sum() )**( 1 - gamma ) / ( 1 - gamma )
    ↪  ).mean().values

def crra_prime(r, gamma=5):
    result = ( 1 + r )**( -gamma )
    return result

def r_T_dot_x(d):

    return d['r_next'].T @ d[['me','btm','mom']]

def u_der(theta, data=data, gamma=5):
    '''Gradient of the objective function; only supports one theta array
    ↪  at a time'''
    df =  pd.DataFrame(

        data[['w_bar']].values + data[['me','btm','mom']].values @ theta,
        columns = ['r_next'],
        index = data.index

        ) * data[['r_next']]

    df.columns = ['contribs']
    df[['me','btm','mom','t','r_next']] =
    ↪  data[['me','btm','mom','t','r_next']]


    data_grouped = df.groupby('t')
    u_p = data_grouped[['contribs']].sum().apply(crra_prime,
    ↪  gamma=gamma).values
    right_factor =
    ↪  data_grouped[['me','btm','mom','r_next']].apply(r_T_dot_x)
    return (u_p * right_factor).mean().values
```

```
def crra_second(r, gamma=5):
    result = ( 1 + r )**( -1-gamma ) * ( -gamma )
    return result

def u_hes(theta, data=data, gamma=5):
    '''Gradient of the objective function; only supports one theta array
    ↪  at a time'''
    T = len(data.t.unique())
    df =  pd.DataFrame(

        data[['w_bar']].values + data[['me','btm','mom']].values @ theta,
        columns = ['r_next'],
        index = data.index

        ) * data[['r_next']]

    df.columns = ['contribs']
    df[['me','btm','mom','t','r_next']] =
    ↪  data[['me','btm','mom','t','r_next']]


    data_grouped = df.groupby('t')
    u_s = data_grouped[['contribs']].sum().apply(crra_second,
    ↪  gamma=gamma).values
    right_factor =
    ↪  data_grouped[['me','btm','mom','r_next']].apply(r_T_dot_x)
    left_factor = u_s * copy.deepcopy(right_factor)
    result = left_factor.T @ right_factor
    return result.values / T
```

The following script defines 3 different optimization routine (and a bootstrap function) for our base case, as well as evaluation functions.

```
import pandas as pd
import numpy as np
from functions import *
from tqdm import tqdm
from statsmodels.regression.linear_model import OLS
from statsmodels.tools.tools import add_constant
import copy

# Naive: mini batch simulation on u
```

```
def naive(data, gamma=5, batch_size=10, batch_num=1000):

    res_theta, res_u = [], []

    for i in tqdm(range(batch_num)):

        theta_batch = ( np.random.rand(3,batch_size) - 0.25 ) * 20 # Check
        ↪  if naive estimator yields same result
        utilities = u(theta_batch, data=data, gamma=gamma)
        res_theta.append(pd.DataFrame(theta_batch))
        res_u.append(pd.DataFrame(utilities))

    res = pd.concat(res_theta, axis=1).T.reset_index()
    res['utility'] = pd.concat(res_u).reset_index()[0]

    theta_naive = res.loc[res.idxmax(0)['utility']][[0,1,2]].values #
    ↪  Results;

    return theta_naive

# Simple GD

def simple_GD(data, gamma=5, eps=1e-6, eta=10, iter_lim=20):

    theta_iter = np.zeros((3,1))
    grad = copy.deepcopy(u_der(theta_iter, data=data, gamma=gamma))
    loss = grad @ grad.T
    counter = 0
    while loss > eps:
        if counter > iter_lim:
            print(f"Did not converge; stopped at iteration {counter}")
            break
        grad = copy.deepcopy(u_der(theta_iter, data=data, gamma=gamma)) #
        ↪  deepcopy needed
        theta_iter += (eta * grad).reshape(3,1) # Adding because we want to
        ↪  maximize
        loss = grad @ grad.T
        counter += 1
    theta_gd = copy.deepcopy(theta_iter)
    return theta_gd
```

```
# Newton-Ralphson

def newton_ralphson(data, gamma=5, eps=1e-10, iter_lim=20):

    theta_iter = np.zeros((3,1))
    grad = copy.deepcopy(u_der(theta_iter, data=data, gamma=gamma))
    loss = grad @ grad.T
    hess = copy.deepcopy(u_hes(theta_iter, data=data, gamma=gamma))
    counter=0
    while loss > eps:
        if counter > iter_lim:
            print(f"Did not converge; stopped at iteration {counter}")
            break
        grad = copy.deepcopy(u_der(theta_iter, data=data, gamma=gamma))
        hess = copy.deepcopy(u_hes(theta_iter, data=data, gamma=gamma))
        theta_iter -= (np.linalg.inv(hess) @ grad).reshape(3,1)
        loss = grad @ grad.T
        counter+=1
    theta_nr = copy.deepcopy(theta_iter)
    return theta_nr

# Bootstrap

def bootstrap(samples, inference_method, **kwargs):

    thetas_bootstrap = [
        inference_method(
            sample,
            **kwargs
            ) for sample in tqdm(samples)
        ]

    return pd.concat([pd.DataFrame(elt) for elt in thetas_bootstrap],
    ↪  axis=1).T

# Evaluation metrics once estimation is done

def evaluation(theta, data=data, gamma=5):

    df = copy.deepcopy(data)
    df['weights'] = df[['w_bar']].values + df[['me','btm','mom']].values @
    ↪  theta
```

```
dfs = [] # Destined to be the new data
for i in tqdm(df.i.unique()): # For each company
    df_temp = df[df.i == i].sort_values('t') # We extract a sorted
    ↪  dataframe containing all the data that concerns it
    df_temp['w_next'] = df_temp.shift(-1)['weights'] # We append to it
    ↪  the shifted (towards 1 month ahead) returns
    dfs.append(df_temp) # We add this dataframe to the list dfs
df = pd.concat(dfs) # We concatenate everything to get a proper big
↪  dataframe

df['is_short'] = (df.weights < 0)*1
df['short_weights'] = df['is_short'] * df['weights']
df['trade'] = df['w_next'] - df['weights']
df['absolute_trades'] = abs(df['trade'])

df['contributions'] = df['weights'] * df['r_next']
gross_rets = 1 + df.groupby('t')['contributions'].sum()
annualized_gross_rets =
↪  gross_rets.rolling(12).apply(np.prod).dropna().iloc[::12]
rf = pd.read_csv('rf.csv',
↪  index_col=0).set_index('t').loc[gross_rets.index][['rf']]
excess_returns = gross_rets - 1 - rf['rf']
df['market_contributions'] = df['w_bar'] * df['r_next']

market_returns = df.groupby('t')['market_contributions'].sum()
market_excess_returns = market_returns - rf['rf']
model = OLS(excess_returns, add_constant(market_excess_returns))
reg_res = model.fit()

df['me_portfolio'] = df['N'] * df['weights'] * df['me']
df['btm_portfolio'] = df['N'] * df['weights'] * df['btm']
df['mom_portfolio'] = df['N'] * df['weights'] * df['mom']

results = {
    'average absolute weight' : abs(df['weights']).mean() * 100,
    'maximum weight' : df['weights'].max() * 100,
    'minimum weight' : df['weights'].min() * 100,
    'average proportion of shorted stocks' :
    ↪  df.groupby('t')['is_short'].mean().mean(),
    'average sum of shorted stocks' :
    ↪  df.groupby('t')['short_weights'].sum().mean(),
```

```
        'average turnover' :
        ↪  df.groupby('t')['absolute_trades'].sum().mean(),
        'certainty equivalent':
        ↪  ((gross_rets**(1-gamma)).mean())**(1/(1-gamma))-1,
        'average return' : annualized_gross_rets.mean() - 1,
        'standard deviation of returns' : annualized_gross_rets.std(),
        'Sharpe ratio' : excess_returns.mean() / excess_returns.std(),
        'alpha' : reg_res.params['const'],
        'beta' : reg_res.params[0],
        'sigma(eps)' : reg_res.resid.std(),
        'information ratio' : reg_res.params['const'] / reg_res.resid.std(),
        'me' : df.groupby('t')['me_portfolio'].sum().mean(),
        'btm' : df.groupby('t')['btm_portfolio'].sum().mean(),
        'mom' : df.groupby('t')['mom_portfolio'].sum().mean()
        }

    return results


def evaluation_pos_constraint(theta, data=data, gamma=5):

    df = copy.deepcopy(data)
    df['weights'] = df[['w_bar']].values + df[['me','btm','mom']].values @
    ↪  theta
    df['weights'] = (df['weights'] > 0) * df['weights']
    df['weights'] = df.groupby('t')['weights'].apply(lambda x: x/x.sum())

    dfs = [] # Destined to be the new data
    for i in tqdm(df.i.unique()): # For each company
        df_temp = df[df.i == i].sort_values('t') # We extract a sorted
        ↪  dataframe containing all the data that concerns it
        df_temp['w_next'] = df_temp.shift(-1)['weights'] # We append to it
        ↪  the shifted (towards 1 month ahead) returns
        dfs.append(df_temp) # We add this dataframe to the list dfs
    df = pd.concat(dfs) # We concatenate everything to get a proper big
    ↪  dataframe

    df['is_short'] = (df.weights < 0)*1
    df['short_weights'] = df['is_short'] * df['weights']
    df['trade'] = df['w_next'] - df['weights']
    df['absolute_trades'] = abs(df['trade'])
```

```
df['contributions'] = df['weights'] * df['r_next']
gross_rets = 1 + df.groupby('t')['contributions'].sum()
annualized_gross_rets =
↪  gross_rets.rolling(12).apply(np.prod).dropna().iloc[::12]
rf = pd.read_csv('rf.csv',
↪  index_col=0).set_index('t').loc[gross_rets.index][['rf']]
excess_returns = gross_rets - 1 - rf['rf']
df['market_contributions'] = df['w_bar'] * df['r_next']

market_returns = df.groupby('t')['market_contributions'].sum()
market_excess_returns = market_returns - rf['rf']
model = OLS(excess_returns, add_constant(market_excess_returns))
reg_res = model.fit()

df['me_portfolio'] = df['N'] * df['weights'] * df['me']
df['btm_portfolio'] = df['N'] * df['weights'] * df['btm']
df['mom_portfolio'] = df['N'] * df['weights'] * df['mom']

results = {
    'average absolute weight' : abs(df['weights']).mean() * 100,
    'maximum weight' : df['weights'].max() * 100,
    'minimum weight' : df['weights'].min() * 100,
    'average proportion of shorted stocks' :
    ↪  df.groupby('t')['is_short'].mean().mean(),
    'average sum of shorted stocks' :
    ↪  df.groupby('t')['short_weights'].sum().mean(),
    'average turnover' :
    ↪  df.groupby('t')['absolute_trades'].sum().mean(),
    'certainty equivalent':
    ↪  ((gross_rets**(1-gamma)).mean())**(1/(1-gamma))-1,
    'average return' : annualized_gross_rets.mean() - 1,
    'standard deviation of returns' : annualized_gross_rets.std(),
    'Sharpe ratio' : excess_returns.mean() / excess_returns.std(),
    'alpha' : reg_res.params['const'],
    'beta' : reg_res.params[0],
    'sigma(eps)' : reg_res.resid.std(),
    'information ratio' : reg_res.params['const'] / reg_res.resid.std(),
    'me' : df.groupby('t')['me_portfolio'].sum().mean(),
    'btm' : df.groupby('t')['btm_portfolio'].sum().mean(),
    'mom' : df.groupby('t')['mom_portfolio'].sum().mean()
    }
```

```
    return results


def evaluation_tc(theta, data=data, gamma=5):

    df = copy.deepcopy(data)
    df['weights'] = df[['w_bar']].values + df[['me','btm','mom']].values @
    ↪  theta
    df['weights'] = df.groupby('t')['weights'].apply(lambda x: x/x.sum())

    dfs = [] # Destined to be the new data
    for i in tqdm(df.i.unique()): # For each company
        df_temp = df[df.i == i].sort_values('t') # We extract a sorted
        ↪  dataframe containing all the data that concerns it
        df_temp['w_next'] = df_temp.shift(-1)['weights'] # We append to it
        ↪  the shifted (towards 1 month ahead) returns
        dfs.append(df_temp) # We add this dataframe to the list dfs
    df = pd.concat(dfs) # We concatenate everything to get a proper big
    ↪  dataframe

    df['is_short'] = (df.weights < 0)*1
    df['short_weights'] = df['is_short'] * df['weights']
    df['trade'] = df['w_next'] - df['weights']
    df['absolute_trades'] = abs(df['trade'])

    df['contributions'] = df['weights'] * df['r_next'] -
    ↪  df['absolute_trades'] * df['variable_c'] # Here we define
    ↪  contributions net of transaction costs
    gross_rets = 1 + df.groupby('t')['contributions'].sum()
    annualized_gross_rets =
    ↪  gross_rets.rolling(12).apply(np.prod).dropna().iloc[::12]
    rf = pd.read_csv('rf.csv',
    ↪  index_col=0).set_index('t').loc[gross_rets.index][['rf']]
    excess_returns = gross_rets - 1 - rf['rf']
    df['market_contributions'] = df['w_bar'] * df['r_next'] # Here we do
    ↪  not factor in the transaction costs (as a result, implementing a
    ↪  tracker strategy will not be totally correlated to the market
    ↪  performance)

    market_returns = df.groupby('t')['market_contributions'].sum()
    market_excess_returns = market_returns - rf['rf']
    model = OLS(excess_returns, add_constant(market_excess_returns))
```

```
    reg_res = model.fit()

    df['me_portfolio'] = df['N'] * df['weights'] * df['me']
    df['btm_portfolio'] = df['N'] * df['weights'] * df['btm']
    df['mom_portfolio'] = df['N'] * df['weights'] * df['mom']

    results = {
        'average absolute weight' : abs(df['weights']).mean() * 100,
        'maximum weight' : df['weights'].max() * 100,
        'minimum weight' : df['weights'].min() * 100,
        'average proportion of shorted stocks' :
        ↪  df.groupby('t')['is_short'].mean().mean(),
        'average sum of shorted stocks' :
        ↪  df.groupby('t')['short_weights'].sum().mean(),
        'average turnover' :
        ↪  df.groupby('t')['absolute_trades'].sum().mean(),
        'certainty equivalent':
        ↪  ((gross_rets**(1-gamma)).mean())**(1/(1-gamma))-1,
        'average return' : annualized_gross_rets.mean() - 1,
        'standard deviation of returns' : annualized_gross_rets.std(),
        'Sharpe ratio' : excess_returns.mean() / excess_returns.std(),
        'alpha' : reg_res.params['const'],
        'beta' : reg_res.params[0],
        'sigma(eps)' : reg_res.resid.std(),
        'information ratio' : reg_res.params['const'] / reg_res.resid.std(),
        'me' : df.groupby('t')['me_portfolio'].sum().mean(),
        'btm' : df.groupby('t')['btm_portfolio'].sum().mean(),
        'mom' : df.groupby('t')['mom_portfolio'].sum().mean()
        }

    return results
```

The following script defines the models and optimization routines under the PyTorch framework for our base case, long-only case, and transaction costs case.

```
import pandas as pd
import torch, copy
import torch.nn as nn
import torch.nn.functional as F
import numpy as np
from tqdm import tqdm

class UnconstrainedLinearModel(nn.Module):
```

```
    def __init__(self, K):
        super(UnconstrainedLinearModel, self).__init__()
        self.theta = nn.Parameter(torch.zeros(K, 1).double())

    def forward(self, x, w_bar):
        w = (torch.nan_to_num(x) @ self.theta) + w_bar
        w = w / w.sum()
        return w

class PositiveWeightsLinearModel(nn.Module):
    def __init__(self, K):
        super(PositiveWeightsLinearModel, self).__init__()
        self.theta = nn.Parameter(torch.zeros(K, 1).double())
        self.activation = nn.ReLU() # The only change
    def forward(self, x, w_bar):
        w = (torch.nan_to_num(x) @ self.theta) + w_bar
        w = self.activation(w) # The only change
        w = w / w.sum()
        return w

def negU(w, r_next, transaction_costs=0, gamma=5):
    '''Negative utility, that we will use as a loss function'''
    r_p = w.T @ torch.nan_to_num(r_next)
    r_p = r_p - transaction_costs
    return (r_p + 1)**(1-gamma) / (gamma-1)

def dataloader_converter(data, compute_state_transition=True):
    '''Returns a list of exploitable data, from t = 0 to T-1'''

    sorted_t = data.sort_values(by=['t']).t.unique()

    if compute_state_transition:

        # Computing state-transition matrices

        # Initializing
        firstDataSlice = data[data.t == data.t.min()]
        M = [torch.zeros(len(firstDataSlice), len(firstDataSlice))] #
        ↪  setting M0

        # Looping
```

```
        for t in tqdm(sorted_t[1:-1]):
            dataPrev = data[data.t == t-1].reset_index()
            dataNow = data[data.t == t]
            deaths = [elt for elt in dataPrev.i.values if elt not in
            ↪  dataNow.i.values]
            to_kill = dataPrev[dataPrev.i.isin(deaths)].index
            M_t = np.delete(np.identity(len(dataPrev)), to_kill, 0)
            zeros_to_add =  len(to_kill) + len(dataNow) - len(dataPrev)
            zero_matrix = np.zeros((zeros_to_add, len(dataPrev)))
            M_t = np.vstack((M_t, zero_matrix))
            M.append(torch.tensor(M_t))

        # Indexing
        M_indexed = dict(zip(sorted_t, M))

    else:
        M = torch.zeros(len(sorted_t))
        M_indexed = dict(zip(sorted_t, M))

    return [{
        'x': torch.tensor(data[data.t == t][['me','btm','mom']].values),
        'w_bar': torch.tensor(data[data.t == t][['w_bar']].values),
        'r_next': torch.tensor(data[data.t == t][['r_next']].values),
        'r' : torch.tensor(data[data.t == t][['r']].values),
        'variable_c': torch.tensor(data[data.t ==
        ↪  t][['variable_c']].values),
        'constant_c': torch.tensor(data[data.t ==
        ↪  t][['constant_c']].values),
        'M': M_indexed[t].double()
    } for t in tqdm(sorted_t[:-1])]

def inference(model, data, gamma=5, dataloader=None, epoch_num=10, lr=0.1,
↪  is_me_btm_mom=True):
    optimizer = torch.optim.SGD(model.parameters(), lr=lr)

    if dataloader is None:
        dataloader = dataloader_converter(data,
        ↪  compute_state_transition=False) # May take some time

    for _ in tqdm(range(epoch_num)):
        for d in dataloader:
            x, w_bar, r_next = d['x'], d['w_bar'], d['r_next']
```

```
            w = model(x, w_bar)
            loss = negU(w, r_next, gamma=gamma)
            model.zero_grad()
            loss.backward()
            optimizer.step()

    if is_me_btm_mom:
        params =
        ↪  copy.deepcopy(pd.DataFrame(nn.utils.parameters_to_vector(model.parameters())
        ↪  index=['me','btm','mom']))
    else:
        params =
        ↪  copy.deepcopy(nn.utils.parameters_to_vector(model.parameters()).detach().num
    return params

def inference_with_transaction_costs(model, data, dataloader = None,
↪  cost_type = 'variable_c', gamma=5, epoch_num=50,lr=0.05,
↪  is_me_btm_mom=True):
    optimizer = torch.optim.SGD(model.parameters(), lr=lr)

    if dataloader is None:
        dataloader = dataloader_converter(data) # May take some time

    for _ in tqdm(range(epoch_num)):
        # Doing the first step
        d = dataloader[0]
        r_next = d['r_next']
        hold_w = torch.zeros_like(r_next).double()
        x, w_bar, r_next, c, M = d['x'], d['w_bar'], d['r_next'],
        ↪  d['variable_c'], d['M'] # We then load tha values of x_t,
        ↪  r_t+1, w_bar_t, and the matrix M_t
        w = model(x, w_bar) # We compute the weights
        transaction_costs = c.T @ abs(w - hold_w) # No M at first
        loss = negU(w, r_next, transaction_costs, gamma=5)

        for d in dataloader[1:-1]: # For all t from 0 to T-1
            hold_w = w * (1 + r_next) / (1 + w.T @ r_next) # We need to
            ↪  compute the hold portfolio before we update the values of
            ↪  w and r_next
```

```
            x, w_bar, r_next, c, M = d['x'], d['w_bar'], d['r_next'],
            ↪  d['variable_c'], d['M'] # We then load tha values of x_t,
            ↪  r_t+1, w_bar_t, and the matrix M_t
            w = model(x, w_bar) # We compute the weights
            transaction_costs = c.T @ abs(w - M @ hold_w) #
            loss = loss + negU(w, r_next, transaction_costs, gamma=5) # We
            ↪  add the instantaneous 'loss' (i.e. negative utility, -h)
            ↪  to the overall loss (-H)

        model.zero_grad() # Resets the jacobian matrix of the model
        loss.backward(retain_graph=True) # Computes the gradient of the
        ↪  loss ; need to retain the graph so that intermediate values
        ↪  are kept (w_t-1 for instance)

        optimizer.step() # Performs the optimization algorithm step, i.e.
        ↪  theta <- theta - eta * gradient

    if is_me_btm_mom:
        params =
        ↪  copy.deepcopy(pd.DataFrame(nn.utils.parameters_to_vector(model.parameters())
        ↪  index=['me','btm','mom']))
    else:
        params =
        ↪  copy.deepcopy(nn.utils.parameters_to_vector(model.parameters()).detach().num
    return params

def bootstrap(model, data, sample_size=10000, sample_num=1000, **kwargs):
    sampling = [data.sample(sample_size) for i in range(sample_num)]
    thetas = [
        inference(model, sample, **kwargs) for sample in tqdm(sampling)
    ]
    return pd.concat([pd.DataFrame(elt) for elt in thetas], axis=1).T
```

The following scripts compute and store as csv files the inferred parameters for all our cases (as well as their standard deviations).

```
import pandas as pd
from functions import *
from estimation import *

data = pd.read_csv('data_tilde_market_benchmark.csv', index_col=0).dropna()

# Hyperparameters for naive estimation
```

```
kw_naive = {
    'gamma':5,
    'batch_size':10,
    'batch_num':100
}

# Hyperparameters for GD
kw_gd = {
    'gamma':5,
    'eps':1e-8,
    'eta':100,
    'iter_lim':100
}

# Hyperparameters for NR
kw_nr = {
    'gamma':5,
    'eps':1e-10,
    'iter_lim':10
}

# Estimates

theta_naive = naive(data, **kw_naive)
theta_gd = simple_GD(data, **kw_gd)
theta_nr = newton_ralphson(data, **kw_nr)

estimates = pd.concat([pd.DataFrame(elt.reshape(1,3)) for elt in
↪  (theta_naive, theta_gd, theta_nr)])
estimates.columns = ('me','btm','mom')
estimates.index = ('naive','simple_GD','newton_ralphson')

estimates.to_csv('estimates_IS.csv')

# Bootstrapped standard deviations

# Sampling once and for all
samples = [data.sample(frac=0.20) for i in range(100)] # 40 minutes

# Estimates of standard deviations
std_naive = bootstrap(samples, naive, **kw_naive).std()
std_gd = bootstrap(samples, simple_GD, **kw_gd).std()
```

```
std_nr = bootstrap(samples, newton_ralphson, **kw_nr).std()

stds = pd.concat([std_naive, std_gd, std_nr], axis=1).T
stds.columns = ('std_me','std_btm','std_mom')
stds.index = ('naive','simple_GD','newton_ralphson')

stds.to_csv('standard_deviations_IS.csv')
```

```
import pandas as pd
from functions import *
from estimation import *

data = pd.read_csv('data_tilde_market_benchmark.csv', index_col=0).dropna()
datelist = pd.read_csv('datelist.csv', index_col=0) # Loading list of dates
↪  to select the right index
stop_date = 19740131 # Defining a stop date to limit the training set
stop_t = datelist.loc[stop_date].values[0] # Determining the right index in
↪  the list
data = data[data.t < stop_t] # Refocusing the dataset

# Hyperparameters for naive estimation
kw_naive = {
    'gamma':5,
    'batch_size':10,
    'batch_num':100
}

# Hyperparameters for GD
kw_gd = {
    'gamma':5,
    'eps':1e-8,
    'eta':100,
    'iter_lim':100
}

# Hyperparameters for NR
kw_nr = {
    'gamma':5,
    'eps':1e-10,
    'iter_lim':10
}

# Estimates
```

```
theta_naive = naive(data, **kw_naive)
theta_gd = simple_GD(data, **kw_gd)
theta_nr = newton_ralphson(data, **kw_nr)

estimates = pd.concat([pd.DataFrame(elt.reshape(1,3)) for elt in
↪  (theta_naive, theta_gd, theta_nr)])
estimates.columns = ('me','btm','mom')
estimates.index = ('naive','simple_GD','newton_ralphson')

estimates.to_csv('estimates_OOS.csv')

# Bootstrapped standard deviations

# Sampling once and for all
samples = [data.sample(frac=0.20) for i in range(100)] # 40 minutes

# Estimates of standard deviations
std_naive = bootstrap(samples, naive, **kw_naive).std()
std_gd = bootstrap(samples, simple_GD, **kw_gd).std()
std_nr = bootstrap(samples, newton_ralphson, **kw_nr).std()

stds = pd.concat([std_naive, std_gd, std_nr], axis=1).T
stds.columns = ('std_me','std_btm','std_mom')
stds.index = ('naive','simple_GD','newton_ralphson')

stds.to_csv('standard_deviations_OOS.csv')
```

```
from graph_representation_functions import *
import pandas as pd
import copy

data = pd.read_csv('data_tilde_market_benchmark.csv', index_col=0)

model = PositiveWeightsLinearModel(3)
thetas_bootstrap = bootstrap(model, data, sample_size=100000,
↪  sample_num=1000)

stds = pd.DataFrame(thetas_bootstrap.std()).T
stds.index = ['positive_constraint']
stds.to_csv('standard_deviations_pos_constraint_IS.csv')

theta_hat = inference(model, data).T
```

```
theta_hat.index = ['positive_constraint']
theta_hat.to_csv('estimate_pos_constraint_IS.csv')

from graph_representation_functions import *
import pandas as pd

data = pd.read_csv('data_tilde_market_benchmark.csv', index_col=0)
datelist = pd.read_csv('datelist.csv', index_col=0) # Loading list of dates
↪  to select the right index
stop_date = 19740131 # Defining a stop date to limit the training set
stop_t = datelist.loc[stop_date].values[0] # Determining the right index in
↪  the list
data = data[data.t < stop_t] # Refocusing the dataset

model = PositiveWeightsLinearModel(3)
thetas_bootstrap = bootstrap(model, data, sample_size=100000,
↪  sample_num=1000)

stds = pd.DataFrame(thetas_bootstrap.std()).T
stds.index = ['positive_constraint']
stds.to_csv('standard_deviations_pos_constraint_OOS.csv')

theta_hat = inference(model, data).T
theta_hat.index = ['positive_constraint']
theta_hat.to_csv('estimate_pos_constraint_OOS.csv')

# Step I: imports
import torch
import pandas as pd
import numpy as np
from graph_representation_functions import *
from tqdm import tqdm

data = pd.read_csv('data_tilde_market_benchmark.csv', index_col=0)

# Step II: instanciate dataloader
dataloader = dataloader_converter(data) # May crash

# Step III: estimate unconstrained with TC
L = 3
model = UnconstrainedLinearModel(L)
params = inference_with_transaction_costs(model, data,
↪  dataloader=dataloader, epoch_num=20, lr=0.3)
```

```
params.columns = ['transaction_costs']
params.T.to_csv('estimates_transaction_costs_IS.csv')

# Step IV: bootstrapping with tc

L=3
sample_size = 100
sample_number = len(dataloader) - sample_size - 1

bootstrap_estimates = []
for t in tqdm(range(sample_number)):
    model=UnconstrainedLinearModel(L)
    temp = inference_with_transaction_costs(model, data,
    ↪  dataloader=dataloader[t:t+sample_size], epoch_num=20, lr=1)
    bootstrap_estimates.append(temp)

std_IS_tc = pd.DataFrame(pd.concat(bootstrap_estimates,
↪  axis=1).std(axis=1)).T
std_IS_tc.index = ['transaction_costs']
std_IS_tc.to_csv('standard_deviations_transaction_costs_IS.csv')
```

```
# Step I: imports
import torch
import pandas as pd
import numpy as np
from graph_representation_functions import *
from tqdm import tqdm

df_name = 'data_tilde_market_benchmark.csv'
data = pd.read_csv(df_name, index_col=0).dropna()
datelist = pd.read_csv('datelist.csv', index_col=0) # Loading list of dates
↪  to select the right index
stop_date = 19740131 # Defining a stop date to limit the training set
stop_t = datelist.loc[stop_date].values[0] # Determining the right index in
↪  the list
data = data[data.t < stop_t] # Refocusing the dataset

# Step II: instanciate dataloader
dataloader = dataloader_converter(data) # May crash

# Step III: estimate unconstrained with TC
L = 3
model = UnconstrainedLinearModel(L)
```

```
params = inference_with_transaction_costs(model, data,
↪  dataloader=dataloader, epoch_num=20, lr=0.3)
params.columns = ['transaction_costs']
params.T.to_csv('estimates_transaction_costs_OOS.csv')

# Step IV: bootstrapping with tc
L=3
sample_size = 50 # We reduce the sample size
sample_number = len(dataloader) - sample_size - 1

bootstrap_estimates = []
for t in tqdm(range(sample_number)):
    model=UnconstrainedLinearModel(L)
    temp = inference_with_transaction_costs(model, data,
    ↪  dataloader=dataloader[t:t+sample_size], epoch_num=20, lr=1)
    bootstrap_estimates.append(temp)

std_OOS_tc = pd.DataFrame(pd.concat(bootstrap_estimates,
↪  axis=1).std(axis=1)).T
std_OOS_tc.index = ['transaction_costs']
std_OOS_tc.to_csv('standard_deviations_transaction_costs_OOS.csv')
```

The following script evaluates the portfolio we build in each of these cases, according to a number of metrics.

```
"""This script aims at providing some evaluation metrics to different
↪  thetas estimates."""

import pandas as pd
from estimation import evaluation, evaluation_pos_constraint, evaluation_tc

'''Picking the right timeframe'''
data = pd.read_csv('data_tilde_market_benchmark.csv', index_col=0).dropna()
datelist = pd.read_csv('datelist.csv', index_col=0) # Loading list of dates
↪  to select the right index
start_date = 19740131 # Defining a start date to limit the training set
start_t = datelist.loc[start_date].values[0] # Determining the right index
↪  in the list
data = data[data.t >= start_t] # Refocusing the dataset

'''Loading theta estimates'''
estimates_IS = pd.read_csv('estimates_IS.csv', index_col=0)
estimates_IS.loc['paper'] = (-1.451, 3.606, 1.772)
```

```
estimates_IS.loc['market'] = (0, 0, 0)
estimates_OOS = pd.read_csv('estimates_OOS.csv', index_col=0)
estimates_OOS.loc['paper'] = (-1.124, 3.611, 3.057)
estimates_OOS.loc['market'] = (0, 0, 0)

estimates_IS_pos = pd.read_csv('estimate_pos_constraint_IS.csv',
↪  index_col=0)
estimates_IS_pos.loc['paper'] = (-1.277, 3.215, 1.416)
estimates_IS_pos.loc['market'] = (0, 0, 0)
estimates_OOS_pos = pd.read_csv('estimate_pos_constraint_OOS.csv',
↪  index_col=0)
estimates_OOS_pos.loc['paper'] = (0.651, 2.679, 3.780)
estimates_OOS_pos.loc['market'] = (0, 0, 0)

estimates_IS_tc = pd.read_csv('estimates_transaction_costs_IS.csv',
↪  index_col=0)
estimates_IS_tc.loc['no_penalty'] = estimates_IS.loc['simple_GD']
estimates_IS_tc.loc['paper'] = (-1.167, 3.160, 1.307)
estimates_IS_tc.loc['market'] = (0, 0, 0)
estimates_OOS_tc = pd.read_csv('estimates_transaction_costs_OOS.csv',
↪  index_col=0)
estimates_OOS_tc.loc['no_penalty'] = estimates_OOS.loc['simple_GD']
estimates_OOS_tc.loc['paper'] = (-0.925, 3.468, 2.497)
estimates_OOS_tc.loc['market'] = (0, 0, 0)

'''Loading theta standard deviations'''
std_IS = pd.read_csv('standard_deviations_IS.csv', index_col=0)
std_IS.loc['paper'] = (0.548, 0.921, 0.743)
std_IS.loc['market'] = (0, 0, 0)
std_OOS = pd.read_csv('standard_deviations_OOS.csv', index_col=0)
std_OOS.loc['paper'] = (0.709, 1.110, 0.914)
std_OOS.loc['market'] = (0, 0, 0)

std_IS_pos = pd.read_csv('standard_deviations_pos_constraint_IS.csv',
↪  index_col=0)
std_IS_pos.loc['paper'] = (1.217, 1.131, 1.213)
std_IS_pos.loc['market'] = (0, 0, 0)
std_OOS_pos = pd.read_csv('standard_deviations_pos_constraint_OOS.csv',
↪  index_col=0)
std_OOS_pos.loc['paper'] = (1.510, 1.417, 1.505)
std_OOS_pos.loc['market'] = (0, 0, 0)
```

```
std_IS_tc = pd.read_csv('standard_deviations_transaction_costs_IS.csv',
↪  index_col=0)
std_IS_tc.loc['no_penalty'] = std_IS.loc['simple_GD'].values
std_IS_tc.loc['paper'] = (0.550, 0.924, 0.745)
std_IS_tc.loc['market'] = (0, 0, 0)
std_OOS_tc = pd.read_csv('standard_deviations_transaction_costs_OOS.csv',
↪  index_col=0)
std_OOS_tc.loc['no_penalty'] = std_OOS.loc['simple_GD'].values
std_OOS_tc.loc['paper'] = (0.780, 1.305, 0.961)
std_OOS_tc.loc['market'] = (0, 0, 0)


'''Computing results for IS'''
results_IS = []
for elt in estimates_IS.T:
    res = evaluation(estimates_IS.T[elt].values.reshape(3,1), data=data,
    ↪  gamma=5)
    res['theta_me'] = estimates_IS.T[elt]['me']
    res['std_theta_me'] = std_IS.T[elt]['std_me']
    res['theta_btm'] = estimates_IS.T[elt]['btm']
    res['std_theta_btm'] = std_IS.T[elt]['std_btm']
    res['theta_mom'] = estimates_IS.T[elt]['mom']
    res['std_theta_mom'] = std_IS.T[elt]['std_mom']
    results_IS.append(res)
results_IS = pd.DataFrame(results_IS).T
results_IS.columns = estimates_IS.index + '_IS'

results_IS_pos = []
for elt in estimates_IS_pos.T:
    res =
    ↪  evaluation_pos_constraint(estimates_IS_pos.T[elt].values.reshape(3,1),
    ↪  data=data, gamma=5)
    res['theta_me'] = estimates_IS_pos.T[elt]['me']
    res['std_theta_me'] = std_IS_pos.T[elt]['me']
    res['theta_btm'] = estimates_IS_pos.T[elt]['btm']
    res['std_theta_btm'] = std_IS_pos.T[elt]['btm']
    res['theta_mom'] = estimates_IS_pos.T[elt]['mom']
    res['std_theta_mom'] = std_IS_pos.T[elt]['mom']
    results_IS_pos.append(res)
results_IS_pos = pd.DataFrame(results_IS_pos).T
results_IS_pos.columns = estimates_IS_pos.index + '_IS_pos'
```

```
results_IS_tc = []
for elt in estimates_IS_tc.T:
    res = evaluation_tc(estimates_IS_tc.T[elt].values.reshape(3,1),
    ↪  data=data, gamma=5)
    res['theta_me'] = estimates_IS_tc.T[elt]['me']
    res['std_theta_me'] = std_IS_tc.T[elt]['me']
    res['theta_btm'] = estimates_IS_tc.T[elt]['btm']
    res['std_theta_btm'] = std_IS_tc.T[elt]['btm']
    res['theta_mom'] = estimates_IS_tc.T[elt]['mom']
    res['std_theta_mom'] = std_IS_tc.T[elt]['mom']
    results_IS_tc.append(res)
results_IS_tc = pd.DataFrame(results_IS_tc).T
results_IS_tc.columns = estimates_IS_tc.index + '_IS_tc'

'''Computing results for OOS'''
results_OOS = []
for elt in estimates_OOS.T:
    res = evaluation(estimates_OOS.T[elt].values.reshape(3,1), data=data,
    ↪  gamma=5)
    res['theta_me'] = estimates_OOS.T[elt]['me']
    res['std_theta_me'] = std_OOS.T[elt]['std_me']
    res['theta_btm'] = estimates_OOS.T[elt]['btm']
    res['std_theta_btm'] = std_OOS.T[elt]['std_btm']
    res['theta_mom'] = estimates_OOS.T[elt]['mom']
    res['std_theta_mom'] = std_OOS.T[elt]['std_mom']
    results_OOS.append(res)
results_OOS = pd.DataFrame(results_OOS).T
results_OOS.columns = estimates_OOS.index + '_OOS'

results_OOS_pos = []
for elt in estimates_OOS_pos.T:
    res =
    ↪  evaluation_pos_constraint(estimates_OOS_pos.T[elt].values.reshape(3,1),
    ↪  data=data, gamma=5)
    res['theta_me'] = estimates_OOS_pos.T[elt]['me']
    res['std_theta_me'] = std_OOS_pos.T[elt]['me']
    res['theta_btm'] = estimates_OOS_pos.T[elt]['btm']
    res['std_theta_btm'] = std_OOS_pos.T[elt]['btm']
    res['theta_mom'] = estimates_OOS_pos.T[elt]['mom']
    res['std_theta_mom'] = std_OOS_pos.T[elt]['mom']
    results_OOS_pos.append(res)
results_OOS_pos = pd.DataFrame(results_OOS_pos).T
```

```
results_OOS_pos.columns = estimates_OOS_pos.index + '_OOS_pos'

results_OOS_tc = []
for elt in estimates_OOS_tc.T:
    res = evaluation_tc(estimates_OOS_tc.T[elt].values.reshape(3,1),
    ↪  data=data, gamma=5)
    res['theta_me'] = estimates_OOS_tc.T[elt]['me']
    res['std_theta_me'] = std_OOS_tc.T[elt]['me']
    res['theta_btm'] = estimates_OOS_tc.T[elt]['btm']
    res['std_theta_btm'] = std_OOS_tc.T[elt]['btm']
    res['theta_mom'] = estimates_OOS_tc.T[elt]['mom']
    res['std_theta_mom'] = std_OOS_tc.T[elt]['mom']
    results_OOS_tc.append(res)
results_OOS_tc = pd.DataFrame(results_OOS_tc).T
results_OOS_tc.columns = estimates_OOS_tc.index + '_OOS_tc'

'''Preparing the index in advance, otherwise it can become a bit long to
↪  modify it by hand on LateX'''

latex_index = (
    r'$\hat{\theta}_{me}$',
    r'$\sigma_{\hat{\theta}_{me}}$',
    r'$\hat{\theta}_{btm}$',
    r'$\sigma_{\hat{\theta}_{btm}}$',
    r'$\hat{\theta}_{mom}$',
    r'$\sigma_{\hat{\theta}_{mom}}$',
    r'$\abs{w_{i,t}}$',
    r'$\max{w_{i,t}}$',
    r'$\min{w_{i,t}}$',
    r'$\Sigma \mathbf{I}(w_{i,t} < 0) / N_t$',
    r'$\Sigma w_{i,t} \mathbf{I}(w_{i,t} < 0)$',
    r'$\Sigma \abs{w_{i,t+1} - w_{i,t}}$',
    r'CE',
    r'$\Bar{r}$',
    r'$\sigma(r)$',
    r'SR',
    r'$\alpha$',
    r'$\beta$',
    r'$\sigma(\epsilon)$',
    r'IR',
    r'me',
    r'btm',
```

```
    r'mom'
)

'''Concatenating and saving'''

results = pd.concat([results_IS, results_OOS], axis=1)
results = results.reindex(list(results.index[17:]) +
↪  list(results.index[:17])).applymap(lambda x: round(x, 3)) # Reordering
↪  numbers and rounding so that it becomes pretty
results.to_csv('results.csv')

# We do the same for IS and OOS individually
results_IS = results_IS.reindex(list(results_IS.index[17:]) +
↪  list(results_IS.index[:17])).applymap(lambda x: round(x, 3)) #
↪  Reordering numbers and rounding so that it becomes pretty
results_IS.to_csv('results_IS.csv')

results_IS_pos = results_IS_pos.reindex(list(results_IS_pos.index[17:]) +
↪  list(results_IS_pos.index[:17])).applymap(lambda x: round(x, 3)) #
↪  Reordering numbers and rounding so that it becomes pretty
results_IS_pos.to_csv('results_IS_pos.csv')

results_IS_tc = results_IS_tc.reindex(list(results_IS_tc.index[17:]) +
↪  list(results_IS_tc.index[:17])).applymap(lambda x: round(x, 3)) #
↪  Reordering numbers and rounding so that it becomes pretty
results_IS_tc.to_csv('results_IS_tc.csv')

results_OOS = results_OOS.reindex(list(results_OOS.index[17:]) +
↪  list(results_OOS.index[:17])).applymap(lambda x: round(x, 3)) #
↪  Reordering numbers and rounding so that it becomes pretty
results_OOS.to_csv('results_OOS.csv')

results_OOS_pos = results_OOS_pos.reindex(list(results_OOS_pos.index[17:]) +
↪  list(results_OOS_pos.index[:17])).applymap(lambda x: round(x, 3)) #
↪  Reordering numbers and rounding so that it becomes pretty
results_OOS_pos.to_csv('results_OOS_pos.csv')

results_OOS_tc = results_OOS_tc.reindex(list(results_OOS_tc.index[17:]) +
↪  list(results_OOS_tc.index[:17])).applymap(lambda x: round(x, 3)) #
↪  Reordering numbers and rounding so that it becomes pretty
results_OOS_tc.to_csv('results_OOS_tc.csv')
```

```
'''Exporting to LateX (print)'''

results_IS.index = latex_index
results_OOS.index = latex_index
results_IS_pos.index = latex_index
results_OOS_pos.index = latex_index
results_IS_tc.index = latex_index
results_OOS_tc.index = latex_index

for elt in (results_IS, results_OOS, results_IS_pos, results_OOS_pos,
↪  results_IS_tc, results_OOS_tc):
    print(
        elt.to_latex()\
        .replace('textbackslash ','')\
        .replace('\\_','_')\
        .replace('\\{','{')\
        .replace('\\}','}')\
        .replace('\\$','$')\
        .replace('\\toprule','\\hline')\
        .replace('\\midrule','\\hline')\
        .replace('\\bottomrule','\\hline')\
        .replace(r'{}','Variable')\
        .replace('0.000','-')\
        .replace('--','-')
        )
```

The following script evaluates the different training times between GD methods where numerical and analytical gradients are used.

```
from functions import *
import numpy as np
import time, copy

from graph_representation_functions import *
import torch
import torch.nn as nn

data = pd.read_csv('data_tilde_market_benchmark.csv', index_col=0).dropna()

L = 3 # Three features
eta = 200 # Same learning rate
epoch_num = 100 # Same number of epochs
h = 0.1
```

```
x0 = np.zeros((L,1)) # Same starting point

fun = lambda x : -u(x, data=data)
der = lambda x : -u_der(x, data=data)

t1 = time.time()
x_numerical = copy.deepcopy(x0)
for _ in range(epoch_num):
    grad = (fun(x_numerical + np.eye(L) * h) - fun(x_numerical))/h #
    ↪ Numerical gradient
    x_numerical -= (eta * grad).reshape(L,1)
t2 = time.time()

t3 = time.time()
x_analytical = copy.deepcopy(x0)
for _ in range(epoch_num):
    grad = der(x_analytical) # Analytical gradient
    x_analytical -= (eta * grad).reshape(L,1)
t4 = time.time()


model = UnconstrainedLinearModel(3)
optimizer = torch.optim.SGD(model.parameters(), lr=0.075)
dataloader = dataloader_converter(data, compute_state_transition=False)

t5 = time.time()

for _ in tqdm(range(epoch_num)):
    loss=0
    for d in dataloader:
        x, w_bar, r_next = d['x'], d['w_bar'], d['r_next']
        w = model(x, w_bar)
        loss = loss + negU(w, r_next, gamma=5)
    model.zero_grad()
    loss.backward()
    optimizer.step()

t6 = time.time()

x_torch = nn.utils.parameters_to_vector(model.parameters()).detach().numpy()

df = pd.DataFrame(
```

```
    np.concatenate([x_numerical.T[0], x_analytical.T[0],
    ↪  x_torch]).reshape(3,3).T,
    index = ('me','btm','mom'),
    columns = ('Numerical', 'Analytical (pandas/numpy)', 'Analytical
    ↪  (PyTorch)')
    )
df.loc['Time elapsed'] = (t2-t1, t4-t3, t6-t5)
df = df.round(3)
df.to_csv('analytical_v_numerical_gradient.csv')
print(
    df.to_latex()\
        .replace(r'{}','Variable')\
        .replace(r'toprule','hline')\
        .replace(r'midrule','hline')\
        .replace(r'bottomrule','hline')
    )
```

# Bibliography


[1] Pedro Santa-Clara Michael W. Brandt and Rossen Valkanov. Parametric portfolio policies: Exploiting characteristics in the cross-section of equity returns. *Review of Financial Studies*, 22:3411–3447, 2009.

[2] Lars P. Hansen. Large sample properties of generalized method of moments estimators. *Econometrica*, 50:1029–1054, 1982.

[3] David G. Booth Eugene F. Fama, Kenneth R. French and Rex Sinquefield. Differences in the risks and returns of nyse and nasd stocks. *Financial Analysts Journal*, 41:37–41, 1993.

[4] Mark. M. Carhart. On persistance in mutual fund performance. *The Journal of Finance*, 52:57–82, 1997.

[5] Donald B. Keim and Ananth Madhavan. Transactions costs and investment style: An inter-exchange analysis of institutional equity trades. *The Journal of Financial Economics*, 46:265–292, 1997.

[6] Jack Glen Ian Domowitz and Ananth Madhavan. Liquidity, volatility, and equity trading costs across countries and over time. *International Finance*, 4:221–255, 2001.

[7] Joel Hasbrouck. Trading costs and returns for us equities: Estimating effective costs from daily data. *Journal of Finance*, 64:1445–1477, 2006.