Project 1

Reproduce Key Results in a ML-In-Finance Study

Joshua Gloor

13.10.2025

We always mean KELLY et al. (2024) when we refer to results or sections from a paper.

For reproducibility, we fix the pseudo-random generator:

```
rng = np.random.default_rng(seed=42)
```

Task 1: Misspecified Simulation With Ridge

We will first show and justify our parameter choices. Next, we elaborate on how we performed the simulation and data generation. Then, we show the VoC plots produced using our code. In the last subsection, we explain the idea behind our implementation of Ridge regression.

Choice of Parameters

The exercise description fixes $c = 10$. Since we have $c = \frac{P}{T_{\text{tr}}}$, this gives us the following formula for the number of features:

$$P = 10 \cdot T_{\text{tr}}.$$

That in turn means that P is fixed after we choose the training set size.

Initially we wanted to choose $T_{\text{tr}} = 1000$ because in section VC of the paper, P as high as 12000 are shown. With $T_{\text{tr}} = 1000$ and thus $P = 10 \cdot T_{\text{tr}} = 10000$, we would be close to that number.

However, this results in complexity we cannot support on our machine, especially for multiple simulation runs. The reason for this is the $\mathcal{O}(T_{\text{tr}}^3)$ runtime complexity of SVD. We ended up opting for a reduction of the training set size by an order of magnitude. We choose $T_{\text{te}} = T_{\text{tr}}$ for simplicity and to generate more meaningful plots.

For the shrinkage grid, z, and cq, we follow the exercise's suggestions.

Below, we initialize the parameters that stay fixed for both simulation runs of task 1 and task 2:

```
T_tr = 100
T_te = T_tr
```

```
T = T_tr + T_te
c = 10
P = c * T_tr
b_star = 0.2  # As in paper's theoretical section
zs = np.array([0, 0.1, 0.25, 0.5, 1, 2, 5, 10, 25, 50, 100])
cqs = np.array(
    [0.5, 0.75, 0.9, 0.95, 0.98, 1.02, 1.05, 1.1, 1.25, 1.5, 2, 3, 5, 7, 10]
)
```

Simulation and Data Generation Details

We first explain the data generation process and after that we elaborate on how the simulation code uses it to coordinate Monte Carlo experiments across different parameter settings.

The data generation logic, which can be found at `src/simulation/data_generation.py`, constructs synthetic signal and corresponding return datasets. The `generate_run` method generates one simulation run. As stated in the exercise description, it uses helper methods to perform the following steps in order:

- Generate S with $\Psi = I$, with rows $S_t \sim \mathcal{N}(0, I_P)$.
- Generate a dense β^* by normalizing it to $\left\|\beta^*\right\|_2^2 = b^*$.
- Generate returns as $S\beta^* + \epsilon$, where $\epsilon = \left[\epsilon_{t+1,1}, \epsilon_{t+1,2}, \ldots, \epsilon_{t+1,T}\right]'$ and $\epsilon_{t+1,i} \sim \mathcal{N}(0,1)$.
- Generate a permutation that is later used together with the method `observed_block` to get a random permutation of columns of S once per simulation run.

The simulation logic can be found at `src/simulation/simulator.py`. We use the `SimConfig` class to ensure that we do not accidentally reassign or update key parameters during the simulation run. In the `Simulator` class, the main entry point is the `run_simulation` method. That method then executes `n_runs` many Monte Carlo experiments using the internal method `_simulate_one_run`. For a single run, we perform the following main steps:

- We call the data generation code that returns simulated training and test data $S_{\text{tr}}, R_{\text{tr}}, S_{\text{te}}, R_{\text{te}}$, together with a permutation vector, `pem`.
- We sweep over cq. For each, we:
  - Calculate P_1 and get the respective observed part of the training and test data.
  - We fit the model. We fit the model here because as we will explain later, we can avoid recomputation of the SVD for different shrinkage values.
  - We sweep over z, for each, we:
    - Use the model to make a prediction.
    - Calculate the four metrics.

Other than the main logic, the `Simulator` class also includes some helper methods to compute the average metrics across the simulation runs. Another method helps with transforming the metrics grid into a better data structure used in plotting.

VoC Curves of the Simulation

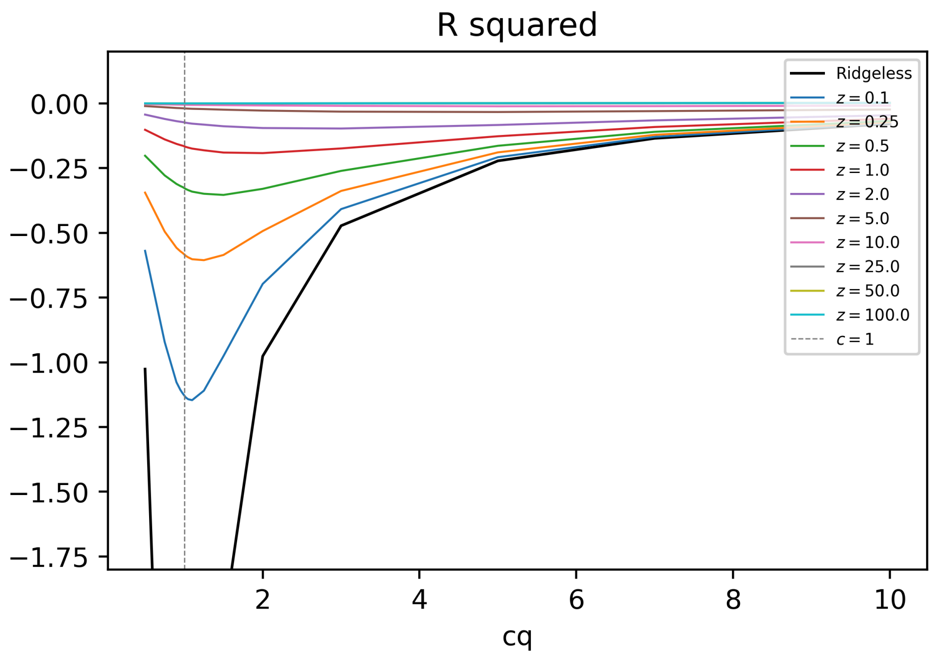
Next, we perform the simulation. If we want less noisy curves, we can always increase the number of simulation runs. However, of course this also results in an increase in runtime.

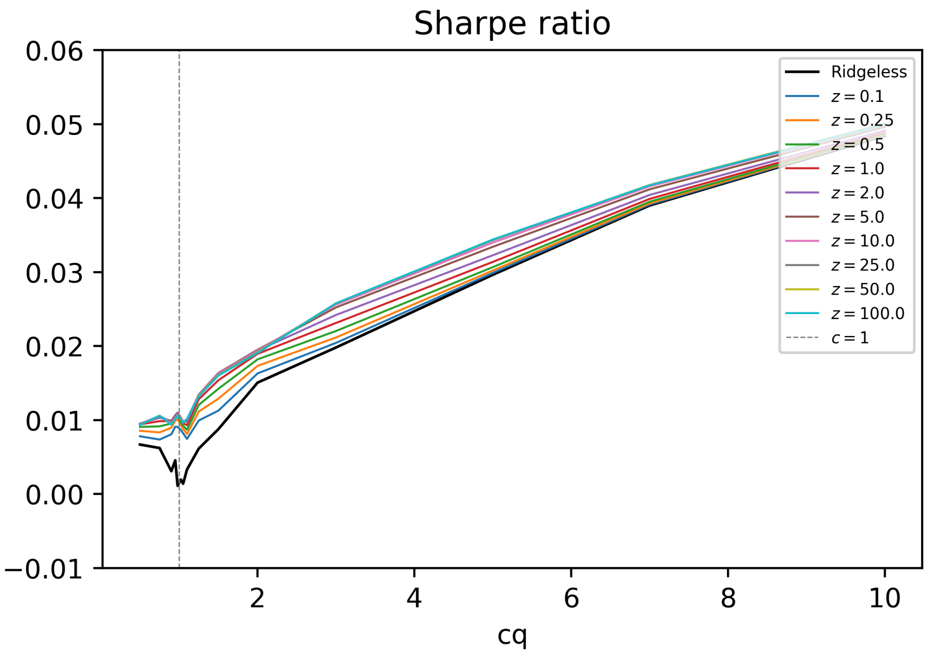
```
from src.simulation.simulator import SimConfig, Simulator
from src.models.factory import create_model

n_runs = 1500  # Number of runs (1500+ produces good results for ridge)
cfg = SimConfig(n_runs, T_tr, T_te, P, b_star, cqs, zs) # Freezing our parameters
model = create_model("ridge")  # Uses our Ridge implementation
simulator = Simulator(cfg, model, rng)
metrics = simulator.run_simulation()
```

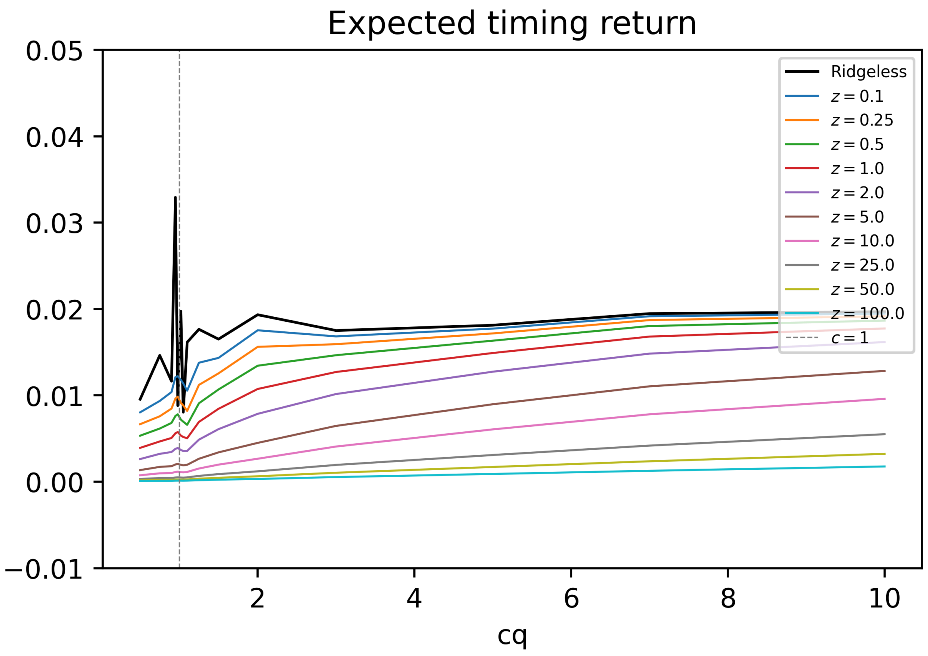
Below, we plot the suggested metrics from the exercise description. Note that we hide some code logic to avoid cluttering the report. However, it can be found in the `.qmd` file. Also, we show the method call for the plot only once since it is analogous for the other plots.

```
m = "r2"
plot_curves(metrics, m, cqs, metric_to_ylim[m], metric_to_title[m])
```

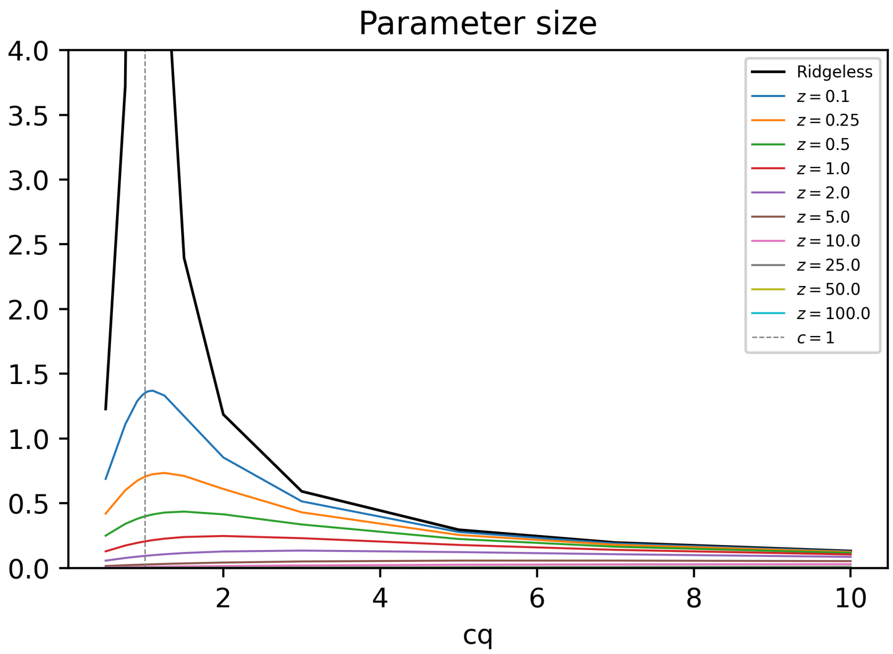




Sharpe ratio
0.06
0.05
0.04
0.03
0.02
0.01
0.00
−0.01
2
4
6
8
10
cq
Ridgeless
$z = 0.1$
$z = 0.25$
$z = 0.5$
$z = 1.0$
$z = 2.0$
$z = 5.0$
$z = 10.0$
$z = 25.0$
$z = 50.0$
$z = 100.0$
$c = 1$



Expected timing return
0.05
0.04
0.03
0.02
0.01
0.00
−0.01
2
4
6
8
10
cq
Ridgeless
$z = 0.1$
$z = 0.25$
$z = 0.5$
$z = 1.0$
$z = 2.0$
$z = 5.0$
$z = 10.0$
$z = 25.0$
$z = 50.0$
$z = 100.0$
$c = 1$



We can clearly see the VoC curves as discussed in the paper. We can also see that our Ridge implementation likely suffers slightly from numerical issues for the Ridgeless case. Since they are not major and are smoothed out over a sufficient number of MC runs, we did not modify our implementation further.

Implementation of Ridge Regression

In this section, we explain the main idea behind our Ridge implementation, which can be found in `src/models/ridge.py`.

We start the derivation from the matrix representation of the Ridge estimator given in the exercise:

$$\hat{\beta}(z) = \Big(X'X + \underbrace{zT_{\text{tr}}}_{\triangleq\lambda} I\Big)^{-1} X'y$$

Leveraging SVD, i.e., $X = U\Sigma V'$, we rewrite it as follows:

$$\begin{aligned}\hat{\beta}(z) &= (U\Sigma V')'(U\Sigma V') + \lambda I)^{-1}(U\Sigma V')'y \\ &= \Big(V\Sigma'\underbrace{U'U}_{=I}\Sigma V' + \lambda I\Big)^{-1}(V\Sigma' U')y \\ &= (V\Sigma'\Sigma V' + \lambda I)^{-1}(V\Sigma' U')y\end{aligned}$$

The trick is realizing that $V\Sigma'\Sigma V'$ represents a diagonalization. Next, we show that if a matrix A is diagonalizable, i.e., $A = PDP^{-1}$ with a diagonal matrix D, and $-\lambda$ is not an eigenvalue of A, we have:

$$(A+\lambda I)^{-1} = P(D+\lambda I)^{-1}P^{-1} = P\ \mathrm{diag}\left(\frac{1}{D_{ii}+\lambda}\right)P^{-1},$$

i.e., if we know P and P^{-1}, we can rewrite the inverse of the addition of A and a scaled identity matrix using another diagonal matrix which has diagonal entries equal to the reciprocal of the added diagonal terms.

We show this directly:

$$\begin{aligned}
& A+\lambda I = PDP^{-1} + \lambda PP^{-1} \\
\Rightarrow\quad & (A+\lambda I)P = PD + \lambda P \\
\Rightarrow\quad & P^{-1}(A+\lambda I)P = (D+\lambda I) \\
\Rightarrow\quad & \left(P^{-1}(A+\lambda I)P\right)^{-1} = (D+\lambda I)^{-1} \\
\Rightarrow\quad & P^{-1}(A+\lambda I)^{-1}P = (D+\lambda I)^{-1} \\
\Rightarrow\quad & (A+\lambda I)^{-1} = P(D+\lambda I)^{-1}P^{-1},
\end{aligned}$$

where we used the second assumption that $-\lambda$ is not an eigenvalue of A in the fourth line where we take the inverse of the matrices on both sides.

In our specific case, $-\lambda$ is not an eigenvalue of $V\Sigma'\Sigma V'$ because it is a positive semidefinite matrix. It is positive semidefinite, because we can write it as a product, $(\Sigma V')'(\Sigma V')$. Since $V\Sigma'\Sigma V'$ is positive semidefinite, it follows that all its eigenvalues are non-negative.

We use the above result to continue our derivation:

$$\begin{aligned}
\hat{\beta}(z) &= \big(V \underbrace{\Sigma'\Sigma}_{=\,\mathrm{diag}\,(\sigma_i^2)} V' + \lambda I\big)^{-1}(V\Sigma'U')y \\
&= V\big(\mathrm{diag}\,(\sigma_i^2) + \lambda I\big)^{-1}V'(V\Sigma'U')y \\
&= V\ \mathrm{diag}\left(\frac{1}{\sigma_i^2+\lambda}\right)\underbrace{V'V}_{=I}\Sigma'U'y \\
&= V\ \mathrm{diag}\left(\frac{1}{\sigma_i^2+\lambda}\right)\Sigma'U'y \\
&= V\ \mathrm{diag}\left(\frac{\sigma_i}{\sigma_i^2+\lambda}\right)U'y,
\end{aligned}$$

where $\sigma_i = \Sigma_{ii}$ is the i-th singular value.

This formula allows us to compute the SVD once for a given cq, and afterwards we can iterate over the shrinkage grid very efficiently. Instead of having to calculate the SVD every time, we only have to perform matrix multiplication, replacing λ in the formula above with $T_{\mathrm{tr}}z$.

Task 2: New Model on the Same Data

We decided to go with a small, fixed, single-hidden-layer neural net instead of a LASSO based approach because our β^* is dense by construction. The paper also discourages the use of LASSO in footnote 22 on page 470.

The implementation can be found in `src/models/mlp.py`. We used an abstract base class, `src/models/base_model.py`, to standardize the interface for both Ridge and the MLP so we can reuse the same simulation code.

One way we can make the multilayer perceptron (MLP) compare to the ridge simulation is by also applying shrinkage. In the MLP case, we use weight decay, which is a regularization technique that penalizes large weights by adding an L_2 norm penalty of the weights to the loss function.

The main problem we face with a small neural net is that we cannot use our trick from Ridge to more efficiently compute each z for a given cq. We are forced to retrain the whole model for each z.

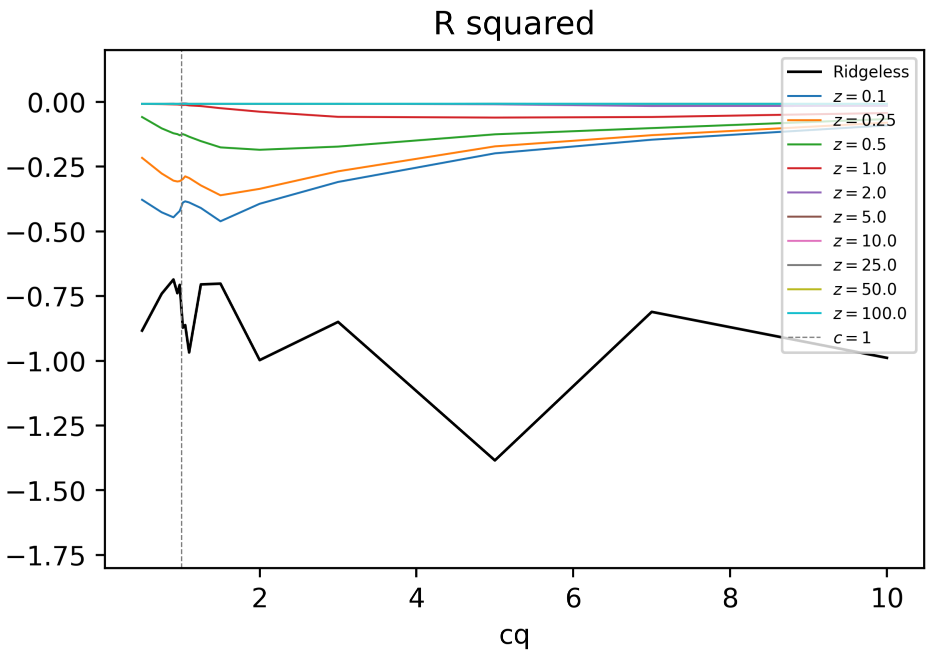
While increasing the number of simulation runs also smooths the curves, we were forced to choose a smaller number of runs because of hardware limitations. Apart from `n_runs`, the rest of the parameters are the same as for task 1.

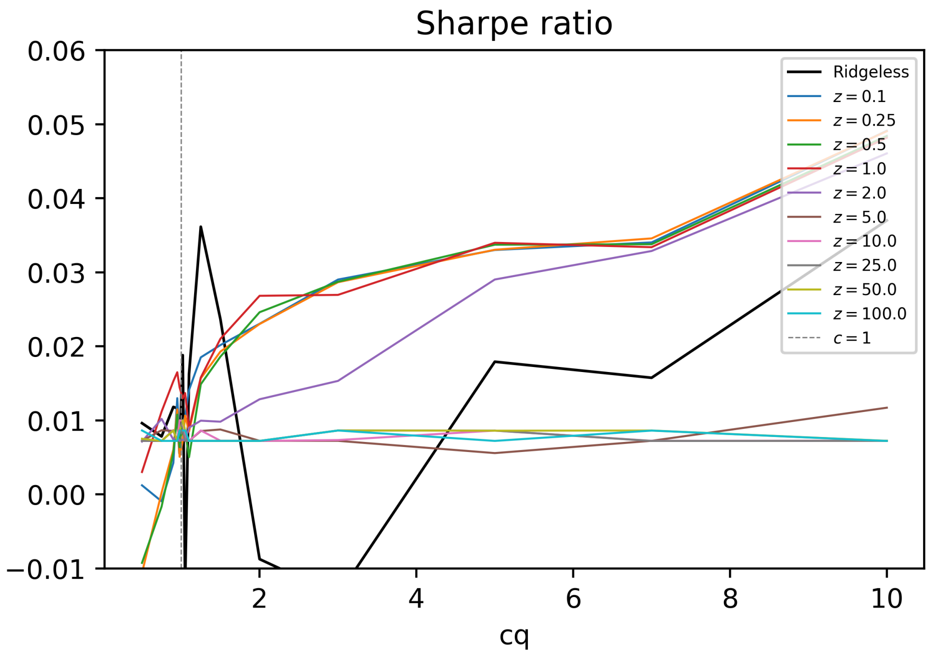
Due to the lower number of MC runs, we naturally expect considerably noisier curves.

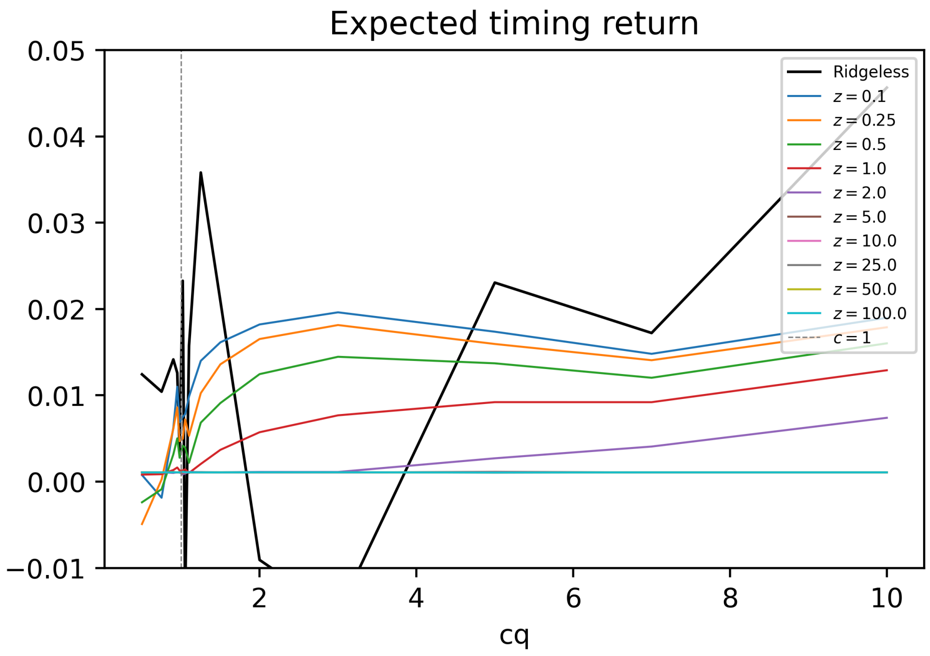
```
n_runs = 50  # Number of runs (50+ produce half-decent results)
cfg = SimConfig(n_runs, T_tr, T_te, P, b_star, cqs, zs)
model = create_model("mlp")  # Uses our MLP
simulator = Simulator(cfg, model, rng)
metrics = simulator.run_simulation()
```

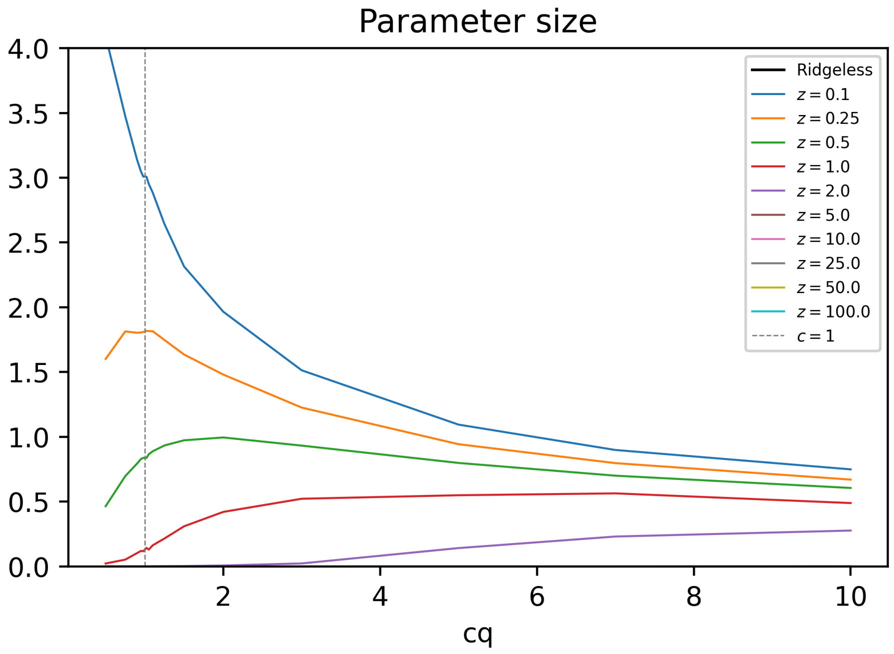
We reuse the same plotting functions as we used in task 1. We again show only one code snippet since the other plots are generated analogously.

```
m = "r2"
plot_curves(metrics, m, cqs, metric_to_ylim[m], metric_to_title[m])
```


R squared
0.00
−0.25
−0.50
−0.75
−1.00
−1.25
−1.50
−1.75
2
4
6
8
10
cq
Ridgeless
$z = 0.1$
$z = 0.25$
$z = 0.5$
$z = 1.0$
$z = 2.0$
$z = 5.0$
$z = 10.0$
$z = 25.0$
$z = 50.0$
$z = 100.0$
$c = 1$


Sharpe ratio
0.06
0.05
0.04
0.03
0.02
0.01
0.00
−0.01
2
4
6
8
10
cq
Ridgeless
$z = 0.1$
$z = 0.25$
$z = 0.5$
$z = 1.0$
$z = 2.0$
$z = 5.0$
$z = 10.0$
$z = 25.0$
$z = 50.0$
$z = 100.0$
$c = 1$





As mentioned earlier, we see noisier plots.

However, we observe that the curves do not deviate all that much from the Ridge benchmarks. We think this is because we also impose regularization using the weight decay method.

Task 3: Predictions on Three Data Sets

For this task, we first performed some very basic analysis using the command line to find out the number of features and samples for each file.

We summarize the results in the following table:

file	features	samples
`pairA_train`	600	600
`pairB_train`	2400	240
`pairC_train`	1800	360

This shows us that while, for `pairA`, we have the same number of features as samples, for `pairB` and `pairC` we have features $\gg$ samples.

Our initial approach was to perform time series cross validation over a wide shrinkage grid. However, we noticed that the largest z was always selected, which in turn leads to all `yhat` values being near zero. Our idea to fix this issue was to add some weights to the calculation of the means over the different splits. The idea being that later splits, which have longer training windows, are more important. The weights seemed to work for `pairA`, unfortunately the issue remained for `pairB` and `pairC`.

For the remaining two, we observed that there was barely any improvement in the Sharpe ratio for different shrinkage levels. As stated in Appendix B of the exercise, this could be the result of a sparse β^*.

Based on that observation and the hint in the exercise, we changed the model for `pairB` and `pairC` to LASSO. However, we were unable to get LASSO to work properly. We observed that all values for `yhat` were the same.

Running out of ideas and time, we reverted to Ridge and reduced the maximum value of the shrinkage grid. This will unlikely produce good results but avoids `yhat` values being too close to zero.

The code to arrive at our predictions can be executed by running: `python -m src.task3_with_my_ridge` from the root of the repository.

Bibliography

KELLY, B., MALAMUD, S., & ZHOU, K. (2024). The Virtue of Complexity in Return Prediction. *The Journal of Finance*, *79*(1), 459–503. https://doi.org/https://doi.org/10.1111/jofi.13298