Lab 002

Cross validation and simulation

Edward Rubin
24 January 2020

Admin

Admin

In lab today

- Check in
- Cross validation and parameter tuning (with `caret`)
- Simulation

Upcoming

- Keep up with readings (*ISL* 3–4, 6)
- Assignment will be posted soon

Check in

Some questions

(Be honest)

1. How is **this class** going?
    - Are there areas/topics you would like more coverage/review?
    - How is the speed?
    - How were the assignments?
    - Is there more we or you could be doing?
2. How is your **quarter** going?
3. How is your **program** going?
4. **Anything** else?

Cross validation

Cross validation

Review

Cross validation[†] (CV) aims to estimate out-of-sample performance

1. **efficiently**, using the full dataset to both *train* and *test* (validate)
2. **without overfitting**, separating observations used to *train* or *test*

CV (*e.g.*, LOOCV and k-fold) aids with several tasks

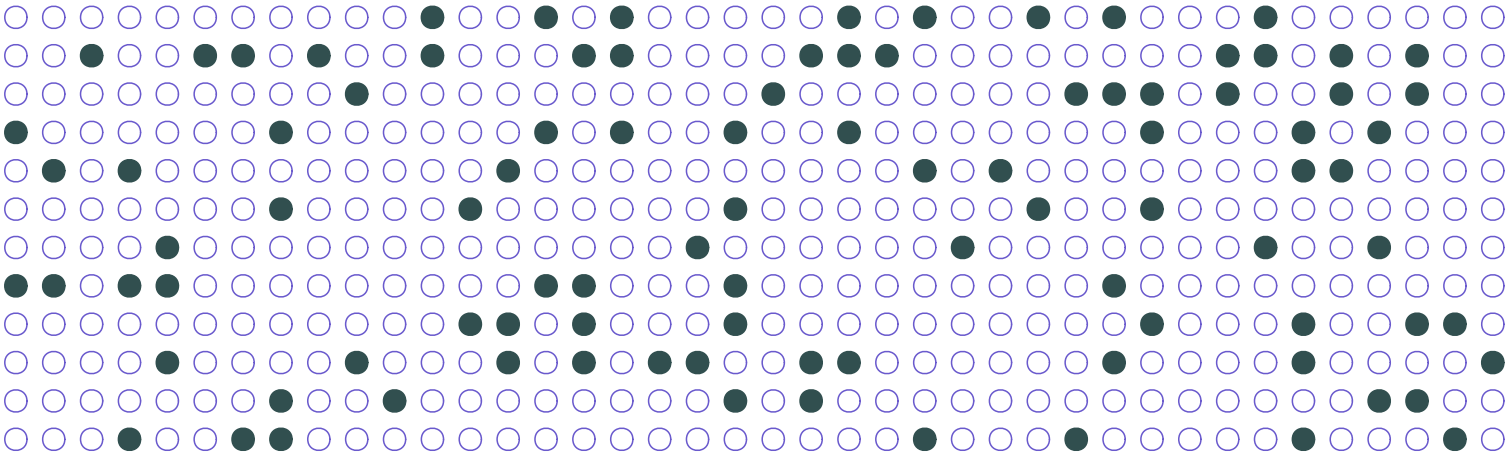
- **model selection**: choosing key parameters for a model's flexibility
  e.g., *K* in KNN, polynomials and interactions for regression (*AKA* tuning)
- **model assessment**: determining how well the model performs
  e.g., estimating the true test MSE, MAE, or error rate

† Plus other resampling (and specifically hold-out) methods

Cross validation

k-fold cross validation, review

1. **Divide** training data into k folds (mutally exclusive groups)
2. *For each fold i*:
    - **Train** your model on all observations, excluding members of i
    - **Test** and **assess** model_i on the members of fold i, *e.g.*, MSE_i
3. **Estimate test performance** by averaging across folds, *e.g.*, $\text{Avg}(\text{MSE}_i)$

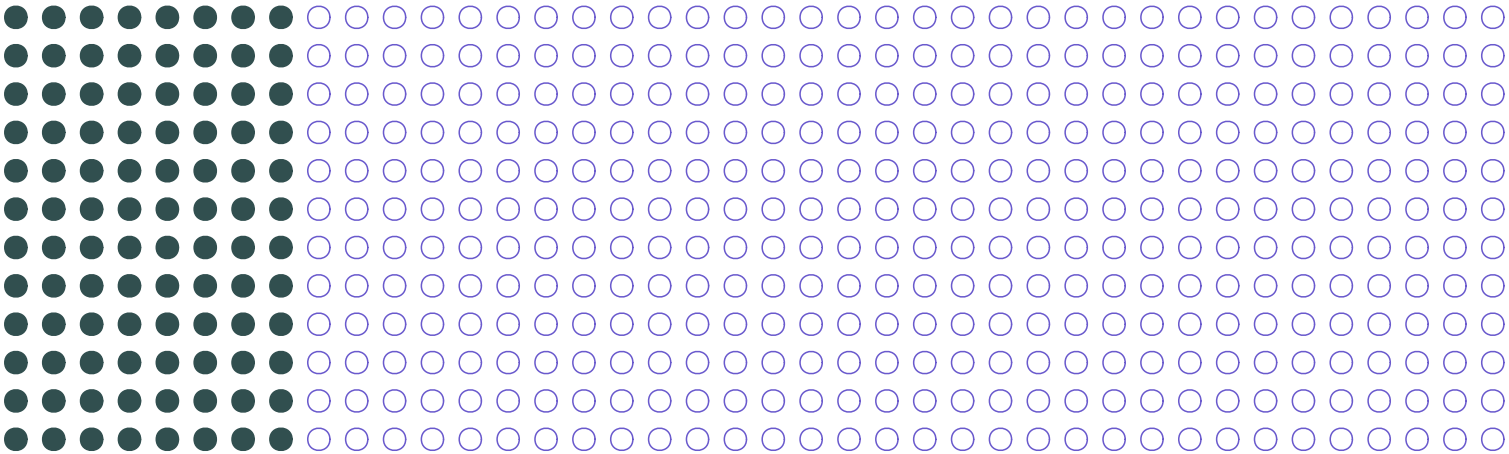


Each fold takes a turn at **validation**. The other $k - 1$ folds **train**.

Cross validation

k-fold cross validation, review

1. **Divide** training data into k folds (mutally exclusive groups)
2. *For each fold i*:
    - **Train** your model on all observations, excluding members of i
    - **Test** and **assess** model_i on the members of fold i, *e.g.*, MSE_i
3. **Estimate test performance** by averaging across folds, *e.g.*, $\text{Avg}(\text{MSE}_i)$

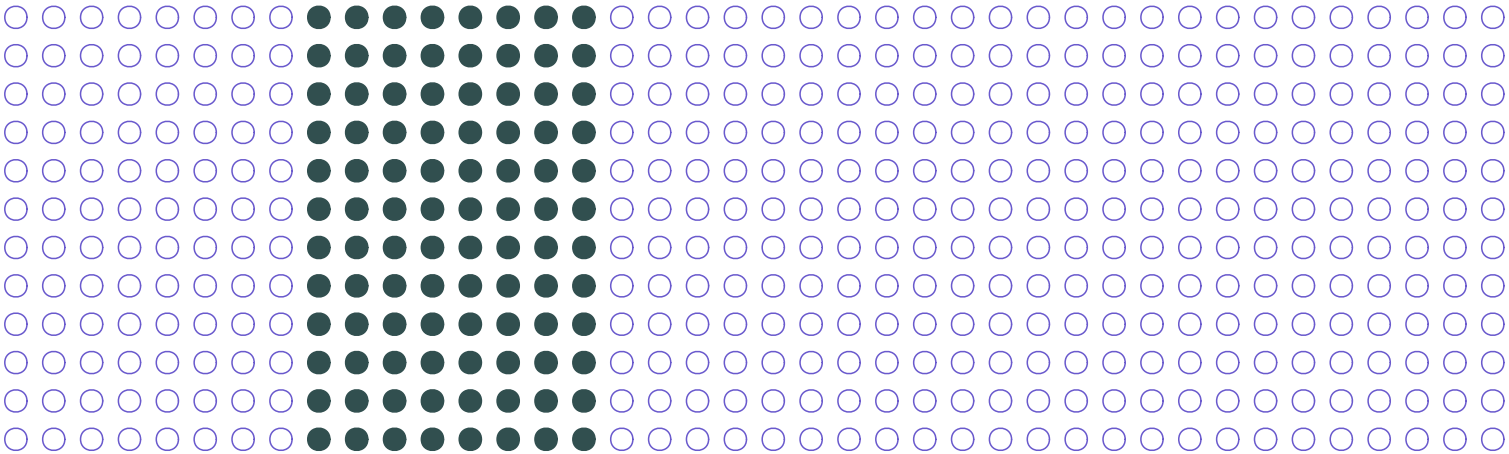


For $k = 5$, fold number 1 as the **validation set** produces $\text{MSE}_{k=1}$.

Cross validation

k-fold cross validation, review

1. **Divide** training data into k folds (mutally exclusive groups)
2. *For each fold i*:
    - **Train** your model on all observations, excluding members of i
    - **Test** and **assess** model_i on the members of fold i, *e.g.*, MSE_i
3. **Estimate test performance** by averaging across folds, *e.g.*, $\text{Avg}(\text{MSE}_i)$

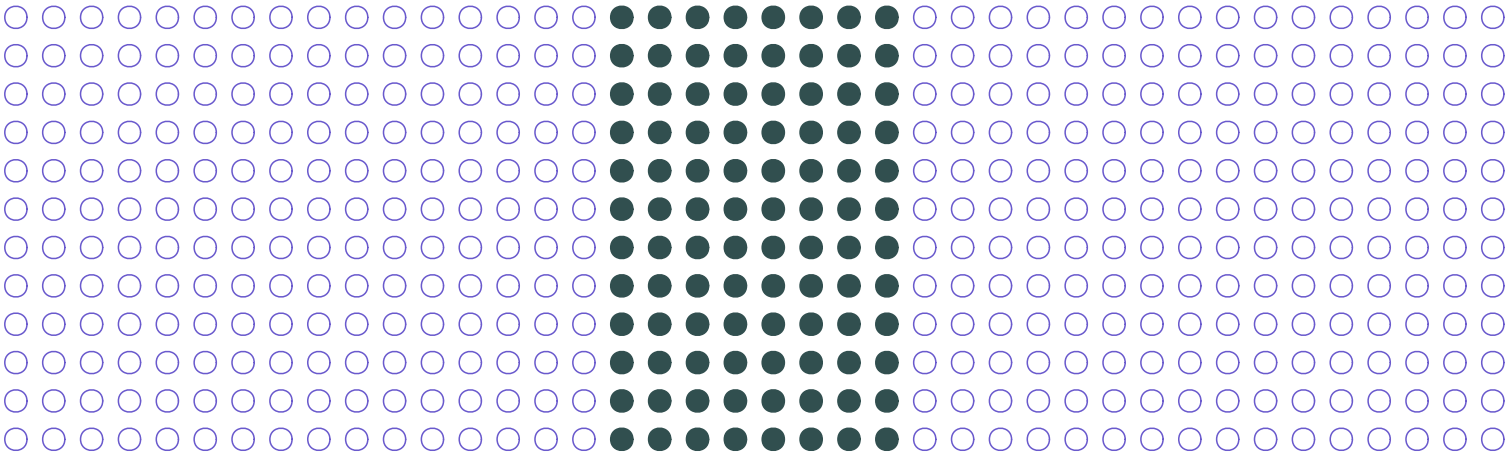


For $k = 5$, fold number 2 as the **validation set** produces $\text{MSE}_{k=2}$.

Cross validation

k-fold cross validation, review

1. **Divide** training data into k folds (mutally exclusive groups)
2. *For each fold i*:
    - **Train** your model on all observations, excluding members of i
    - **Test** and **assess** model_i on the members of fold i, *e.g.*, MSE_i
3. **Estimate test performance** by averaging across folds, *e.g.*, $\text{Avg}(\text{MSE}_i)$

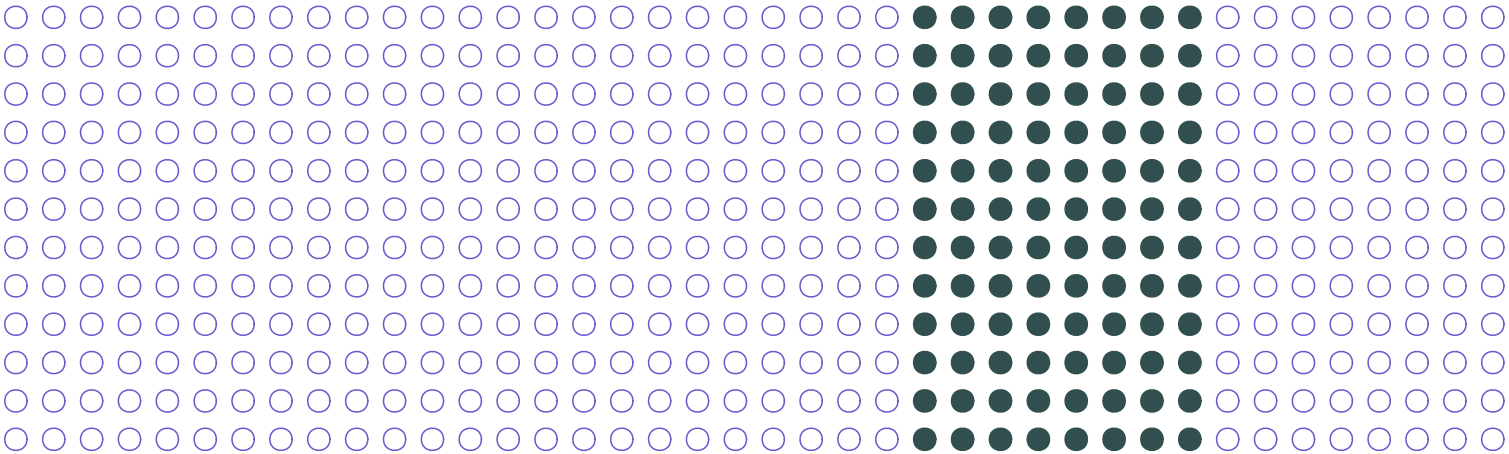


For $k = 5$, fold number 3 as the **validation set** produces $\text{MSE}_{k=3}$.

Cross validation

k-fold cross validation, review

1. **Divide** training data into k folds (mutally exclusive groups)
2. *For each fold i*:
    - **Train** your model on all observations, excluding members of i
    - **Test** and **assess** model_i on the members of fold i, *e.g.*, MSE_i
3. **Estimate test performance** by averaging across folds, *e.g.*, $\text{Avg}(\text{MSE}_i)$

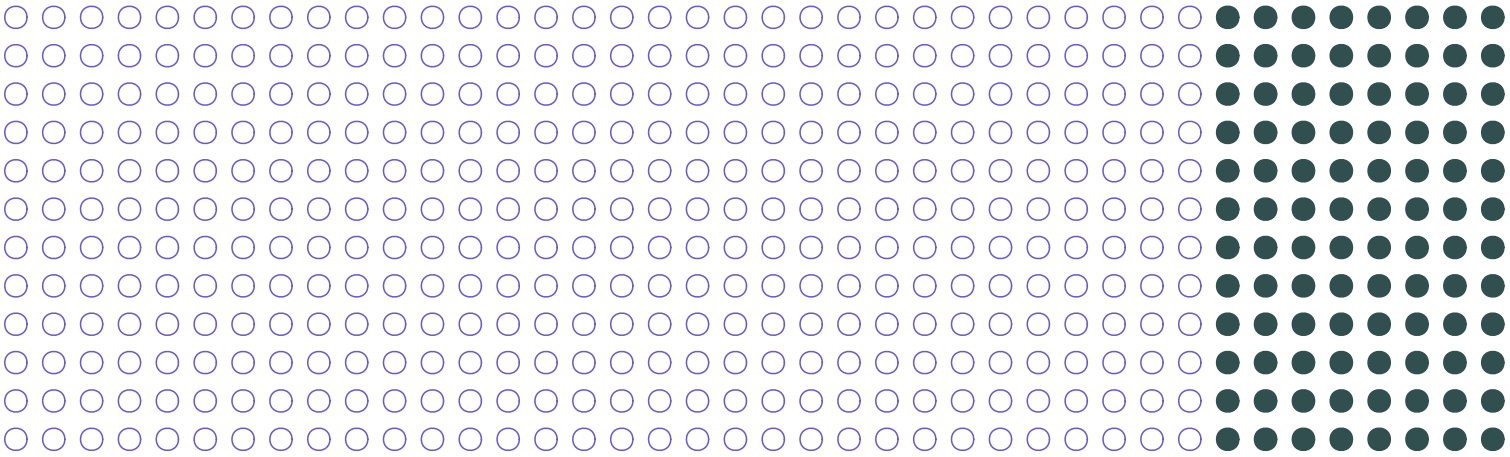


For $k = 5$, fold number 4 as the **validation set** produces $\text{MSE}_{k=4}$.

Cross validation

k-fold cross validation, review

1. **Divide** training data into k folds (mutally exclusive groups)
2. *For each fold i*:
    - **Train** your model on all observations, excluding members of i
    - **Test** and **assess** model_i on the members of fold i, *e.g.*, MSE_i
3. **Estimate test performance** by averaging across folds, *e.g.*, $\text{Avg}(\text{MSE}_i)$



For $k = 5$, fold number 5 as the **validation set** produces $\text{MSE}_{k=5}$.

Cross validation

Independence

Resampling methods assume something similar to independence: our resampling must match the original sampling procedure.

If observations are "linked" but we resample independently, CV may break.

If we have **repeated observations** on individuals i through time t:

- It's pretty likely $y_{i,t}$ and $y_{i,t+1}$ are related—and maybe $y_{i,t+\ell}$.
- Initial sample may draw individuals i, but *standard* CV ignores time.

In other case, **some individuals are linked with other individuals**, *e.g.*,

- $y_{i,t}$ and $y_{j,t}$ my be correlated if i and j live togother
- Also: $y_{i,t}$ and $y_{j,t+\ell}$ could be correlated

Cross validation

Independence

In other words: Spatial or temporal **dependence** between observations **breaks the separation between training and testing samples**.

Breaking this separation train-test separation leads us back to

- **Overfitting** the sample—since training and testing samples are linked
- **Overestimating model performance**—the estimated test MSE will be more of a training MSE.

Solutions to this problem involve **matching the resampling process** to the **original sampling and underlying dependence**.

Examples: Spatial k-fold CV (SKCV) and blocked time-series CV

Cross validation

Dependence

Q So how big of a deal is this type of depedence?

A Let's see! (Sounds like it's time for a simulation.)

Simulations

Simulations

Monte Carlo

Monte Carlo simulation[†] allows us model probabilistic questions.

1. **Generate a population** defined by some data-generating process (DGP), a model of how your fake/simulated world works.
2. Repeatedly **draw samples** s from your population. *For each* s:
   - Apply the relevant methods, sampling techniques, estimators, *etc.*
   - Record your outcome(s) of interest (*e.g.*, MSE or error rate), O_s
3. Use the **distribution** of the O_s to *learn* about your methods, sampling techniques, and/or estimators—the mean, bias, variability *etc.*

Ex_1 We can change DGP in (1) to see how performance (3) changes.

Ex_2 Holding DGP (1) constant, we can compare competing methods (2).

† Also called Monte Carlo methods, experiments, *etc.*

Sound familiar? Monte Carlo is very related to resampling methods.

Introductory example: Define the question

It's always helpful to clearly define the question you want to answer, *e.g.*:

Example question: Is OLS unbiased when $\varepsilon_i \sim$ Uniform(0, 1)?

Now we know our goal: Determine unbiasedness (mean of distribution).

Introductory example: DGP and population

We'll use the DGP $y_i = \mathbf{3} + \mathbf{6}x_i + \varepsilon_i$,
where $\varepsilon_i \sim$ Uniform(0, 1), and $x_i \sim$ N(0, 1).

```
# Set seed
set.seed(123)
# Define population size
pop_size = 1e4
# Generate population
ols_pop = tibble(
  # Generate disturbances: Using Uniform(0,1)
  e = runif(pop_size, min = 0, max = 1),
  # Generate x: Using N(0,1)
  x = rnorm(pop_size),
  # Calculate y
  y = 3 + 6 * x + e
)
```

Introductory example: A single iteration

Now we want to write a function that

1. **samples** from the population `ols_pop`
2. **estimates OLS** regression on the sample
3. **records** the OLS estimate

```
ols_function = function(iter, sample_size) {
  # Draw a sample of size 'sample_size'
  ols_sample = sample_n(ols_pop, size = sample_size)
  # Estimate OLS
  ols_est = lm(y ~ x, data = ols_sample)
  # Return coefficients and iteration number
  data.frame(b0 = coef(ols_est)[1], b1 = coef(ols_est)[2], iter)
}
```

Introductory example: Run the simulation

For the simulation, we run our single-iteration function `ols_function()` **a bunch** of times (like 10,000) and then collect the results.†

```
# Set up parallelization (with 12 cores)
# Warning: Can take a little time to run (esp w/out 12 cores)
plan(multiprocess, workers = 12)
# Set seed
set.seed(12345)
# Run simulation
ols_sim = future_lapply(
  X = 1:1e4,
  FUN = ols_function,
  sample_size = 50,
  future.seed = T
) %>% bind_rows()
```

† The `apply()` family and parallelization are both key here (`future.apply` combines them).

Introductory example: Results

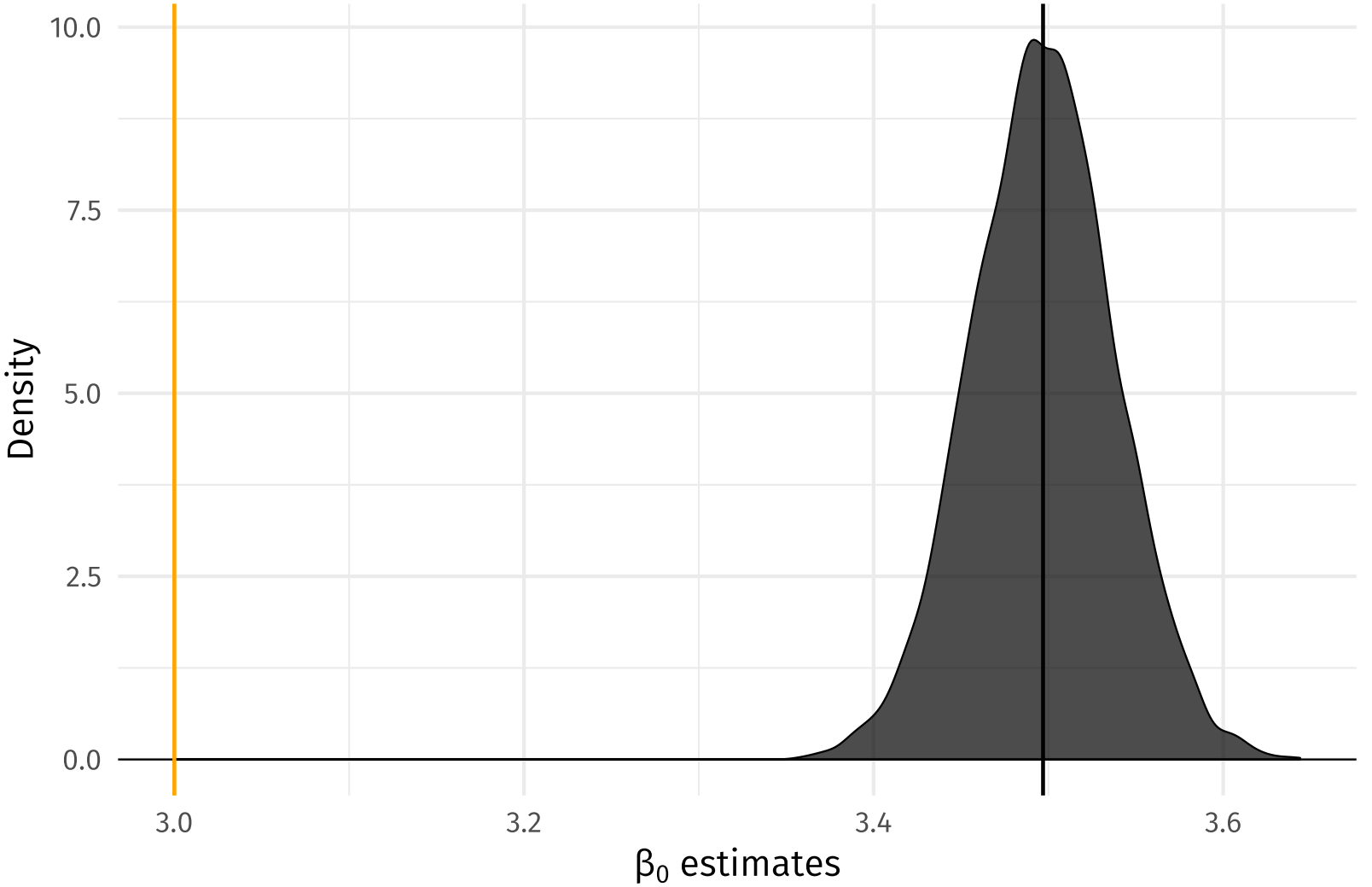
Now we're ready to summarize the results.

We wanted to know **if OLS is still unbiased**.

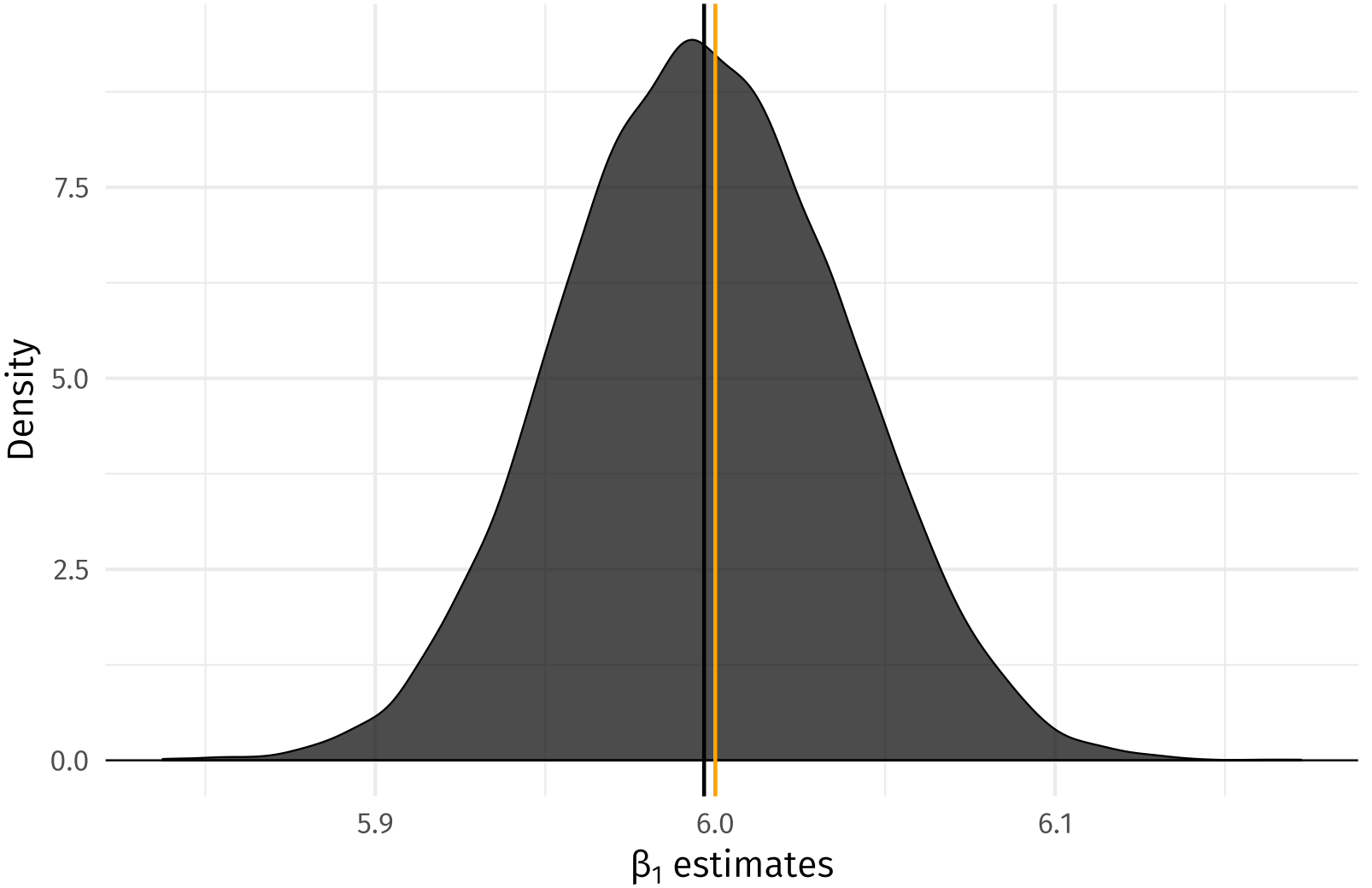
Thus, plotting the **distributions of estimates** for β_0 and β_1 will be of interest—**especially the means** of these distributions.

Recall: If an estimator is unbiased, then the mean of its distribution should like up with the parameter it is attempting to estimate.

Considering OLS's unbiasedness: The distribution of $\hat{\beta}_0$



Considering OLS's unbiasedness: The distribution of $\hat{\beta}_1$



Introductory example: Conclusion

When our disturbances are distributed as Uniform(0,1)

1. OLS is biased for β_0 (the intercept)
2. OLS is still unbiased for β_1 (the slope)

... and we were able to learn all of this information by simulation.

Now back to our question on k-fold cross validation and interdependence.

Simulation: k-fold CV and dependence

Our question

Let's write our previous question in a way we can try to answer it.

Question: How does correlation between observations affect the performance of k-fold cross validation?

Simulation: k-fold CV and dependence

Our question

Let's write our previous question in a way we can try to answer it.

Question: How does **correlation between observations** affect the **performance** of **k-fold cross validation**?

We need to define some terms.

Simulation: k-fold CV and dependence

Our question

Let's write our previous question in a way we can try to answer it.

Question: How does **correlation between observations** affect the **performance** of **k-fold cross validation**?

correlation between observations

- Observations can correlate in many ways. Let's keep it simple: we will simulate repeated observations (through time, t) on individuals, i.
- **DGP:** Continuous $y_{i,t}$ has a predictor $x_{i,t}$ and two random disturbances

$$
\begin{aligned}
y_{i,t} &= 3x_{i,t} - 2x_{i,t}^2 + 0.1x_{i,t}^4 + \gamma_i + \varepsilon_{i,t} \\
x_{i,t} &= 0.9x_{i,t-1} + \eta_{i,t} \\
\varepsilon_{i,t} &= 0.9\varepsilon_{i,t-1} + \nu_{i,t}
\end{aligned}
$$

Simulation: k-fold CV and dependence

Our question

Let's write our previous question in a way we can try to answer it.

Question: How does **correlation between observations** affect the **performance** of **k-fold cross validation**?

performance

- We will focus on **MSE** for observations the model never saw.
- *Note: En route*, we will meet *root* mean squared error (RMSE).

$$\text{RMSE} = \sqrt{\text{MSE}}$$

Simulation: k-fold CV and dependence

Our question

Let's write our previous question in a way we can try to answer it.

Question: How does **correlation between observations** affect the **performance** of **k-fold cross validation**?

k-fold cross validation

- We'll stick with 5-fold cross validation.
- Our answer shouldn't change too much based upon k.
- Feel free to experiment later...

Set up population

- Set seed
- Define number of individuals `N` and time periods `T`

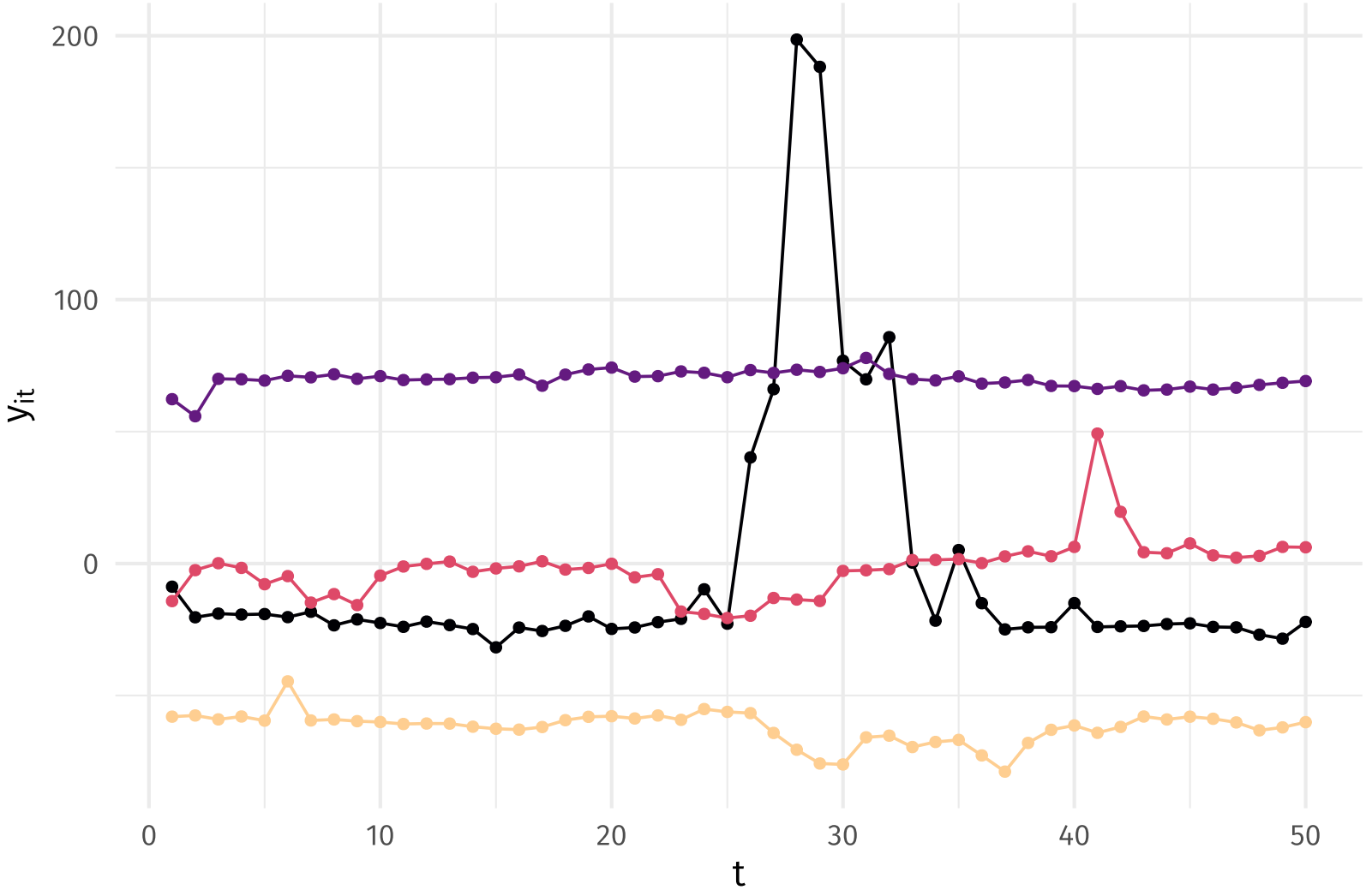
```
# Set seed
set.seed(12345)
# Size of the population
N = 1000
# Number of time periods per individual
T = 50
# Create the indices of our population
pop_df = expand_grid(
  i = 1:N,
  t = 1:T
)
```

Build population

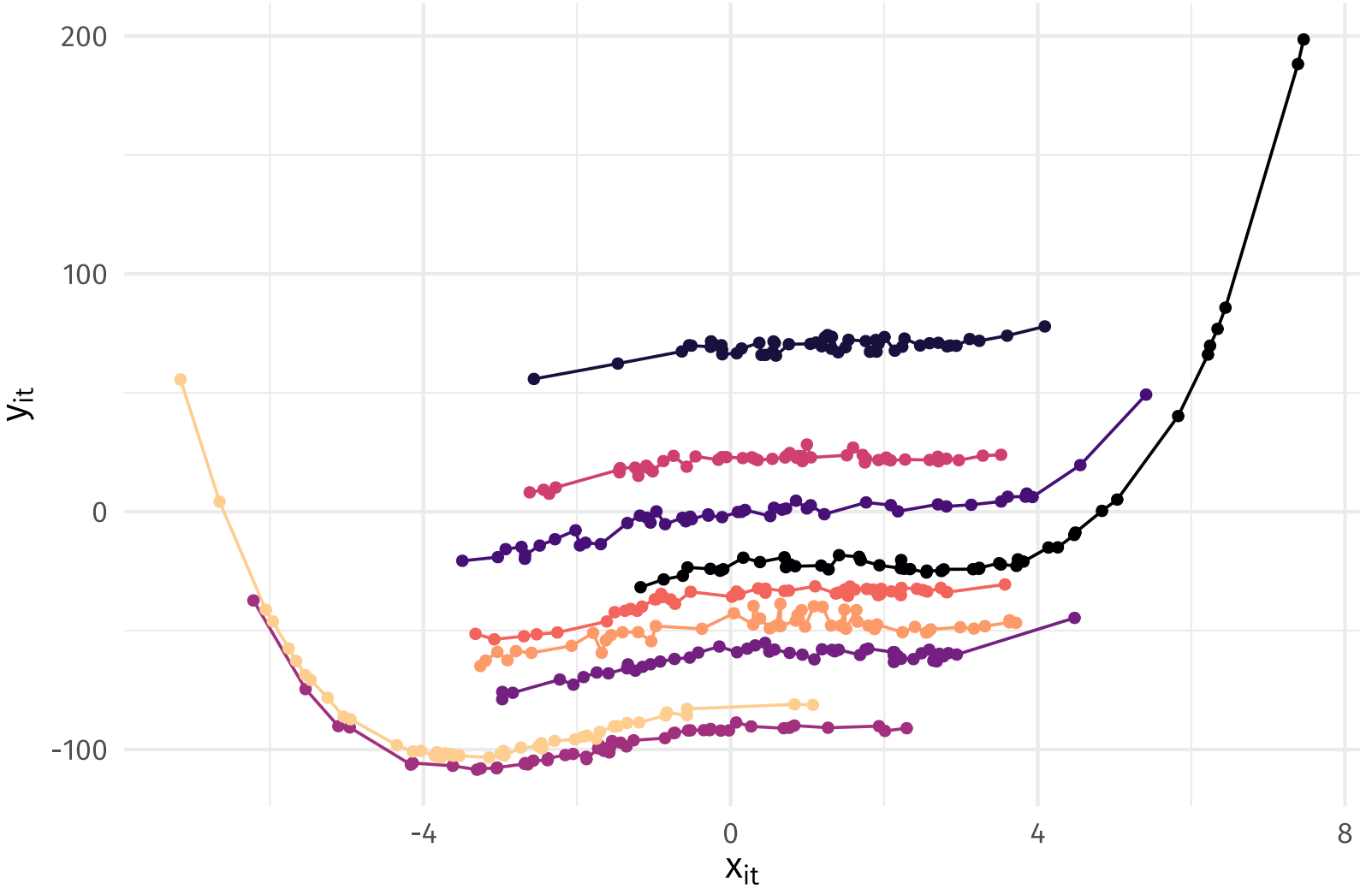
- Generate temporally correlated disturbances and predictor ($x_{i,t}$)
- Calculate outcome variable $y_{i,t}$

```
# Disturbance for (i,t): Correlated within time (not across individuals)
pop_df %<>%
  group_by(i) %>%
  mutate(
    x = arima.sim(model = list(ar = c(0.9)), n = T) %>% as.numeric(),
    e_it = arima.sim(model = list(ar = c(0.9)), n = T) %>% as.numeric()
  ) %>% ungroup()
# Disturbance for (i): Constant within individual
pop_df %<>%
  group_by(i) %>%
  mutate(e_i = runif(n = 1, min = -100, max = 100)) %>%
  ungroup()
# Calculate 'y'; drop disturbances
pop_df %<>% mutate(
  y = e_i + 3 * x - 2 * x^2 + 0.1 * x^4 + e_it
) %>% select(i, t, y, x)
```

Notice the **correlation within observation** across time.



Viewing the correlation in $x_{i,t}$ and $y_{i,t}$.



Next steps: Write out a single iteration (to become a function).

1. Draw a sample.
2. Estimate a model: We're going to use KNN regression.
3. Tune our model: We will use 5-fold CV to choose 'K'.
4. Assess the model's performance (CV and in the population).

Draw a sample

```
# Define sample size (will be in input of our function)
n_i = 50
# Draw sample
i_sampled = sample.int(N, size = n_i)
# Draw a sample
sample_df = pop_df %>% filter(i %in% i_sampled)
```

`sample.int(n, size)` draws `size` integers between 1 and `n`.

Note that we are **sampling individuals** (`i`).

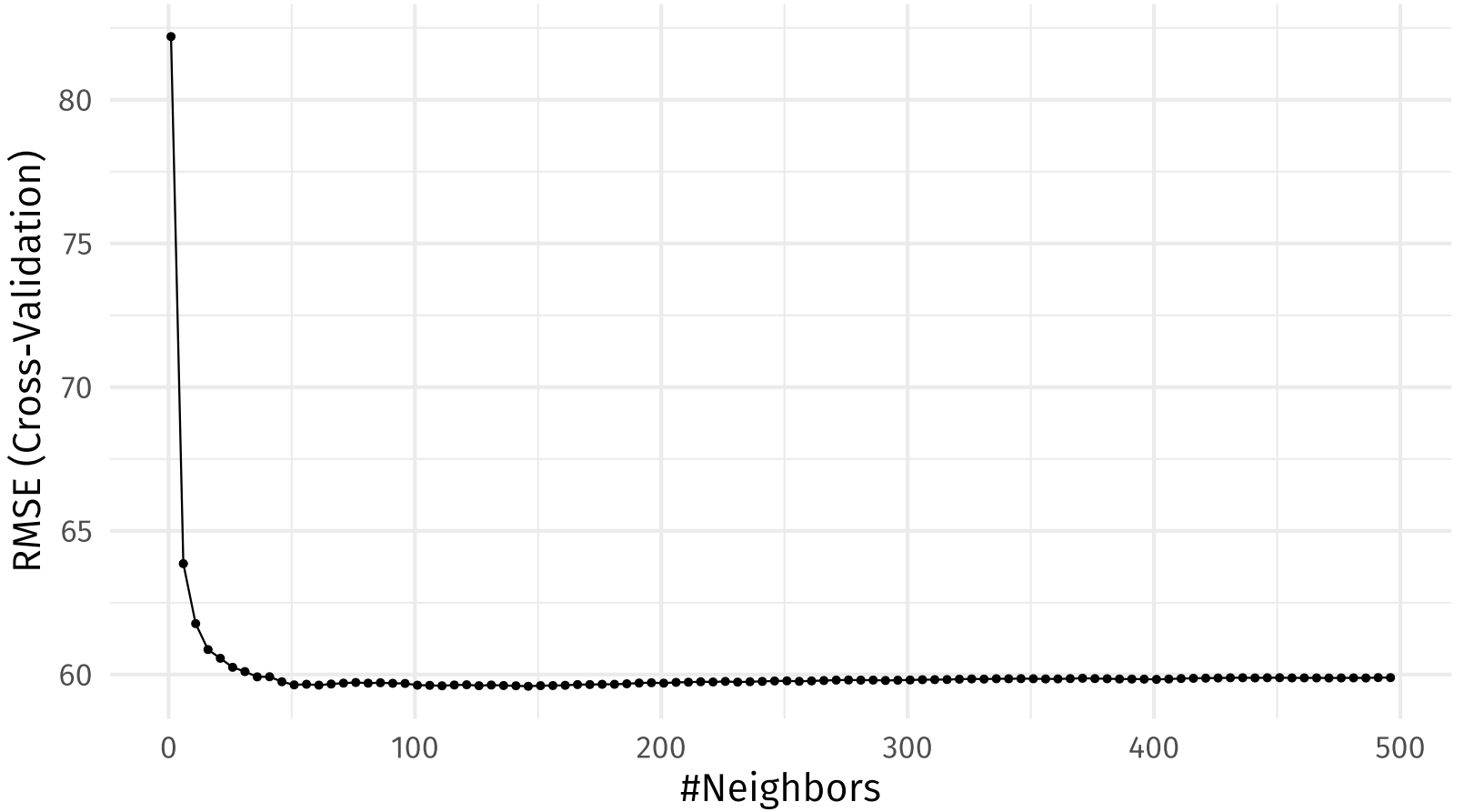
Tune and estimate a model

```
# Define number of folds
k_folds = 5
# k-fold CV
cv_output = train(
  # The relationship: y as a function of w and x
  y ~ .,
  # The method of estimating the model (linear regression)
  method = "knn",
  # The training data (which will be used for cross validation)
  data = sample_df %>% select(y, x),
  # Controls for the model training: k-fold cross validation
  trControl = trainControl(method = "cv", number = k_folds),
  # Allow cross validation to choose best value of K (# nearest neighbors)
  tuneGrid = expand.grid(k = seq(1, 500, by = 5))
)
```

`train()` (from the `caret` package) assists in many training/tuning tasks (note `tuneGrid` argument)—including pre-processing data.

You can actually plot the results from `train()`, *i.e.*,

```
ggplot(cv_output) + theme_minimal()
```



The output from `train()` also contains a lot of additional information, *e.g.*,

```
cv_output$results
```

k	RMSE	Rsquared	MAE	RMSESD	RsquaredSD	MAESD
1	82.199	0.006	66.382	1.751	0.006	1.904
6	63.859	0.012	54.800	2.303	0.016	2.043
11	61.771	0.015	53.856	1.616	0.017	1.206
16	60.869	0.018	53.216	1.512	0.018	1.321
21	60.565	0.017	53.064	1.299	0.016	1.191
26	60.251	0.018	52.847	1.338	0.020	1.134
31	60.103	0.019	52.737	1.207	0.018	1.050
[illegible]	[illegible]	[illegible]	[illegible]	1.158	0.019	0.994
[illegible]	[illegible]	[illegible]	[illegible]	0.921	0.013	0.792
[illegible]	[illegible]	[illegible]	[illegible]	0.905	0.013	0.775

RMSE	Rsquared	MAE	Resample
59.478	0.008	52.421	Fold4
60.316	0.015	52.723	Fold2
59.260	0.025	52.506	Fold3
58.917	0.032	51.603	Fold1
59.985	0.023	52.626	Fold5

Now we **assess the performance** of our chosen/tuned model.

1. Record the **CV-based MSE** (already calculated by `train()`).
2. Since we know the population, we can calculate the **true test MSE**.

(2) is usually not available to use—you can see how simulation helps.

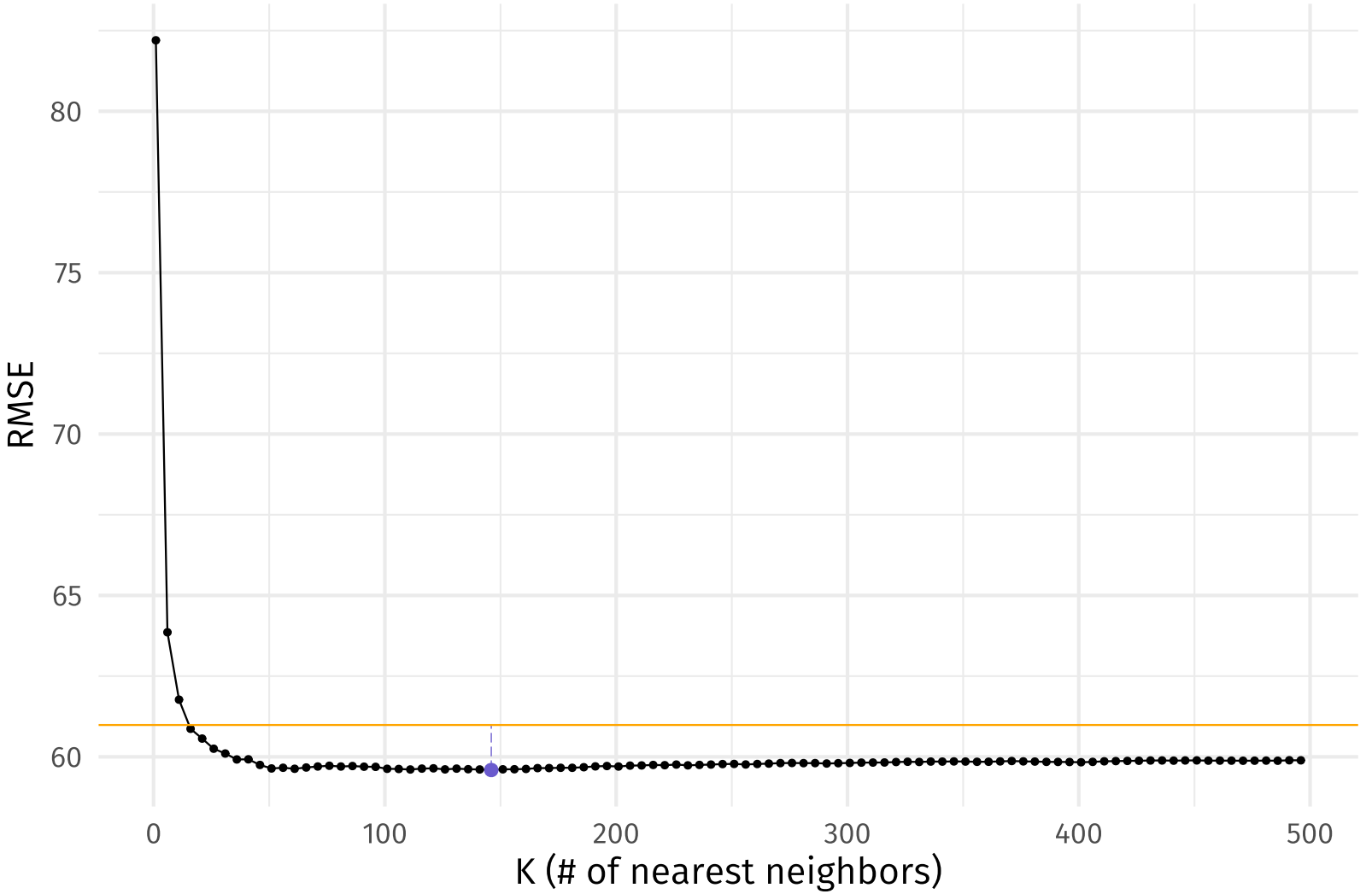
Assess performance

- Subset to unseen data (individuals not sampled)
- Evaluate prediction performance on new individuals

```
# Subset to unseen data
test_df = pop_df %>% filter(i %>% is_in(i_sampled) %>% not())
# Make predictions on
predictions = predict(
  cv_output,
  newdata = test_df
)
# Calculate RMSE in full population
rmse_true = mean((test_df$y - predictions)^2) %>% sqrt()
rmse_true
```

```
#> [1] 60.98547
```

Comparing **CV RMSE** to **true test RMSE**



Now we basically wrap the last 7 slides into a function and we're set!

Function: One iteration of the simulation

```
# Our function for a single iteration of the simulation
sim_fun = function(iter, n_i = 50, k_folds = 5) {
  # Draw sample
  i_sampled = sample.int(N, size = n_i)
  # Draw a sample
  sample_df = pop_df %>% filter(i %in% i_sampled)
  # k-fold CV
  cv_output = cv_function(k_folds = k_folds)
  # Find the estimated MSE
  mse_est = mean(cv_output$resample$RMSE^2)
  # Subset to unseen data
  test_df = pop_df %>% filter(i %>% is_in(i_sampled) %>% not())
  # Make predictions on true test data
  predictions = predict(cv_output, newdata = test_df)
  # Calculate RMSE in full population
  mse_true = mean((test_df$y - predictions)^2)
  # Output results
  data.frame(iter, k = cv_output$bestTune$k, mse_est, mse_true)
}
```

Notice that we can define default values for our function's arguments.

Function: k-fold CV of KNN model for a sample

```
# Wrapper function for caret::train()
cv_function = function(k_folds) {
  # The relationship: y as a function of w and x
  y ~ .,
  # The method of estimating the model (linear regression)
  method = "knn",
  # The training data (which will be used for cross validation)
  data = sample_df %>% select(y, x),
  # Controls for the model training: k-fold cross validation
  trControl = trainControl(method = "cv", number = k_folds),
  # Allow cross validation to choose best value of K (# nearest neighbors)
  tuneGrid = expand.grid(k = seq(1, 200, by = 10))
}
```

Note This function would need to be defined first in a script.

Now run our one-iteration function `sim_fun()` for many iterations!

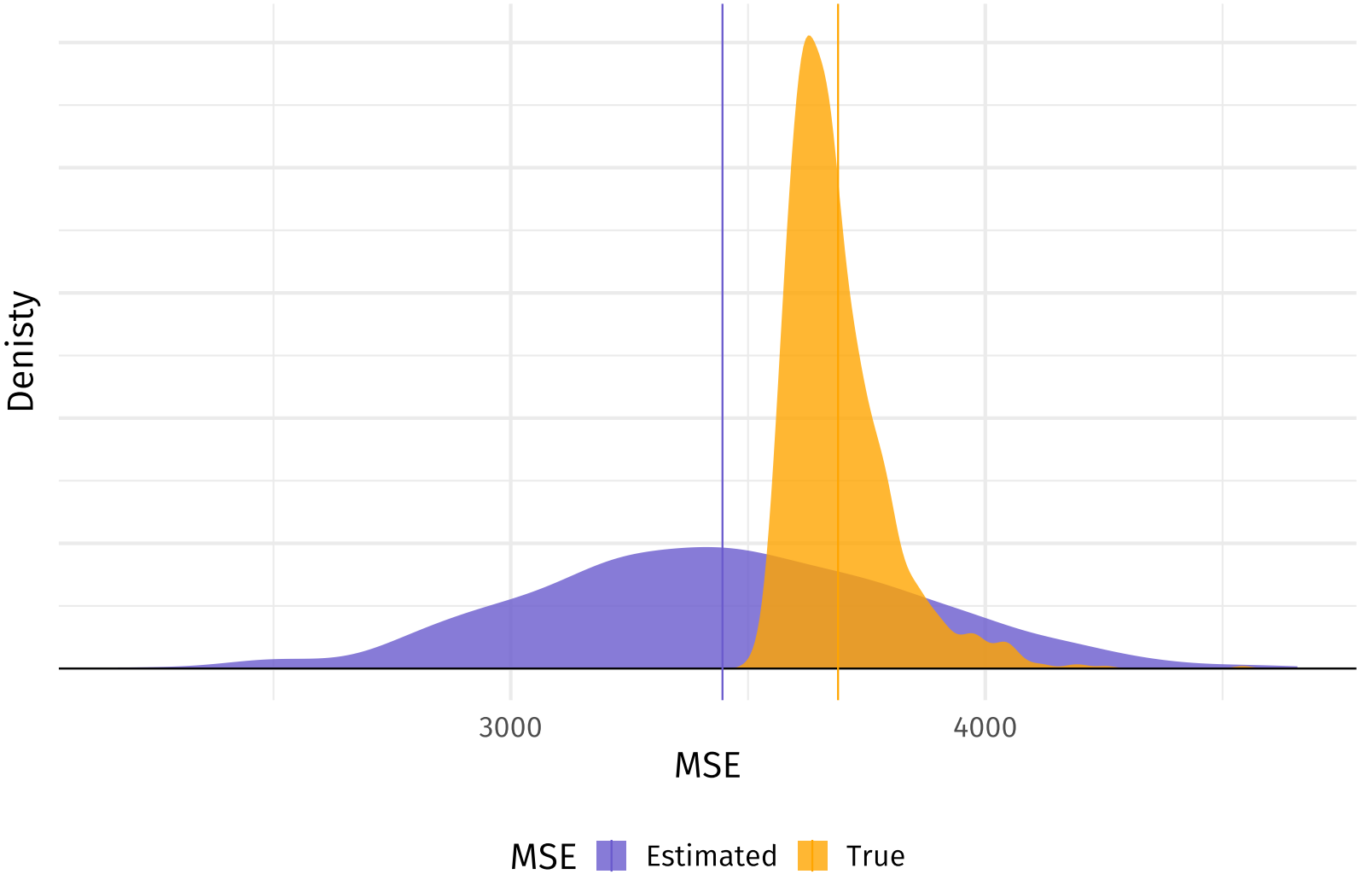
Run the simulation!

```
# Set seed
set.seed(123)
# Run simulation 1,000 times in parallel (and time)
tic()
sim_df = mclapply(
  X = 1:1e3,
  FUN = sim_fun,
  mc.cores = 11
) %>% bind_rows()
toc()
# Save dataset
write_csv(
  x = sim_df,
  path = here("cv-sim-knn.csv")
)
```

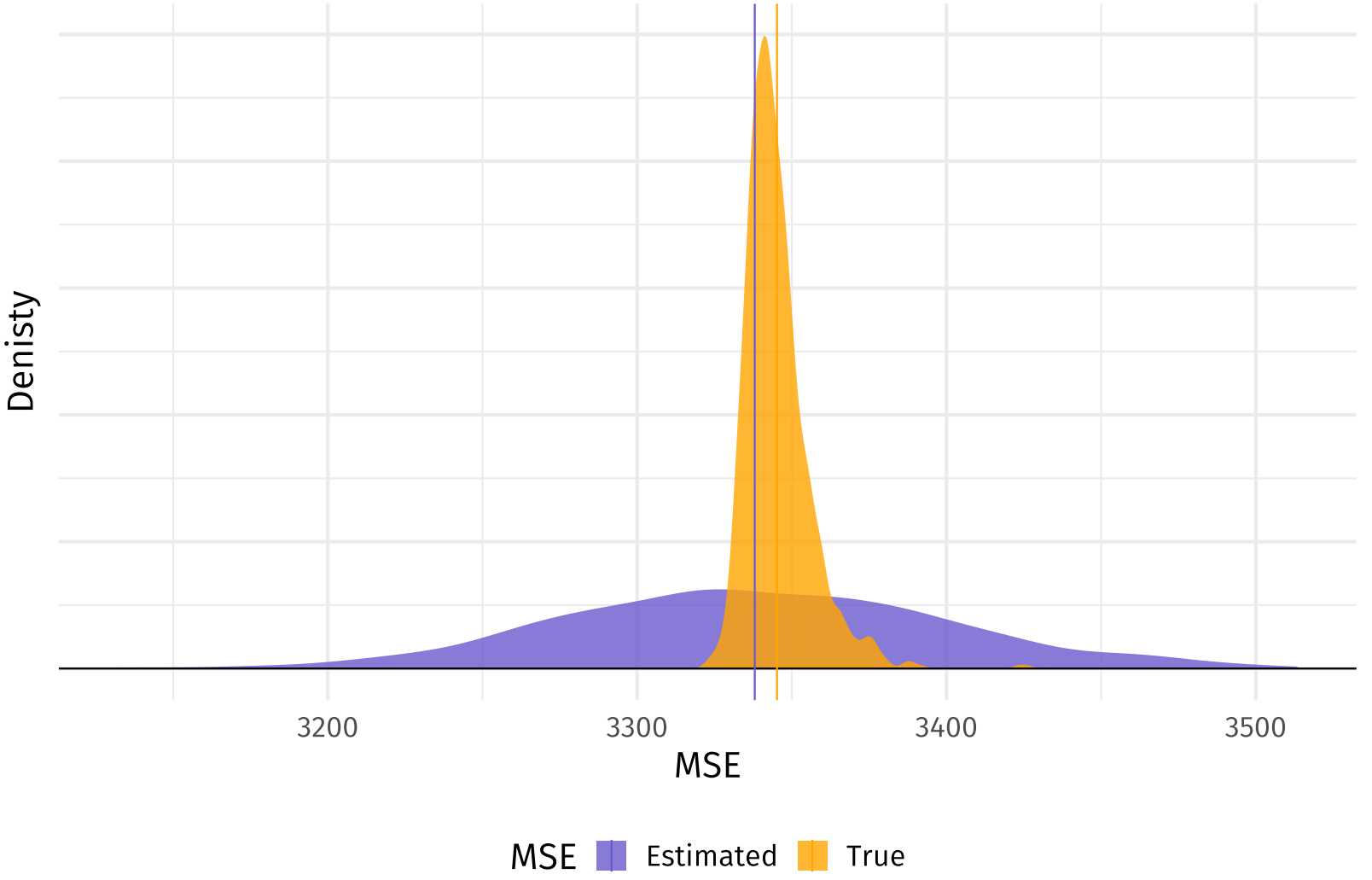
`tic()` and `toc()` come from `tictoc()` and help with timing tasks.
`mclapply()` is a parallelized `lapply()` from `parallel` (sorry, no Windows).

How about some results?

With dependence: distributions of the **true MSE** and the **CV-based MSE**.

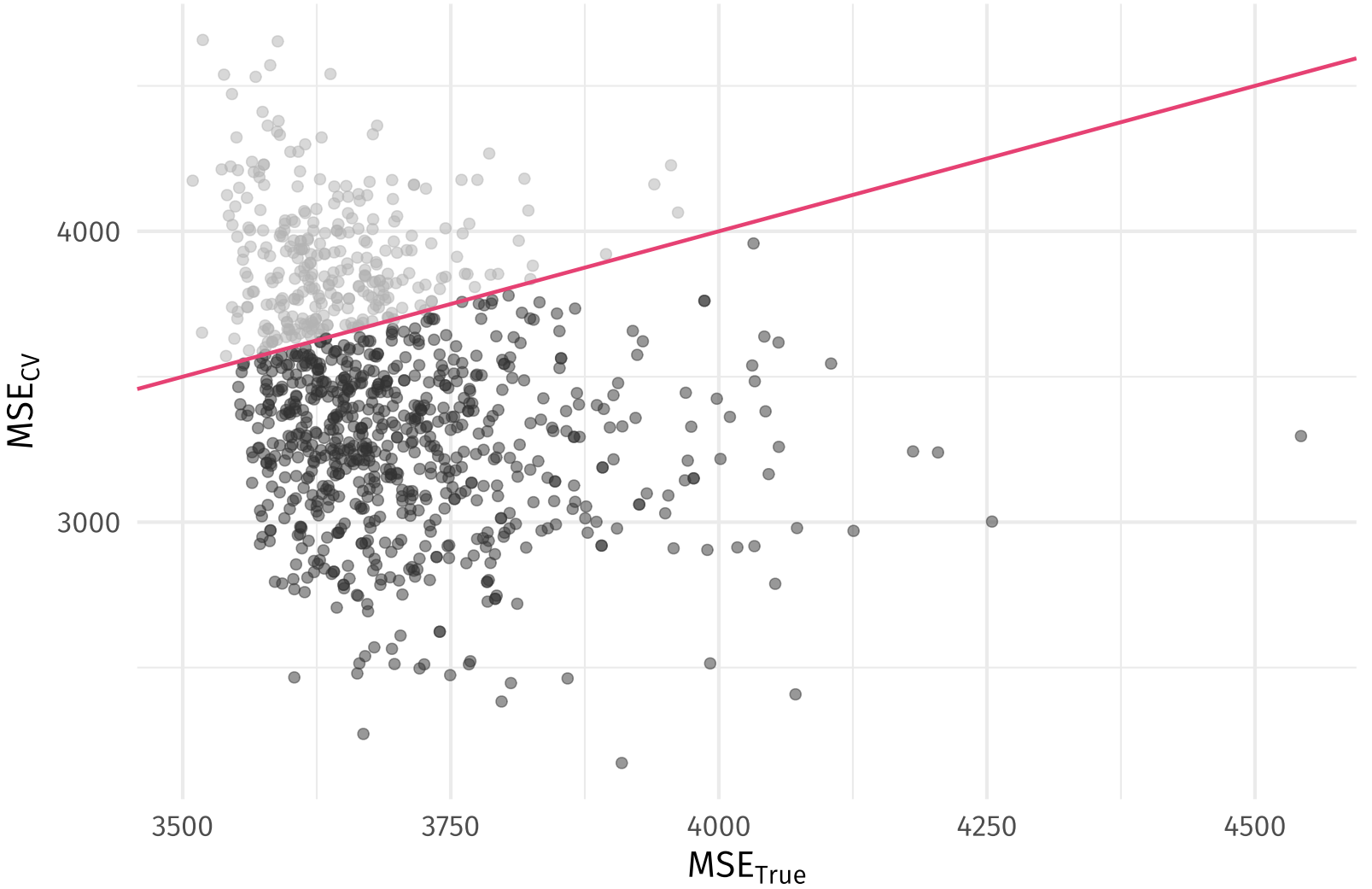


With **independence across observations: true MSE** and the **CV-based MSE**.



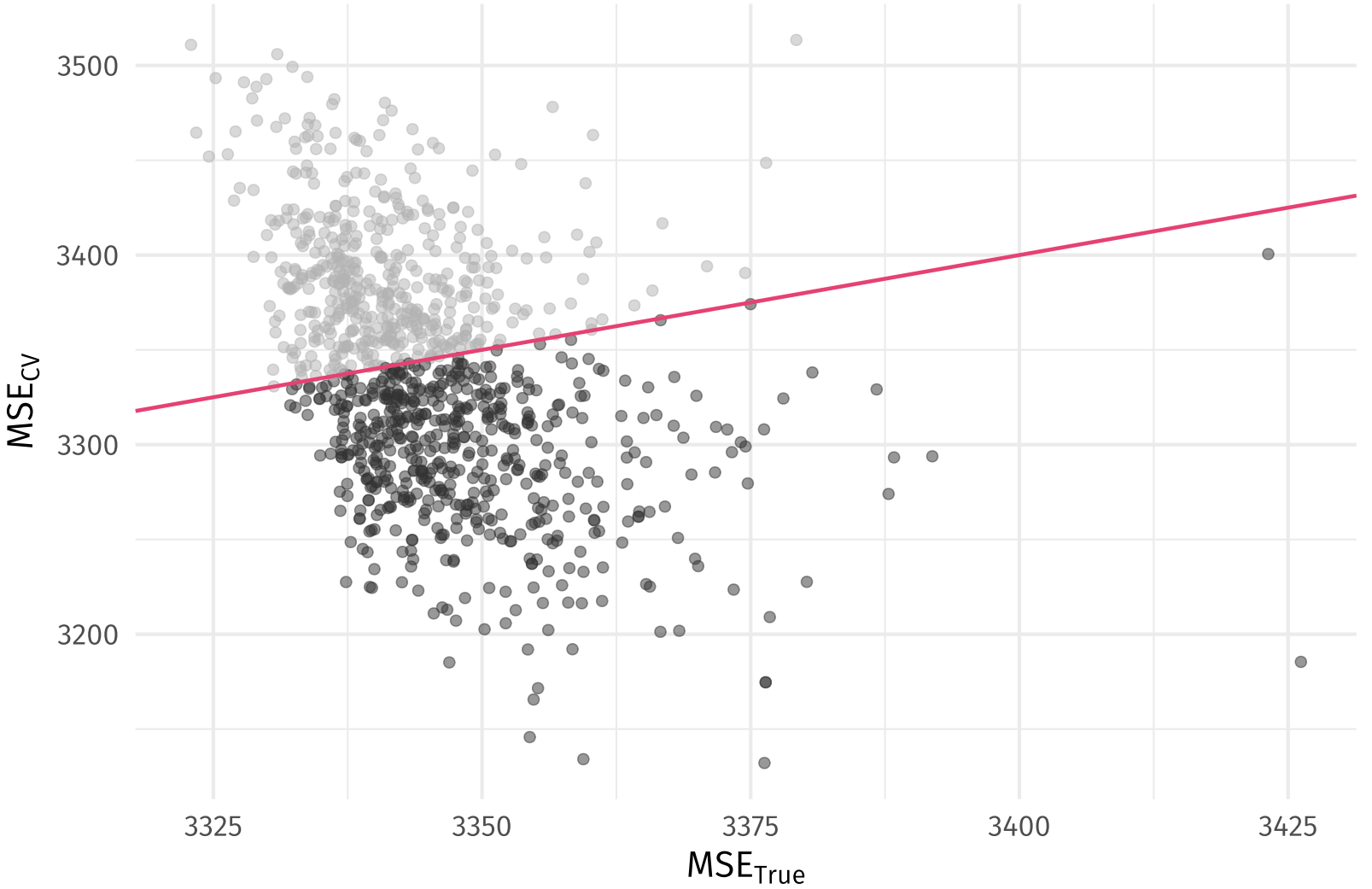
Dependence: Comparing **true MSE** and **CV-based MSE** (45° line)

Tendency to **underestimate test MSE** (rather than overestimate): 70.2%

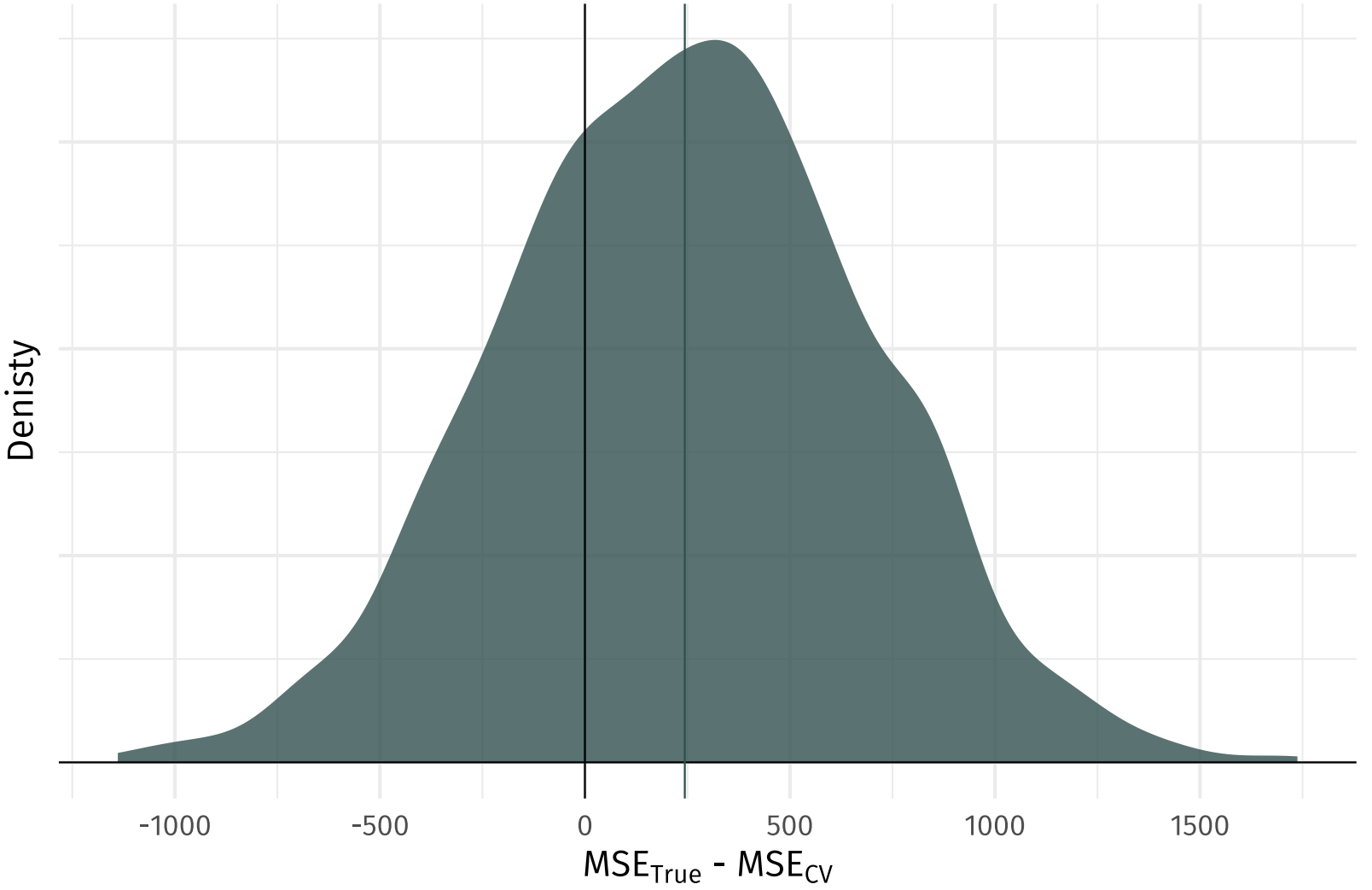


Independence: Comparing **true MSE** and **CV-based MSE** (45° line)

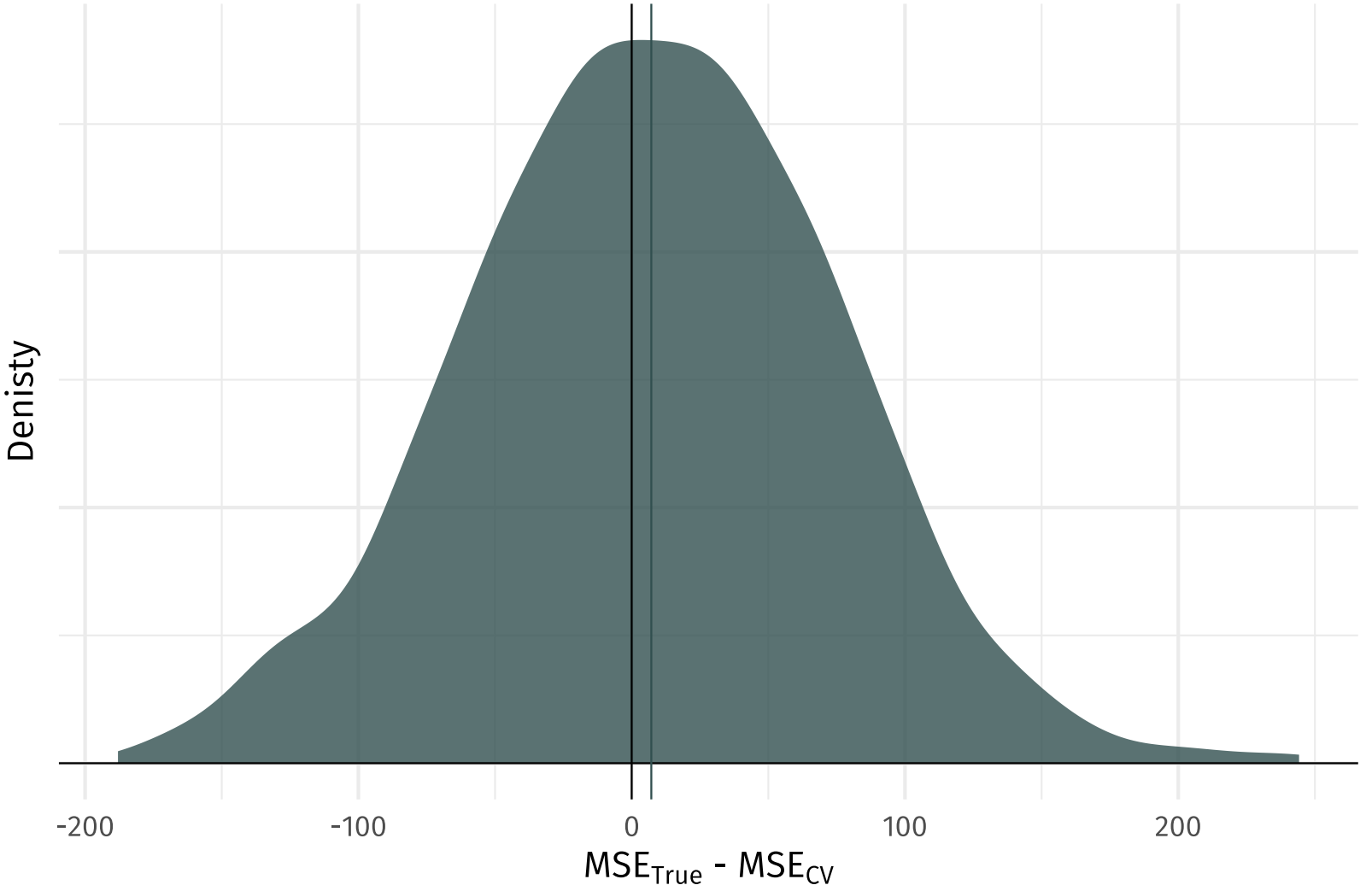
Less likely (53.9%) to underestimate true, test MSE



Dependence: The **difference** between the true and CV-estimated MSE.

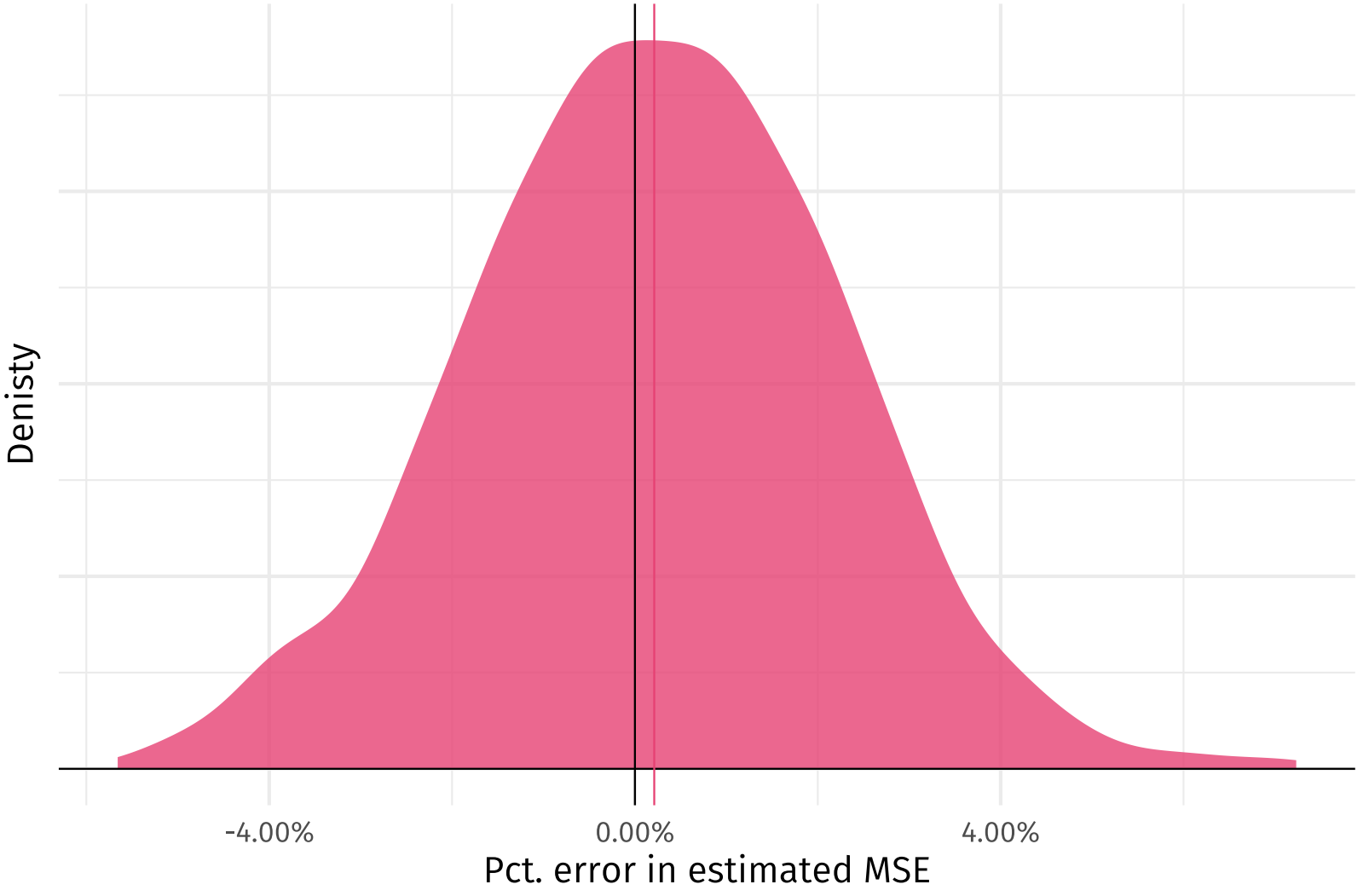


Independence: The **difference** between the true and CV-estimated MSE.

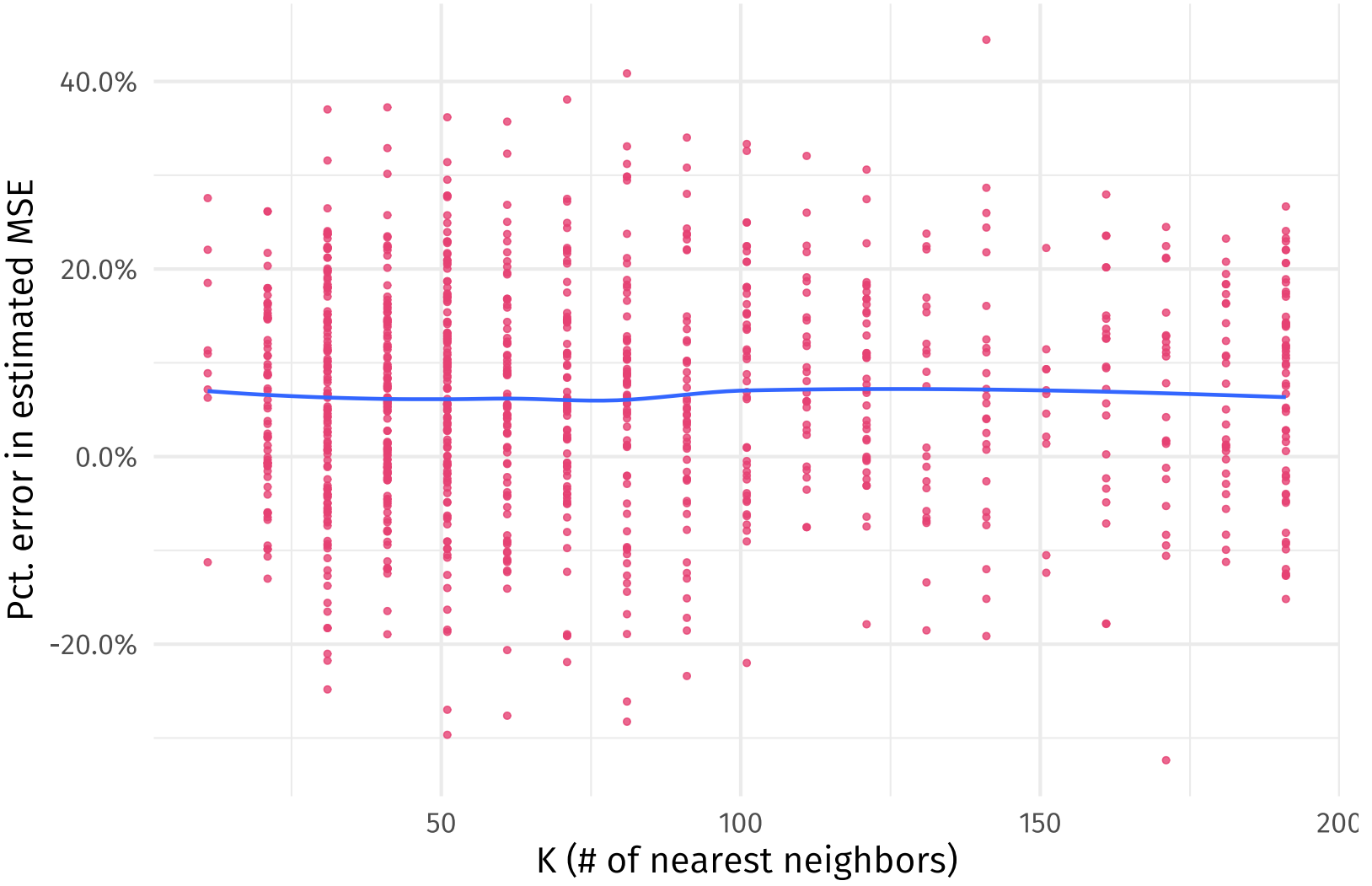


Dependence: **Percent difference** between the true and CV-estimated MSE.

Independence: **Percent difference** between the true and CV-estimated MSE.



Dependence: KNN flexibility (K) and percentage error



Independence: KNN flexibility (K) and percentage error

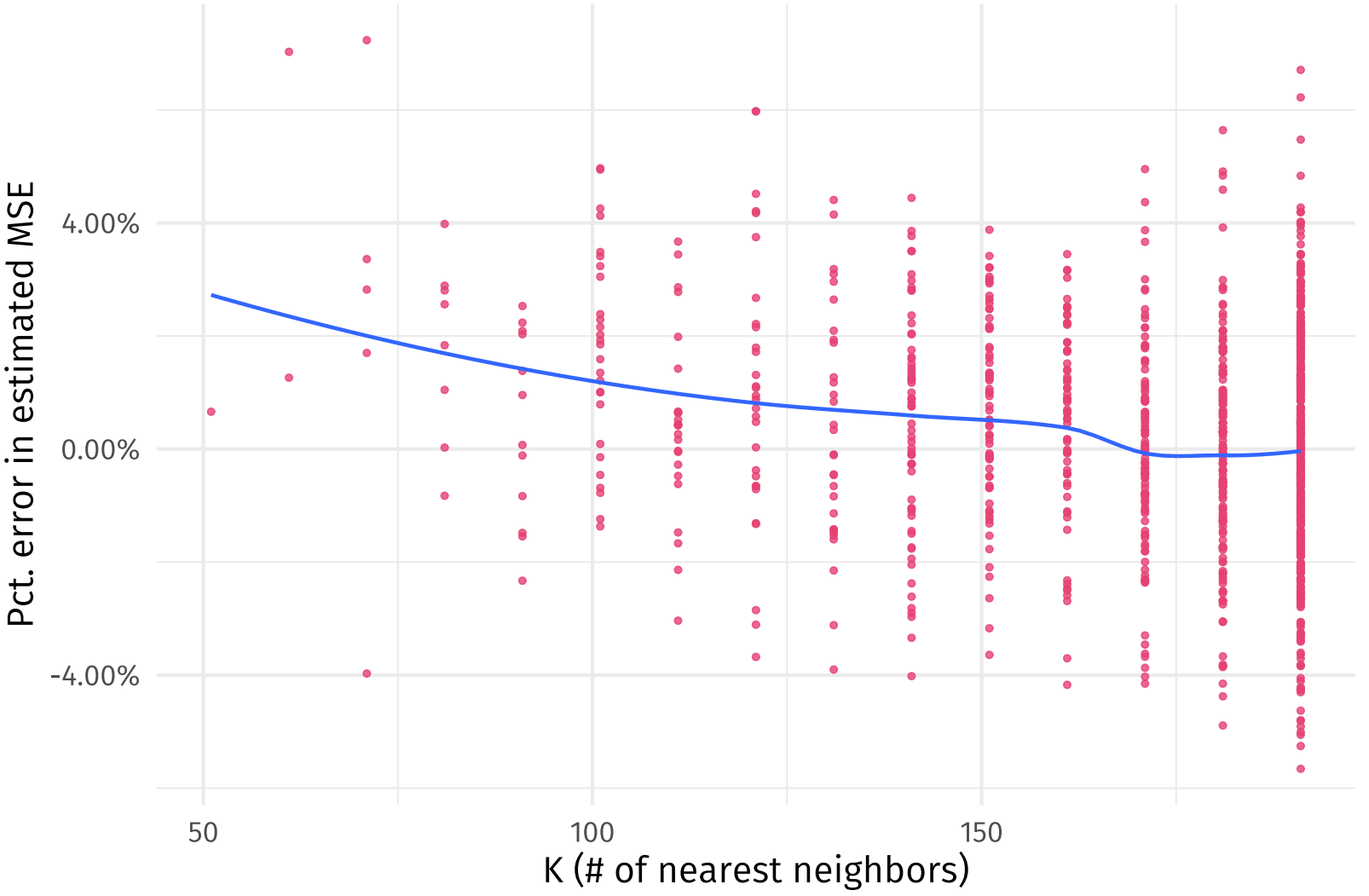


Table of contents