

Lecture 12

Evaluating Regression Models

Dennis Sun
Stanford University

DATASCI 112

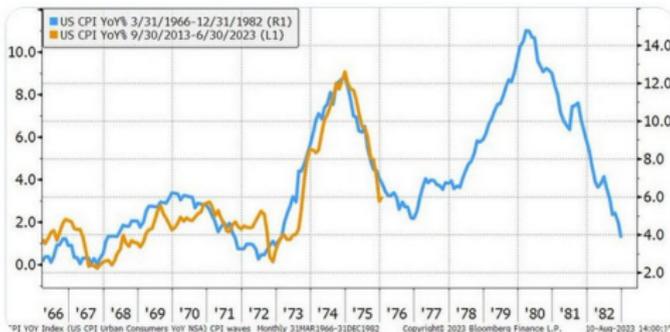
February 4, 2026



Lawrence H. Summers ✓
@LHSummers

...

This picture should be sobering to anyone convinced that we have
reattained price stability.



- 1 Recap
- 2 Measuring Error
- 3 Estimating Test Error
- 4 Cross-Validation



1 Recap

2 Measuring Error

3 Estimating Test Error

4 Cross-Validation



Training and Test Data

The data for which we know the label y is called the **training data**.

The data for which we don't know y (and want to predict it) is called the **test data**.

```
import pandas as pd

df = pd.read_csv("https://dlsun.github.io/pods/data/bordeaux.csv",
                 index_col="year")
df_train = df.loc[:1980].copy()
df_test = df.loc[1981:].copy()
```

Let's separate the inputs X from the labels y .

```
X_train = df_train[["win", "summer"]]
y_train = df_train["price"]

X_test = df_test[["win", "summer"]]
```



K-Nearest Neighbors

We've seen one machine learning model: *k*-nearest neighbors.

```
from sklearn.pipeline import make_pipeline
from sklearn.preprocessing import StandardScaler
from sklearn.neighbors import KNeighborsRegressor
```

```
pipeline = make_pipeline(
    StandardScaler(),
    KNeighborsRegressor(n_neighbors=5))
pipeline.fit(X=X_train, y=y_train)
pipeline.predict(X=X_test)
```

```
array([35.8, 54. , 52.2, 18.4, 35.6, 13.2, 37. , 51.4, 36.6, 36.6, 40.6])
```

Today: How do we know if this model is any good?



1 Recap

2 Measuring Error

3 Estimating Test Error

4 Cross-Validation



Prediction Error

If the true labels are y_1, \dots, y_n and our model predicts $\hat{y}_1, \dots, \hat{y}_n$, how do we measure how well our model did?

- **mean squared error (MSE)**

$$\text{MSE} = \frac{1}{n} \sum_{i=1}^n (y_i - \hat{y}_i)^2.$$

- **mean absolute error (MAE)**

$$\text{MAE} = \frac{1}{n} \sum_{i=1}^n |y_i - \hat{y}_i|.$$

Calculating MSE or MAE requires data where true labels are known. Where can we find such data?



Training Error

On the training data, the true labels y_1, \dots, y_n are known.

Let's calculate the **training error** of our model.

```
pipeline.fit(X_train, y_train)
y_train_ = pipeline.predict(X_train)
((y_train - y_train_) ** 2).mean()
```

207.24148148148146

There's also a Scikit-Learn function for that!

```
from sklearn.metrics import mean_squared_error
mean_squared_error(y_train, y_train_)
```

207.24148148148146

How do we interpret this MSE of 207.24?

Remember, we are predicting the price of wine. So the model is off by 207.24 square dollars on average.

The square root is easier to interpret. The model is off by $\sqrt{207.24} \approx \$14.40$ on average. This is called the **RMSE**.



The Problem with Training Error

What's the training error of a 1-nearest neighbor model?

```
pipeline = make_pipeline(  
    StandardScaler(),  
    KNeighborsRegressor(n_neighbors=1))  
pipeline.fit(X=X_train, y=y_train)  
y_train_ = pipeline.predict(X=X_train)  
mean_squared_error(y_train, y_train_)
```

0.0

Why did this happen?

The 1-nearest neighbor to any observation in the training data is the observation itself!

A 1-nearest neighbor model will always be perfect on the training data. But is it necessarily the best model?



Test Error

We don't need to know how well our model does on *training data*.

We want to know how well it will do on *test data*.

In general, test error $>$ training error.

Analogy: A professor posts a practice exam before an exam.

- If the actual exam is the same as the practice exam, how many points will students miss? That's training error.
- If the actual exam is different from the practice exam, how many points will students miss? That's test error.

It's always easier to answer questions that you've seen before than questions you haven't seen.

Now: How do we estimate the test error?



- 1 Recap
- 2 Measuring Error
- 3 Estimating Test Error**
- 4 Cross-Validation



Validation Set

The training data is the only data we have, where the true labels y are known.

So one way to estimate the test error is to not use all of the training data to fit the model, leaving the remaining data for estimating the test error.



Implementing the Validation Set

Let's implement this idea in a Colab!



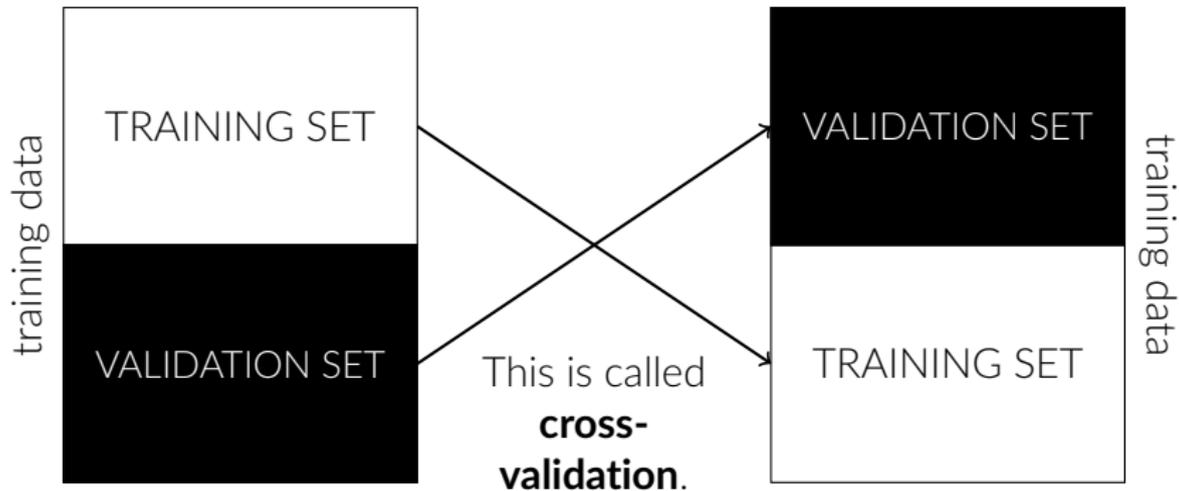
- 1 Recap
- 2 Measuring Error
- 3 Estimating Test Error
- 4 Cross-Validation**



Cross-Validation

The way we split the data into two halves was arbitrary.

Why not use the 2nd half for training and the 1st half for validation?



Implementing Cross-Validation from Scratch

Previously, we fit the model to the training set and evaluated the predictions on the validation set.

```
pipeline.fit(X_train_set, y_train_set)
y_val_set_ = pipeline.predict(X_val_set)
mean_squared_error(y_val_set, y_val_set_)
```

195.71428571428572

Now let's do the same thing, but with the roles of the training and validation sets reversed.

```
pipeline.fit(X_val_set, y_val_set)
y_train_set_ = pipeline.predict(X_train_set)
mean_squared_error(y_train_set, y_train_set_)
```

306.9230769230769

Wow, the estimates can be quite different!

To come up with one overall estimate of the test error, we can average them.

```
(195.71 + 306.92) / 2
```

251.315

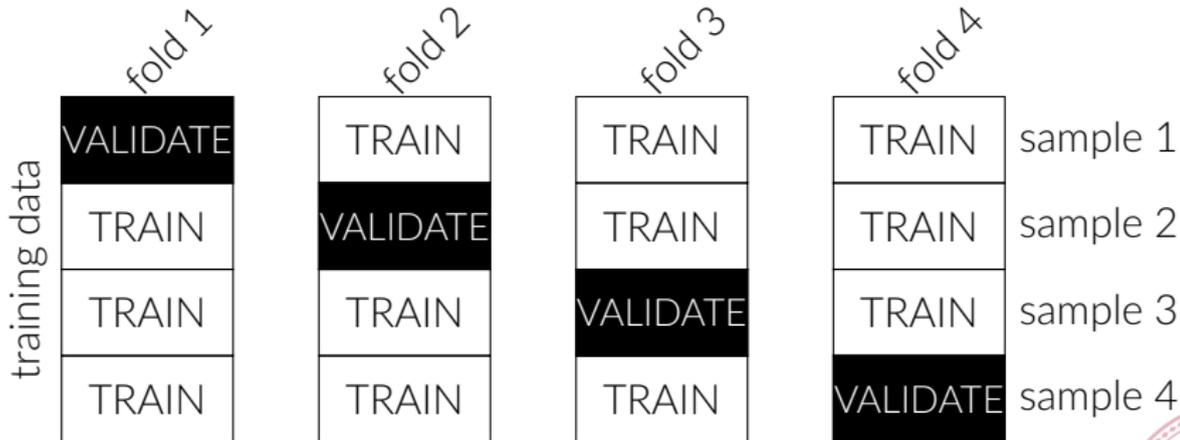


K-Fold Cross Validation

One problem with splitting the data into two is that we only fit the model on half of the data.

A model trained on half of the data may be very different from a model trained on all of the data.

It may be better to split the data into K samples and come up with K validation errors.



This way, we use $1 - 1/K$ of the data for training.



Implementing Cross-Validation in Scikit-Learn

You specify the model, data, and K . Scikit-Learn will:

- split the training data into K samples
- hold out one sample at a time as a validation set
 - fit the model to remaining $1 - 1/K$ of the data
 - predict the labels on the validation set
 - calculate the prediction error

```
from sklearn.model_selection import cross_val_score
scores = cross_val_score(
    pipeline,
    X=df_train[["win", "summer"]],
    y=df_train["price"],          # this is all of the training data!
    scoring="neg_mean_squared_error", # higher is better for a score
    cv=4)
scores
array([-547.          , -405.85714286, -67.          , -31.          ])
```

So an overall estimate of test MSE is:

```
-scores.mean()
```

262.7142857142857

