Lecture 14
Evaluating Regression Models

Dennis Sun
Stanford University
DATASCI / STATS 112

February 13, 2023

1. Review
2. Measuring Error
3. Estimating Test Error
4. Conclusion

K-Nearest Neighbors

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**.

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

```
pipeline = make_pipeline(
           StandardScaler(),
           KNeighborsRegressor(n_neighbors=5, metric="euclidean"))

pipeline.fit(X=X_train, y=y_train)
pipeline.predict(X=pd.DataFrame([x_test]))
```

```
array([13.2])
```

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

1 Review

2 Measuring Error

3 Estimating Test Error

4 Conclusion

Prediction Error

Suppose the true label is y_i and our model predicts $\hat{y}_i$. How do we measure how well our model did?

- **mean squared error (MSE)**

$$\text{MSE} = \text{mean of } (y_i - \hat{y}_i)^2.$$

- **mean absolute error (MAE)**

$$\text{MAE} = \text{mean of } |y_i - \hat{y}_i|.$$

Calculating MSE or MAE requires data where the true labels are known. Where do we get such data?

Training Error

On the training data, we have the true labels y_i.

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
```

Note that you can use Scikit-Learn to calculate the MSE!

```
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 Data

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

```
pipeline = make_pipeline(
           StandardScaler(),
           KNeighborsRegressor(n_neighbors=1, metric="euclidean"))
pipeline.fit(X_train, y_train)
y_train_ = pipeline.predict(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 itself! So the predictions are perfect!

A 1-nearest neighbor model will always have 0 training error. 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 predicts 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.

Problem: we can't measure test error because we don't have the true labels on the test data.

Now: How do we estimate the test error?

1 Review

2 Measuring Error

3 Estimating Test Error

4 Conclusion

Validation Set

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

So one way to estimate the test error is to use only *some* of the training data to fit the model, leaving the rest to estimate the test error.

Implementing the Validation Set

We randomly sample 50% of the data to be in the training set, leaving the rest for the validation set.

```
train = bordeaux_train.sample(frac=.5)
val = bordeaux_train.drop(train.index)
```

train

year	price	summer	har	sep	win	age
1965	11.0	15.4	267	14.8	602	27
1966	47.0	16.5	86	18.4	819	26
1967	19.0	16.2	118	16.5	714	25
1959	66.0	17.5	187	18.7	485	33
1960	14.0	16.4	290	15.8	763	32
...	...	...	...	...	...	...
1977	11.0	15.6	87	16.8	821	15
1964	31.0	17.3	96	18.8	402	28
1968	11.0	16.2	292	16.4	610	24
1957	22.0	16.1	110	16.2	420	35
1973	16.0	17.1	123	17.9	376	19

14 rows × 6 columns

val

year	price	summer	har	sep	win	age
1952	37.0	17.1	160	14.3	600	40
1953	63.0	16.7	80	17.3	690	39
1955	45.0	17.1	130	16.8	502	37
1961	100.0	17.3	38	20.4	830	31
1962	33.0	16.3	52	17.2	697	30
...	...	...	...	...	...	...
1971	27.0	16.8	112	16.9	551	21
1974	11.0	16.3	184	16.2	574	18
1975	30.0	16.9	171	17.2	572	17
1979	21.0	16.2	122	17.3	717	13
1980	14.0	16.0	74	18.4	578	12

13 rows × 6 columns

```
X_train, y_train = train[["win", "summer"]], train["price"]
X_val, y_val = val[["win", "summer"]], val["price"]
```

Implementing Validation Error

Now we fit the model to the training set and predict on the validation set.

```
pipeline = make_pipeline(
           StandardScaler(),
           KNeighborsRegressor(n_neighbors=1, metric="euclidean"))
pipeline.fit(X_train, y_train)
y_val_ = pipeline.predict(X_val)
mean_squared_error(y_val, y_val_)
```

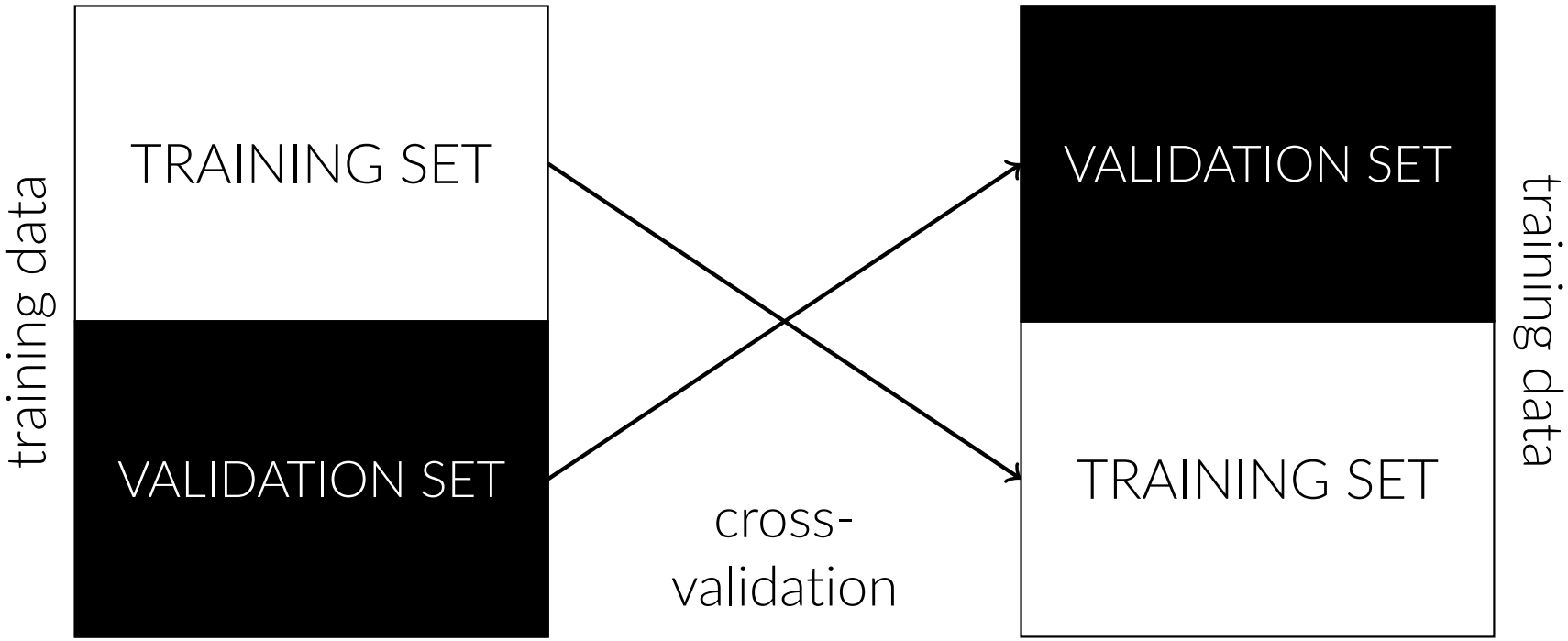
```
579.3076923076923
```

Note that, unlike the training error, the validation error is not 0. It is a better estimate of the test error.

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 predictions on the validation set.

```
pipeline.fit(X_train, y_train)
y_val_ = pipeline.predict(X_val)
mean_squared_error(y_val, y_val_)
```

```
579.3076923076923
```

Now let's do the same thing, with the roles reversed.

```
pipeline.fit(X_val, y_val)
y_train_ = pipeline.predict(X_train)
mean_squared_error(y_train, y_train_)
```

```
213.21428571428572
```

Notice that the estimates can be quite different!

To come up with one overall estimate, we can average the errors:

```
(579.3076923076923 + 213.21428571428572) / 2
```

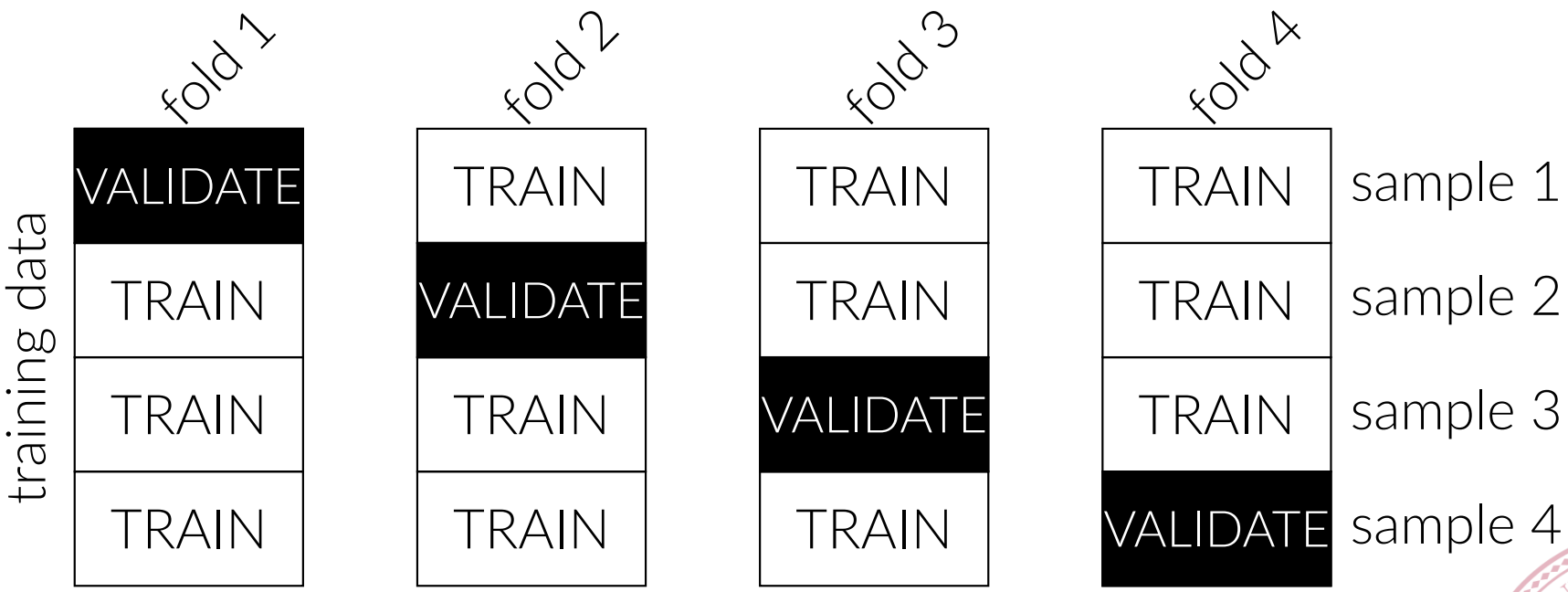
```
396.26098901098896
```

K-Fold Cross Validation

One problem with splitting the data into two is that we only use $1/2$ of the data for training.

A model trained on $1/2$ of the data may behave differently 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
- fit the model to each training set (K times)
- predict on each validation set (K times)
- calculate the prediction error (K times)

```
from sklearn.model_selection import cross_val_score
scores = cross_val_score(
    pipeline,
    X=bordeaux_train[["win", "summer"]],
    y=bordeaux_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
```

1 Review

2 Measuring Error

3 Estimating Test Error

4 Conclusion

Summary

How do we evaluate machine learning models?

- Prediction error (MSE / MAE), but it requires knowing the true label y.
- We could calculate training error…
- …but it is a bad estimate of test error.
- Cross-validation, which involves splitting up your training data, produces a better estimate of the test error.

Reminders

- Assignment 4 is due next Monday. (I gave you a few extra days for this one.)
- You should know most of what you need to finish Assignment 4 by tomorrow.
- You should be starting to gather data for your final project this week.