Model and hyperparameter selection

Magnus Nielsen, SODAS, UCPH

Agenda

- The holdout method
- Regularized regression
- Pipeline
- Gridsearch
- Curves

New field, new lingo

Parameters are the same as weights

Intercept is the same as bias

Covariates are the same as features

$$y = Xw + \epsilon$$

$$y = X\beta + \epsilon$$

Notation is a bit different, but it's the same

Holdout method

Minimizing loss functions

Interested in minizing the expected loss, $E_{(x,y)}[L(\hat{f}(x)), y]$, on *unseen data*

- $\hat{f}(\cdot)$ is a (possible non-linear) function mapping the input X to the output y, i.e. $\hat{y}$
- $L(\cdot)$ could be the mean squared error for regression problems or accuracy for binary classification

Sadly, we have no unseen data, so how do we find the best model?

Introducing the holdout method

We only train on some fraction of the data, leaving a part of the sample only for model evaluation!

We split the data into two parts:

- One for development/training and one for testing.

The testing data must *only* be used **ONCE** at the end to evaluate the models.

The development dataset can be used as one sees fit

We generally split at least once more

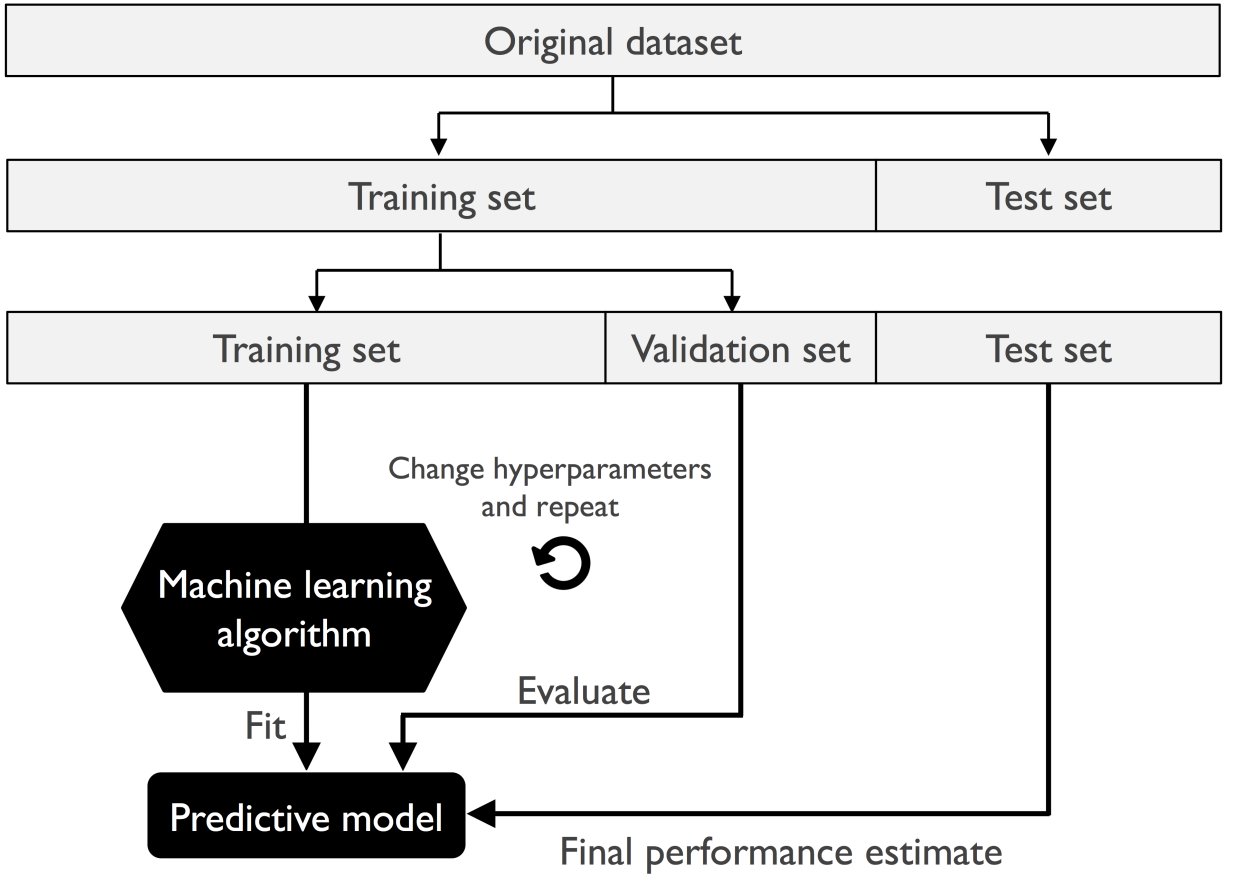


Figure 1: The holdout method

Source: Raschka & Mirjalili, 2019, ch. 6

At least once more!

Using K-fold cross-validation, we split the development data into *K* folds, train on $K-1$ and validate on one, repeating K times

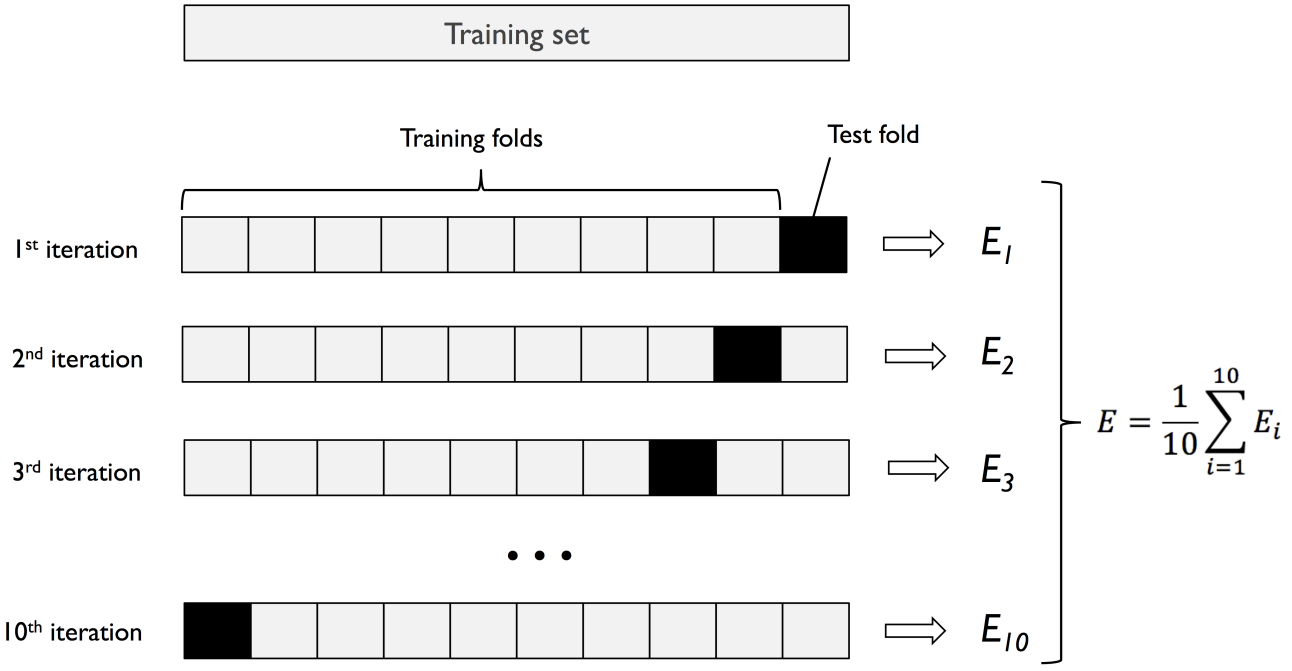


Figure 2: K-fold cross-validation

There are other methods

Repeated K-fold is also popular, but other methods exist, most commonly due to:

- stratification issues
- timeseries structure

More splits require more computation!

- Cost
- Energy
- Time

Variance estimates are not simple

There exists no unbiased estimator of the variance of K-fold (Bengio & Grandvalet, 2003), so you should:

- Refrain from making claims about significance
- Look into the literature, i.e. Nadeau & Bengio (1999), and find a conservative estimator which fits your scenario

We will not cover this and use point estimates only

Regression

Ordinary Least Squares

OLS is a minimization problem

$$\hat{w} = \frac{1}{N} \underset{w}{\operatorname{argmin}} \left\{ ||y - Xw||_2^2 \right\}$$

Where $|| \cdot ||_2$ is the Euclidean norm

It also has a very familiar closed form solution

$$\hat{w} = (X'X)^{-1}(X'y)$$

In general, we let the people who implement the models care about solving the problems, and I won't spend much time on it

Overfitting is an issue

We don’t control the bias-variance (over-underfitting) trade-off, as OLS is unbiased

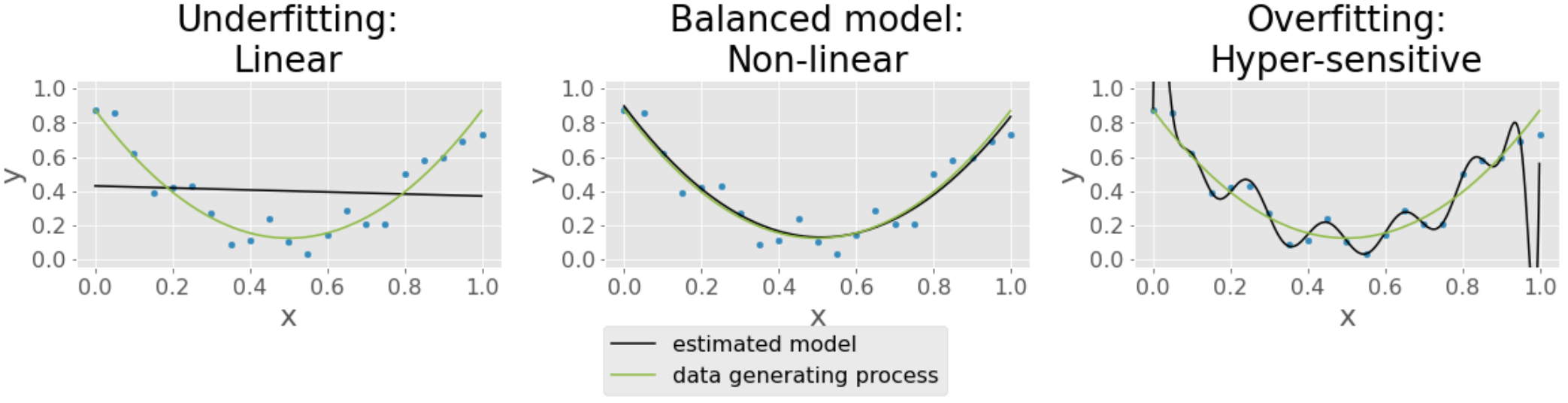


Figure 3: Varying degrees of input complexity

Can only be done by changing the input complexity (polynomials, interactions, etc.)

Regularization

We can add a term to the minimization problem which penalizes model complexity

$$\hat{w} = \underset{w}{\operatorname{argmin}} \left\{ \frac{1}{N} ||y - Xw||_2^2 + \lambda f(w) \right\}, \lambda \geq 0$$

λ is a so-called *hyperparameter*, and the values of these are chosen by us

Regularization happens implicitly or explicitly in all machine learning, as this allows us to control the bias-variance trade-off

Lasso

The penalty could be the sum of the absolute size of the weights, as in the Lasso introduced by Tisbhirani (1996)

$$\hat{w} = \underset{w}{\operatorname{argmin}} \left\{ \frac{1}{N} ||y - Xw||_2^2 + \lambda ||w||_1 \right\}, \lambda \geq 0$$

where $|| \cdot ||_1$ is the L1 or Taxicab norm, corresponding to $\sum_{i=1}^{k} |w_i|$

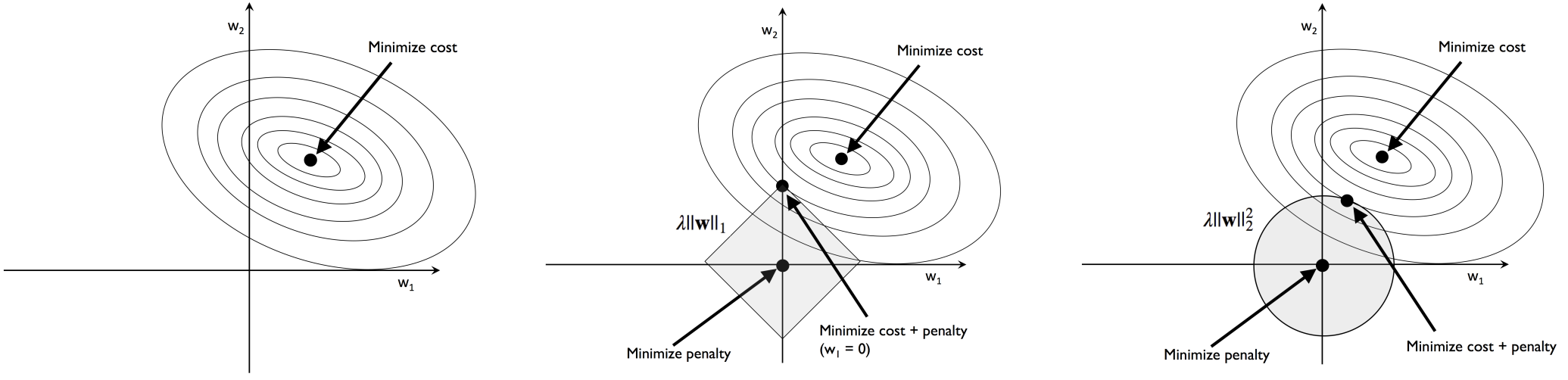
Ridge

Another penalty could be the sum of the squared weights, as in the Ridge introduced by Hoerl & Kennard (1970)

$$\hat{w} = \underset{w}{\operatorname{argmin}} \left\{ \frac{1}{N} ||y - Xw||_2^2 + \lambda ||w||_2^2 \right\}, \lambda \geq 0$$

where $|| \cdot ||_2$ again is the L2 or Euclidean norm, corresponding to $\sqrt{\sum_{i=1}^{k} w_i^2}$

A geometric interpretation



(a) OLS

(b) Lasso

(c) Ridge

Figure 4: Two-dimensional plots of cost minimization

Source: Raschka & Mirjalili, 2019, ch. 4

Scaling

As we penalize weights based on their size, the scale of each covariate matters!

Therefore we scale *all inputs* before it goes into the model

It is common to z-standardize (`StandardScaler`), which corresponds to subtracting the mean and dividing with the standard deviation

The devil is in the details

How to encode dummies?

- One-hot encoding, with **no category left out** due to regularization
- Should also be standardized!

How to include an intercept?

- Should be done after standardizing, else we have a constant column of zeros
- The model itself usually adds one, so we don’t have to worry about it

Building up to an example

StandardScaler

We fit on our train data, transform all data

```
scaler = StandardScaler()
scaler.fit(X_train)
X_train_std = scaler.transform(X_train)
X_test_std = scaler.transform(X_test)
```

This cells in slide and the next three are not executed, and cannot be run without

importing functions from `sklearn` or defining your data

PolynomialFeatures

If we wanted to make polynomial interactions, it would be the same procedure!

```
polfeats = PolynomialFeatures(degree=3, intercept=False)
polfeats.fit(X_train)
X_train_pol = polfeats.transform(X_train)
X_test_pol = polfeats.transform(X_test)
```

Predict

If we want to make predictions, we use predict instead of transform

```
regr = Lasso(alpha=lambda_value)
regr.fit(X_train)
predicted_y_train = regr.predict(X_train)
predicted_y_test = regr.predict(X_train)
```

All at once

```
scaler = StandardScaler()
polfeats = PolynomialFeatures(degree=3, intercept=False)
regr = Lasso(alpha=lambda_value)

# Polynomial features
polfeats.fit(X_train)
X_train_pol = polfeats.transform(X_train)
X_test_pol = polfeats.transform(X_test)

# Scaling
scaler.fit(X_train_pol)
X_train_std = scaler.transform(X_train_pol)
X_test_std = scaler.transform(X_test_pol)

# Model
regr.fit(X_train_std)
predicted_y_train = regr.predict(X_train_std)
predicted_y_test = regr.predict(X_test_std)
```

Takeaways

1. It’s repeating the same process multiple times
2. It’s important to remember to use the output from the last step
3. We only transform on training data!

An example with Lasso

Some data

```
 1 import numpy as np
 2 import pandas as pd
 3 import matplotlib.pyplot as plt
 4
 5 from sklearn.datasets import make_regression
 6
 7 X, y = make_regression(n_samples=10000,
 8                        n_features=15,
 9                        n_informative=3,
10                        noise=10,
11                        n_targets=1,
12                        bias=2,
13                        random_state=1)
14
15 y = y + 6 * (X[:,0] * X[:,1]) - 5 * (X[:,2] * X[:,3] * X[:,4]) + 25 * (X[:,
```

Step 1: Split data

```
from sklearn.model_selection import KFold, GridSearchCV, train_test_split,
from sklearn.preprocessing import PolynomialFeatures, StandardScaler
from sklearn.pipeline import Pipeline
from sklearn.linear_model import Lasso, LinearRegression
from sklearn.metrics import mean_squared_error
# Use mean_squared_error, Lasso, KFold, train_test_split, PolynomialFeature

# Development and test data
X_dev, X_test,  y_dev, y_test = train_test_split(X, y, train_size=0.75, ran
```

Step 2: Cross validation

To reiterate what's going to happen

```
for each lambda:
  for each fold:
    fit preprocess on training data
    transform training data
    transform validation data
    fit model on transformed training data
    predict on transformed validation data
    get score on validation data
    save score
  save mean score for all folds
```

```
# Splits data into 5
kf = KFold(n_splits=5)

# Hyperparameterspace
lambdas = np.logspace(-4,3,10)

# Mean MSE for each lambda
mean_MSE_train = []
mean_MSE_val = []

# For each hyperparameter...
for lambda_ in lambdas:

    MSE_val_list = []
    MSE_train_list = []

    # For each fold...
    for train_index, test_index in kf.split(X_dev):
```

Step 3: Admire your output

```
1 lambda_df
```

	Lambda	MSE Training	MSE Validation
0	0.000100	86.010738	121.000621
1	0.000599	86.011340	120.800925
2	0.003594	86.032232	119.655928
3	0.021544	86.648761	114.128397
4	0.129155	93.965888	102.462882
5	0.774264	102.925777	103.214830
6	4.641589	227.640247	228.388808

	Lambda	MSE Training	MSE Validation
7	27.825594	2978.498180	2980.176269
8	166.810054	11912.563899	11917.745461
9	1000.000000	11912.563899	11917.745461

You may feel somewhat overwhelmed

That was a lot of code for finding the best hyperparamter!

You asked for a recipe:

- I gave you a cookbook

We can do better!

Pipeline

Let’s make life easier

There was a whole lot of `fit`, `transform`, `fit`, `transform` in there…

We must be able to remove all this boilerplate!

This is exactly what the pipeline does:

- Applies an arbitrary amount of `fit` and `transform` and (can) finish with a `fit` and `predict` step, i.e. a regression or classification model

Why?

1. Greatly reduces the amount of code
2. Reduces room for stupid mistakes

What do they look like?

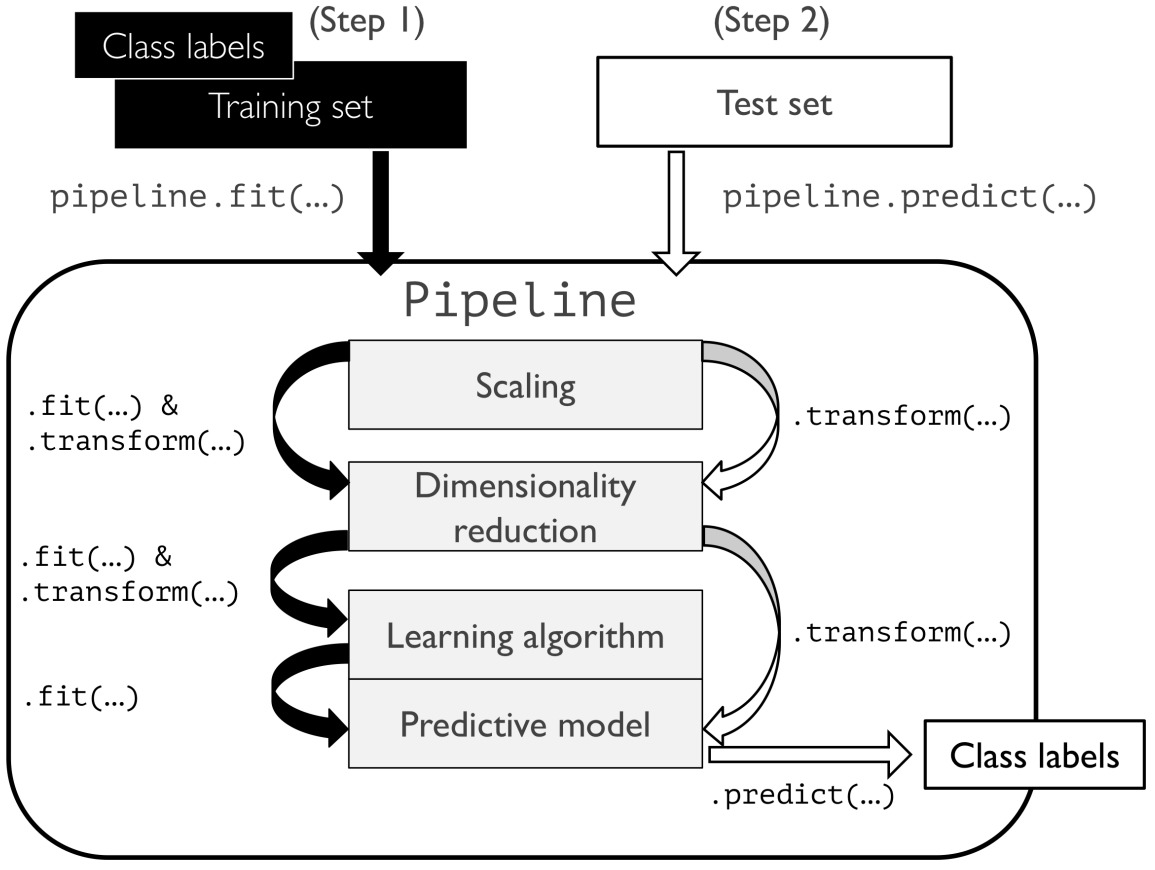


Figure 5: Pipeline

Source: Raschka & Mirjalili, 2019, ch. 6

They’re very flexible

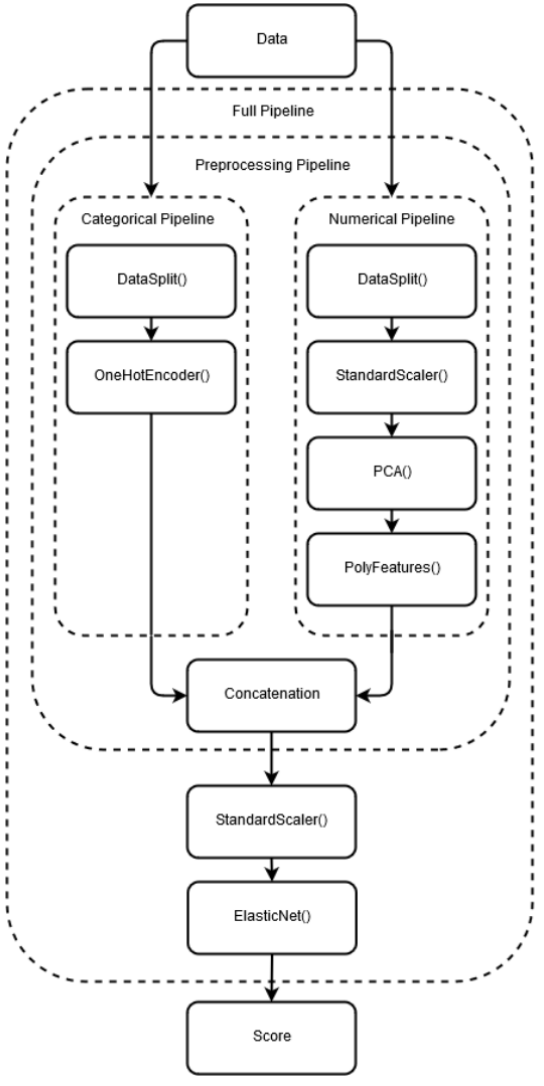


Figure 6: A pipeline from previous work

Python

```
# Splits data into 5
kf = KFold(n_splits=5)

# Mean MSE for each lambda
mean_MSE_train = []
mean_MSE_val = []

# Hyperparameterspace
lambdas = np.logspace(-4,3,10)

# For each hyperparameter...
for lambda_ in lambdas:

    MSE_val_list = []
    MSE_train_list = []

    # For each fold...
    for train_index, test_index in kf.split(X_dev):
```

Admire your output again

```
1 lambda_df_pipe
```

	Lambda	MSE Training	MSE Validation
0	0.000100	86.010738	121.000621
1	0.000599	86.011340	120.800925
2	0.003594	86.032232	119.655928
3	0.021544	86.648761	114.128397
4	0.129155	93.965888	102.462882
5	0.774264	102.925777	103.214830
6	4.641589	227.640247	228.388808

	Lambda	MSE Training	MSE Validation
7	27.825594	2978.498180	2980.176269
8	166.810054	11912.563899	11917.745461
9	1000.000000	11912.563899	11917.745461

Now to predict

We found the best model, but haven't tested on our test data yet!

```
# GET BEST LAMBDA
idx_min = lambda_df['MSE Validation'].idxmin()
best_lambda = lambda_df.iloc[idx_min, 0]

# MAKE PIPELINE WITH BEST LAMBDA
pipeline = Pipeline(
            [
            ('pol_feats', PolynomialFeatures(degree=3, include_bias=False
            ('scaler', StandardScaler()),
            ('lasso', Lasso(alpha=best_lambda, random_state=1))
        ]
            )

# FIT ON DEVELOPMENT
pipeline.fit(X_dev, y_dev)

# PREDICT ON TEST

y_test_hat = pipeline.predict(X_test)
```

MSE on test: 100.76

Quick aside: Are we beating OLS?

```
pipeline = Pipeline(
            [
            ('pol_feats', PolynomialFeatures(degree=3, include_bias=False))
            ('scaler', StandardScaler()),
            ('ols', LinearRegression())
        ]
            )

# FIT ON DEVELOPMENT
pipeline.fit(X_dev, y_dev)

# PREDICT ON TEST
y_test_hat = pipeline.predict(X_test)
print(f"MSE on test: {mean_squared_error(y_test, y_test_hat):.2f}")
```

```
MSE on test: 112.55
```

Thank god!

Gridsearch

Let’s make life easier, once again

There was a whole lot of `for loops` in there, both iterating over folds and hyperparameters values

This seems repetetive and verbose

- It’s the same across all models, and could probably be automated

This is exactly what the Gridsearch does:

- Input a splitting method (default *K* fold), a pipeline and a hyperparametergrid
- Ouput: The best hyperparameters

We've reached our destination

```
# Hyperparameterspace
lambdas = np.logspace(-4,3,10)

# Pipeline
pipeline = Pipeline(
            [
            ('pol_feats', PolynomialFeatures(degree=3, include_bias=False))
            ('scaler', StandardScaler()),
            ('lasso', Lasso(random_state=1)) # No lambda
        ]
            )

# Gridsearch
gs = GridSearchCV(estimator=pipeline,
                  param_grid=[{'lasso__alpha':lambdas}],
                  scoring='neg_mean_squared_error',
                  cv=5,
                  n_jobs=-1)
```

```
{'lasso__alpha': 0.1291549665014884}
MSE on test: 100.76
```

A general recipe

- Step 0: Split your data
- Step 1: Create a pipeline
- Step 2: Define your hyperparameterspace
- Step 3: Do a search over your hyperparameterspace on development set
- Step 4: Evaluate on test set

Curves

A visual method of diagnosing over- and underfitting

Learning and validation curves tells us how our model performs for either:

- Different sample sizes
- Different hyperparameter values

Not an important output per se, but very helpful while building models!

- The performance on the holdout test data is the main metric

`sklearn` has a short write-up with code and Raschka & Mirjalili write about them in chapter 6.

Learning curve

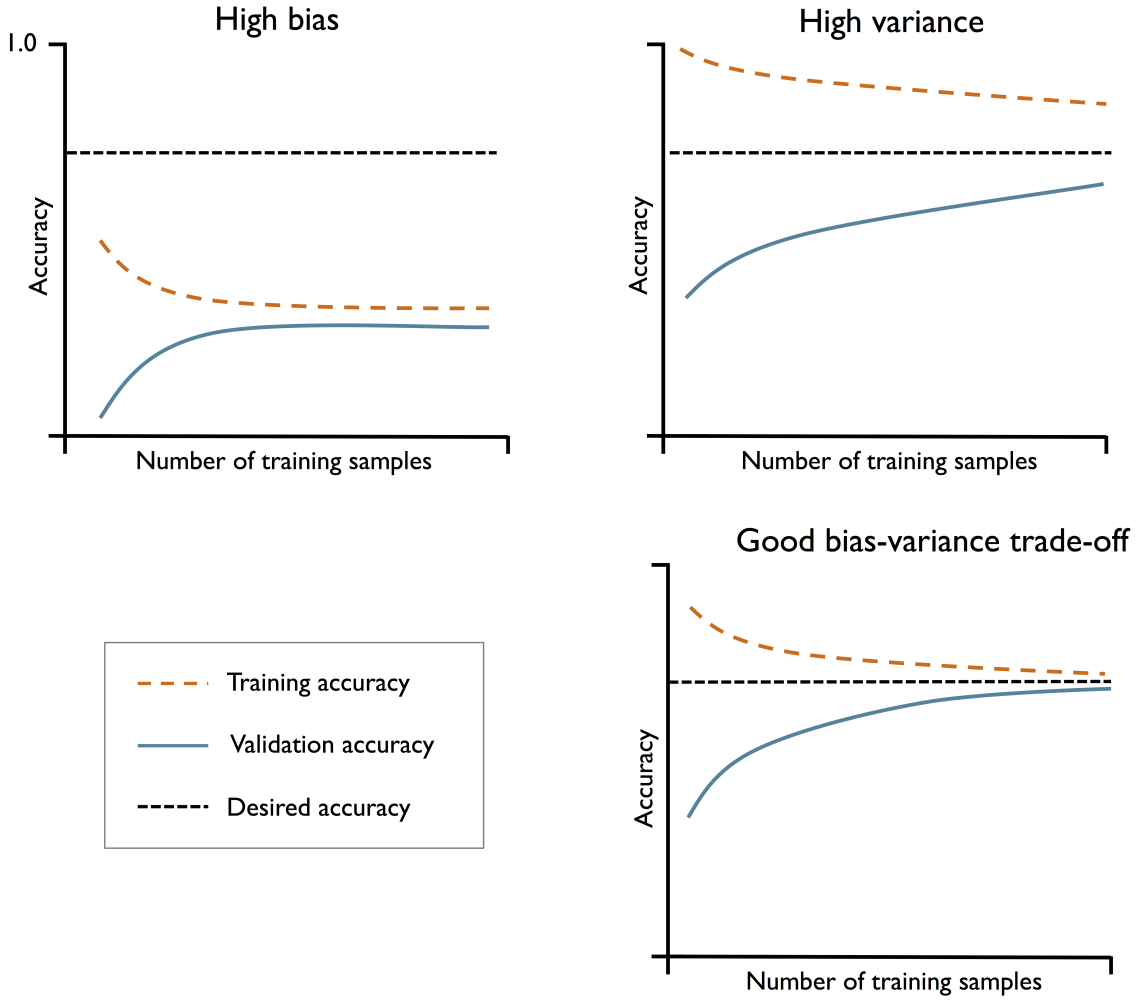
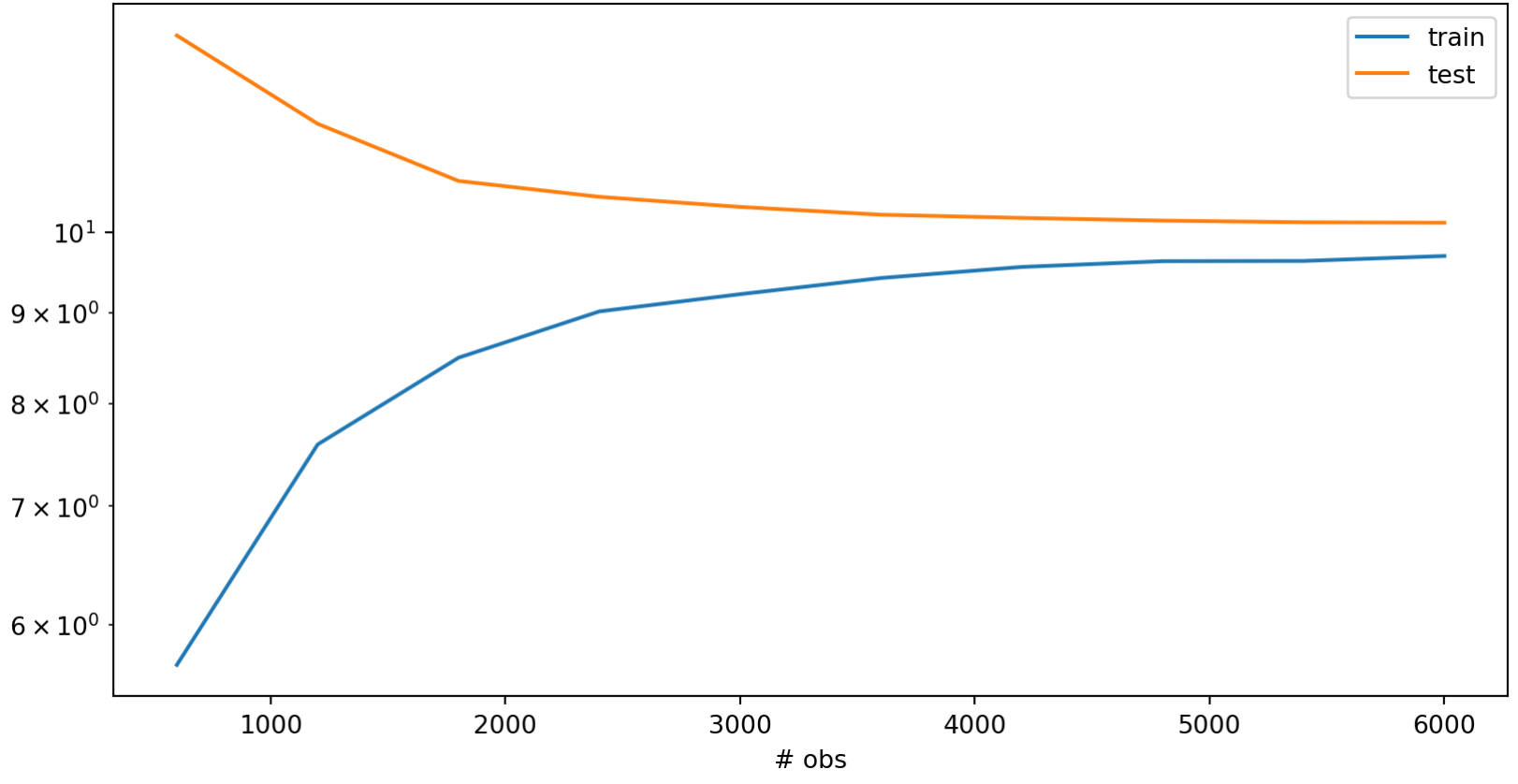


Figure 7: Learning curve

Source: Raschka & Mirjalili, 2019, ch. 6

Python

- Show code



Validation curve

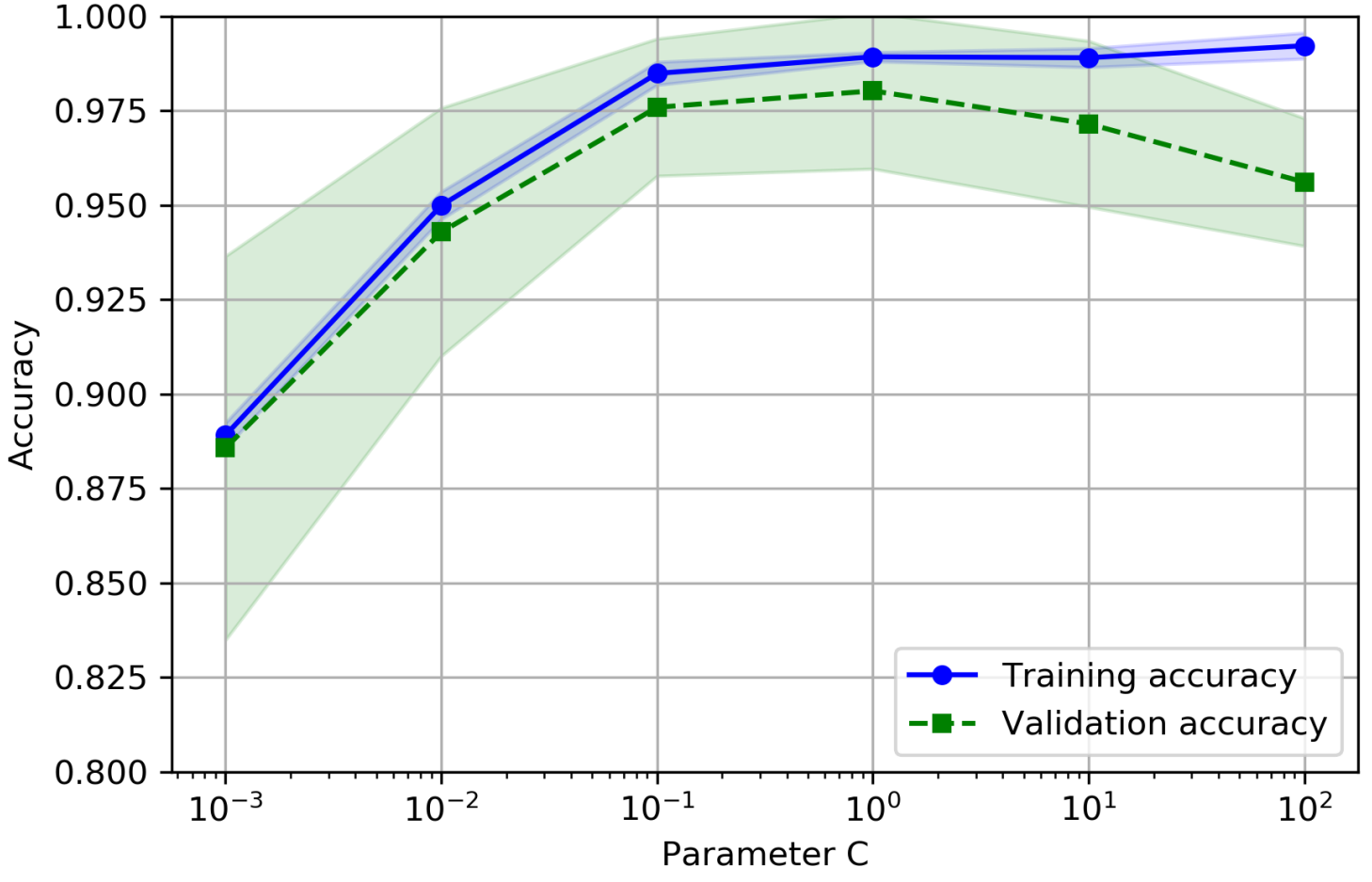
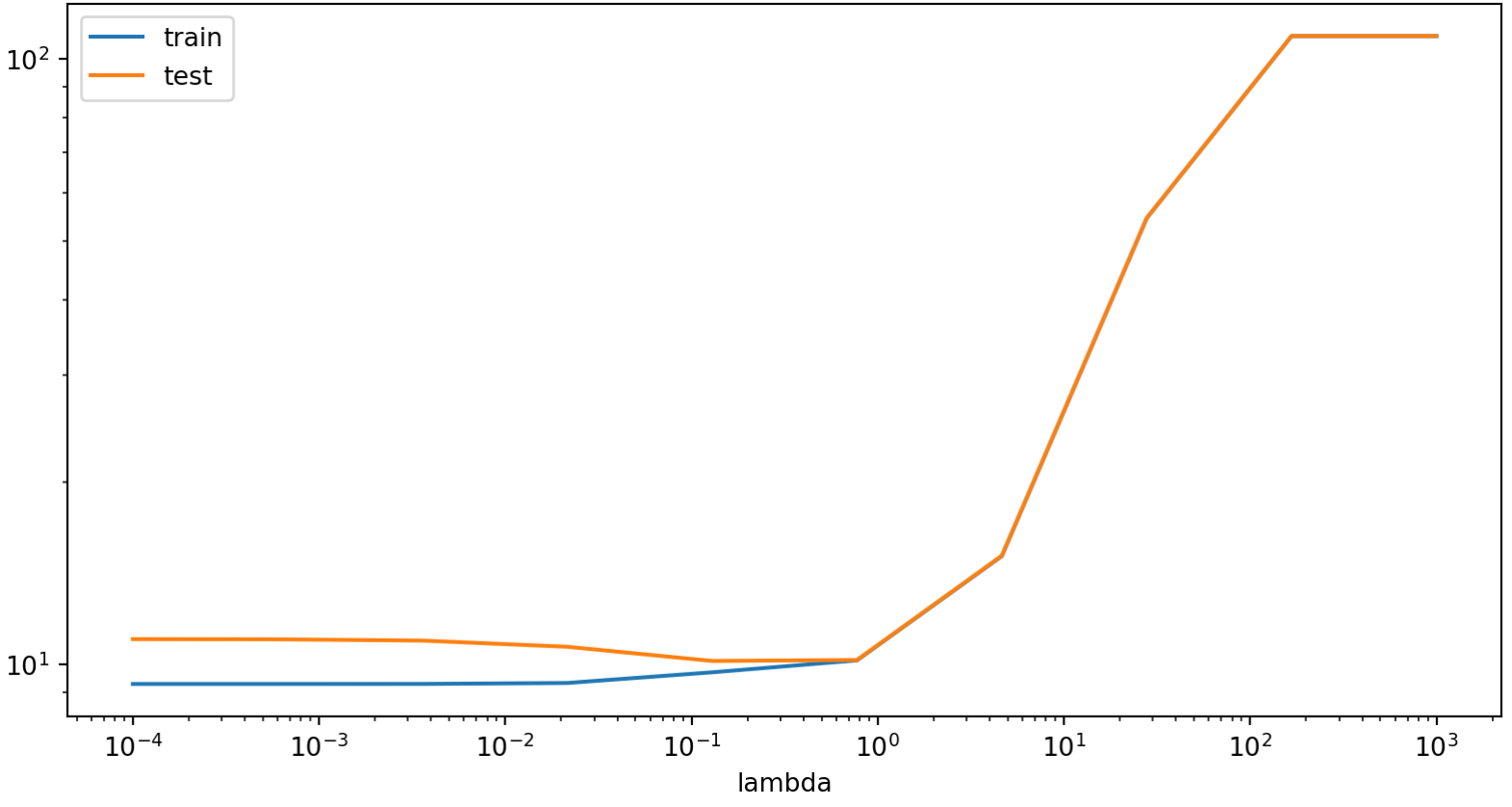


Figure 8: Validation curve

Source: Raschka & Mirjalili, 2019, ch. 6

Python

▶ Show code



References

Bengio, Y., & Grandvalet, Y. (2003). No unbiased estimator of the variance of k-fold cross-validation. Advances in Neural Information Processing Systems, 16.

Hoerl, A. E., & Kennard, R. W. (1970). Ridge regression: Biased estimation for nonorthogonal problems. Technometrics, 12(1), 55-67.

Nadeau, C., & Bengio, Y. (1999). Inference for the generalization error. Advances in neural information processing systems, 12.

Raschka, S., & Mirjalili, V. (2019). Python machine learning: Machine learning and deep learning with Python, scikit-learn, and TensorFlow 2. Packt Publishing Ltd.

Tibshirani, R. (1996). Regression shrinkage and selection via the lasso. Journal of the Royal Statistical Society: Series B (Methodological), 58(1), 267-288.

To the exercises!