Generalisation

APT 3025: APPLIED MACHINE LEARNING

Notes taken from Introduction to Machine Learning with Python by Andreas C. Müller & Sarah Guido, O'Reilley Media, 2016

Generalization

- In supervised learning, we want to build a model on the training data and then be able to make accurate predictions on new, unseen data that has the same characteristics as the training set that we used.
- If a model is able to make accurate predictions on unseen data (held in a test set), we say it is able to generalize from the training set to the unseen data.
- We want to build a model that is able to generalize as accurately as possible.

Generalization

- Usually we build a model in such a way that it can make accurate predictions on the training set.
- If the training and test sets have enough in common, we expect the model to also be accurate on the test set.
- However, there are some cases where this can go wrong. For example, if we allow ourselves to build very complex models, we can always be as accurate as we like on the training set.

Example

- Say a novice data scientist wants to predict whether a customer will buy a boat, given records of previous boat buyers and customers who we know are not interested in buying a boat.
- The goal is to send out promotional emails to people who are likely to actually make a purchase, but not bother those customers who won’t be interested.
- Suppose we have the customer data shown below.

Age	Number of cars owned	Owns house	Number of children	Marital status	Owns a dog	Bought a boat
66	1	yes	2	widowed	no	yes
52	2	yes	3	married	no	yes
22	0	no	0	married	yes	no
25	1	no	1	single	no	no
44	0	no	2	divorced	yes	no
39	1	yes	2	married	yes	no
26	1	no	2	single	no	no
40	3	yes	1	married	yes	no
53	2	yes	2	divorced	no	yes
64	2	yes	3	divorced	no	no
58	2	yes	2	married	yes	yes
33	1	no	1	single	no	no

Notes taken from Introduction to Machine Learning with Python by Andreas C. Müller & Sarah Guido, O'Reilley Media, 2016

Inferring Rules that Seem Accurate

- After looking at the data for a while, our novice data scientist comes up with the following rule: “If the customer is older than 45, and has less than 3 children or is not divorced, then they want to buy a boat.”
- Indeed, on the data that is in the table, the rule is perfectly accurate.
- Another possible rule is “people who are 66, 52, 53, or 58 years old want to buy a boat, while all others don’t.” And there are others.

Predicting Training Data Not The Aim

- However, we already know the answers for this dataset. We don’t need a rule for predicting them.
- We instead want to find a rule that will work well for new customers, and achieving 100 percent accuracy on the training set can be counterproductive.
- We might not expect that the rule our data scientist came up with will work very well on new customers.
- It seems too complex, and it is supported by very little data. For example, the “or is not divorced” part of the rule hinges on a single customer (row 2).

Keep it Simple

- The only measure of whether an algorithm will perform well on new data is the evaluation on the test set.
- However, intuitively we expect simple models to generalize better to new data.
- If the rule was “People older than 50 want to buy a boat,” and this would explain the behavior of all the customers, we would trust it more than the rule involving children and marital status in addition to age.

Overfitting

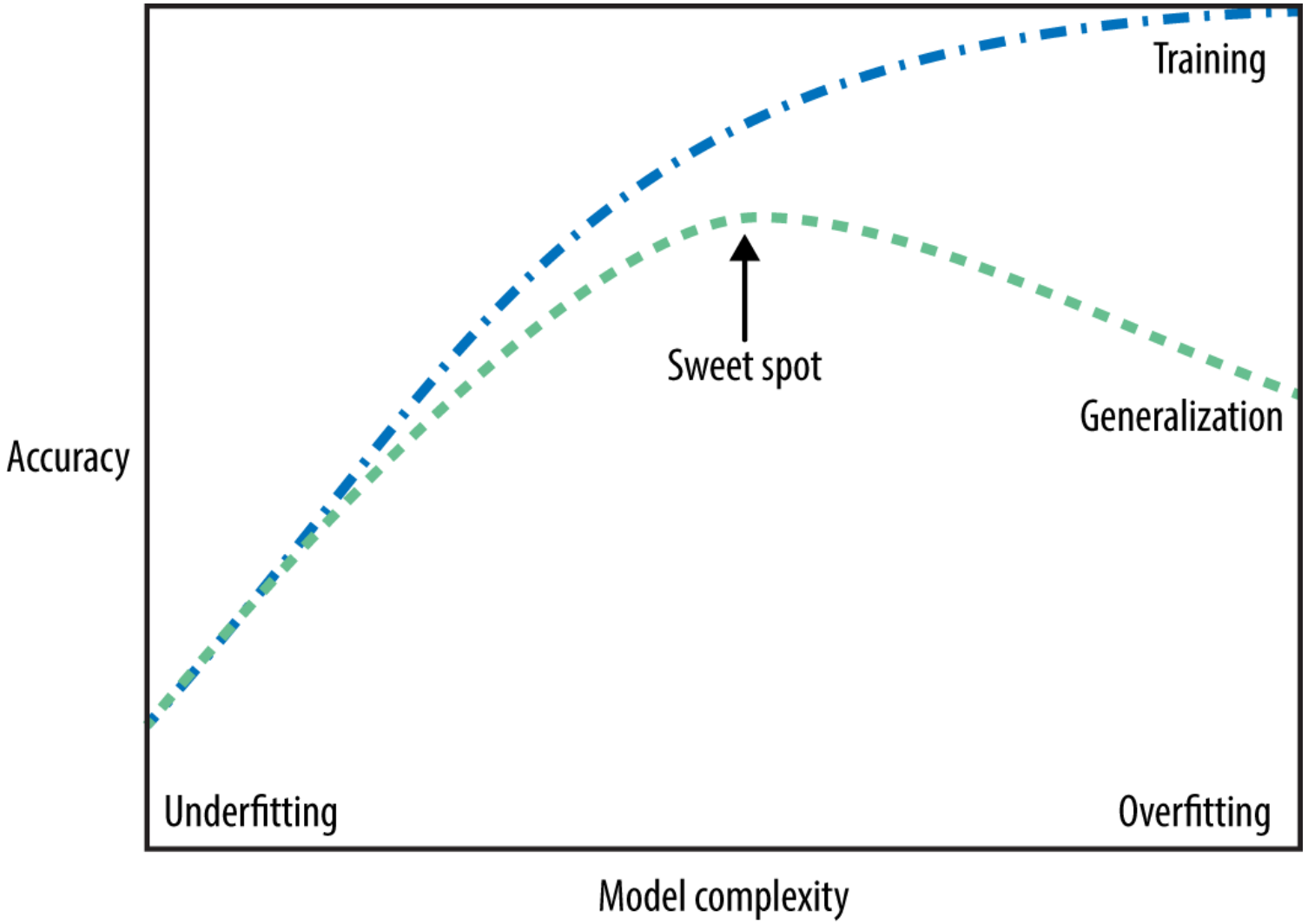
- We always want to find the simplest model. Building a model that is too complex for the amount of information we have, as our novice data scientist did, is called overfitting.
- Overfitting occurs when you fit a model too closely to the particularities of the training set and obtain a model that works well on the training set but is not able to generalize to new data.

Underfitting

- On the other hand, if your model is too simple—say, “Everybody who owns a house buys a boat”—then you might not be able to capture all the aspects of and variability in the data.
- Such a model will do badly even on the training set.
- Choosing too simple a model is called underfitting.

Striking a Balance

- The more complex we allow our model to be, the better we will be able to predict on the training data.
- However, if our model becomes too complex, we start focusing too much on each individual data point in our training set, and the model will not generalize well to new data.
- There is a balance between overfitting and underfitting that will yield the best generalization performance. This is the model we want to find.



Notes taken from Introduction to Machine Learning with Python by Andreas C. Müller & Sarah Guido, O'Reilley Media, 2016

Model Complexity and Dataset Size

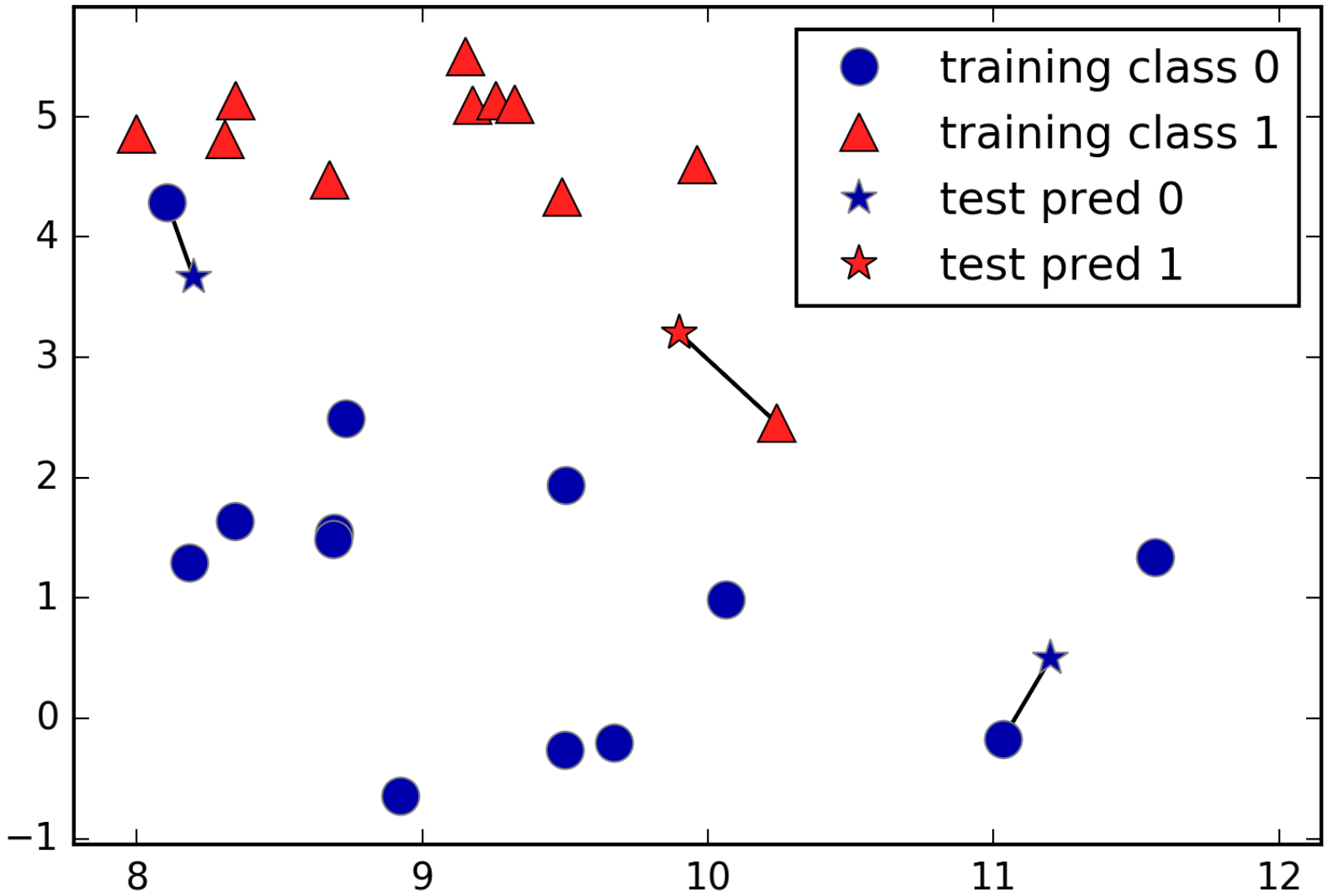
- Model complexity is intimately tied to the variation of inputs contained in your training dataset: the larger the variety of data points, the more complex a model you can use without overfitting.
- Usually, collecting more data points will yield more variety, so larger datasets allow building more complex models.
- However, simply duplicating the same data points or collecting very similar data will not help.

Model Complexity and Dataset Size

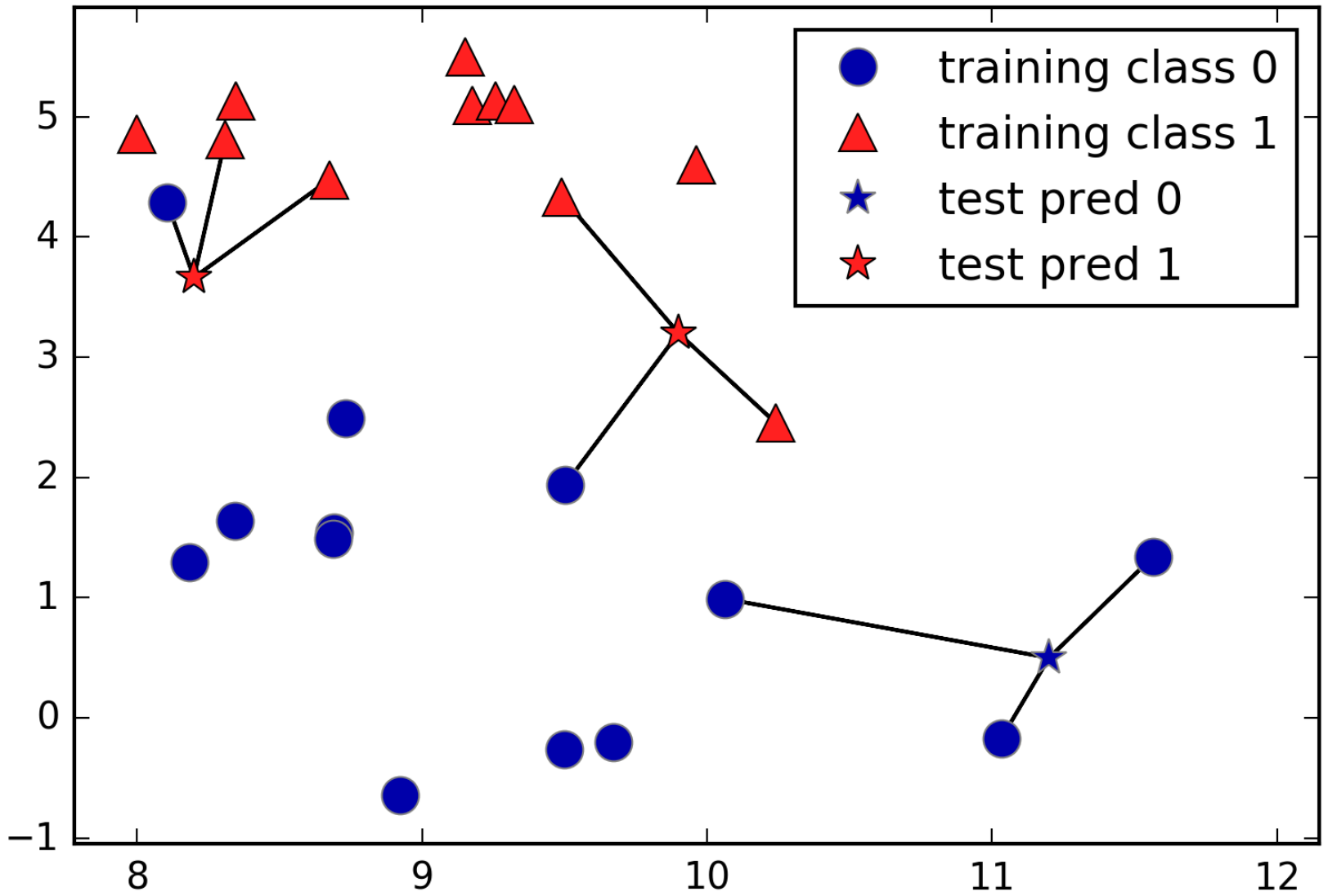
- Going back to the boat selling example, if we saw 10,000 more rows of customer data, and all of them complied with the rule “If the customer is older than 45, and has less than 3 children or is not divorced, then they want to buy a boat,” we would be much more likely to believe this to be a good rule than when it was developed using only the 12 rows.

The k-NN Classification Algorithm

- The k-NN algorithm is arguably the simplest machine learning algorithm.
- Building the model consists only of storing the training dataset.
- To make a prediction for a new data point, the algorithm finds the closest data points in the training dataset—its “nearest neighbors.”
- In its simplest version, the k-NN algorithm only considers exactly one nearest neighbor, which is the closest training data point to the point we want to make a prediction for.
- Next we illustrate kNN is with 1 neighbour and then 3 neighbours.



Notes taken from Introduction to Machine Learning with Python by Andreas C. Müller & Sarah Guido, O'Reilley Media, 2016

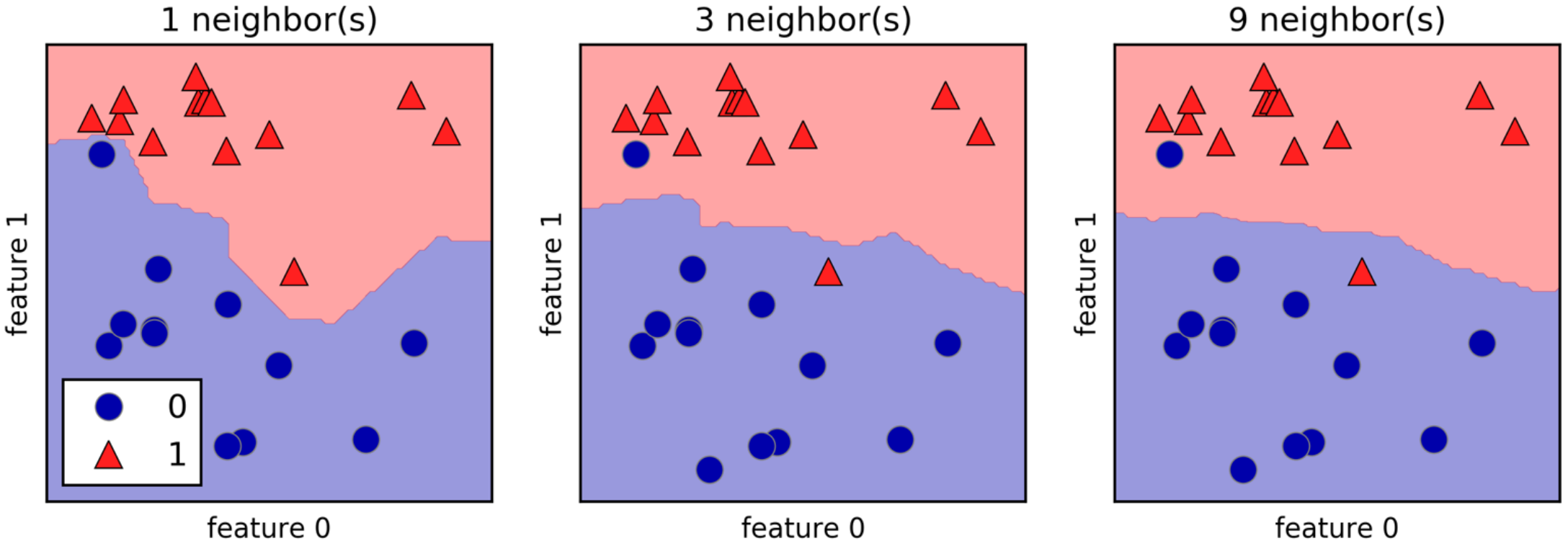


Notes taken from Introduction to Machine Learning with Python by Andreas C. Müller & Sarah Guido, O'Reilley Media, 2016

Analyzing the KNN Classifier

- For two-dimensional datasets, we can also illustrate the prediction for all possible test points in the xy-plane.
- We color the plane according to the class that would be assigned to a point in this region.
- This lets us view the decision boundary, which is the divide between where the algorithm assigns class 0 versus where it assigns class 1.
- On the next slide, we show the visualizations for 1, 3 and 9 neighbors for a fictitious dataset.

Decision Boundaries for kNN, k = 1, 3, 9



Notes taken from Introduction to Machine Learning with Python by Andreas C. Müller & Sarah Guido, O'Reilley Media, 2016

Number of Neighbors and Model Complexity

- As you can see on the left in the figure, using a single neighbor results in a decision boundary that follows the training data closely.
- Considering more and more neighbors leads to a smoother decision boundary.
- A smoother boundary corresponds to a simpler model, which generalizes better.
- Using fewer neighbors results in a more complex model, which tends to overfit.

Effect of Number of Neighbors for a Real Dataset

- Let's investigate whether we can confirm the connection between model complexity and generalization with the real-world Breast Cancer dataset.

```
from sklearn.datasets import load_breast_cancer

cancer = load_breast_cancer()
X_train, X_test, y_train, y_test = train_test_split(
    cancer.data, cancer.target, stratify=cancer.target, random_state=66)
```

Notes taken from Introduction to Machine Learning with Python by Andreas C. Müller & Sarah Guido, O'Reilley Media, 2016

Effect of Number of Neighbors for a Real Dataset

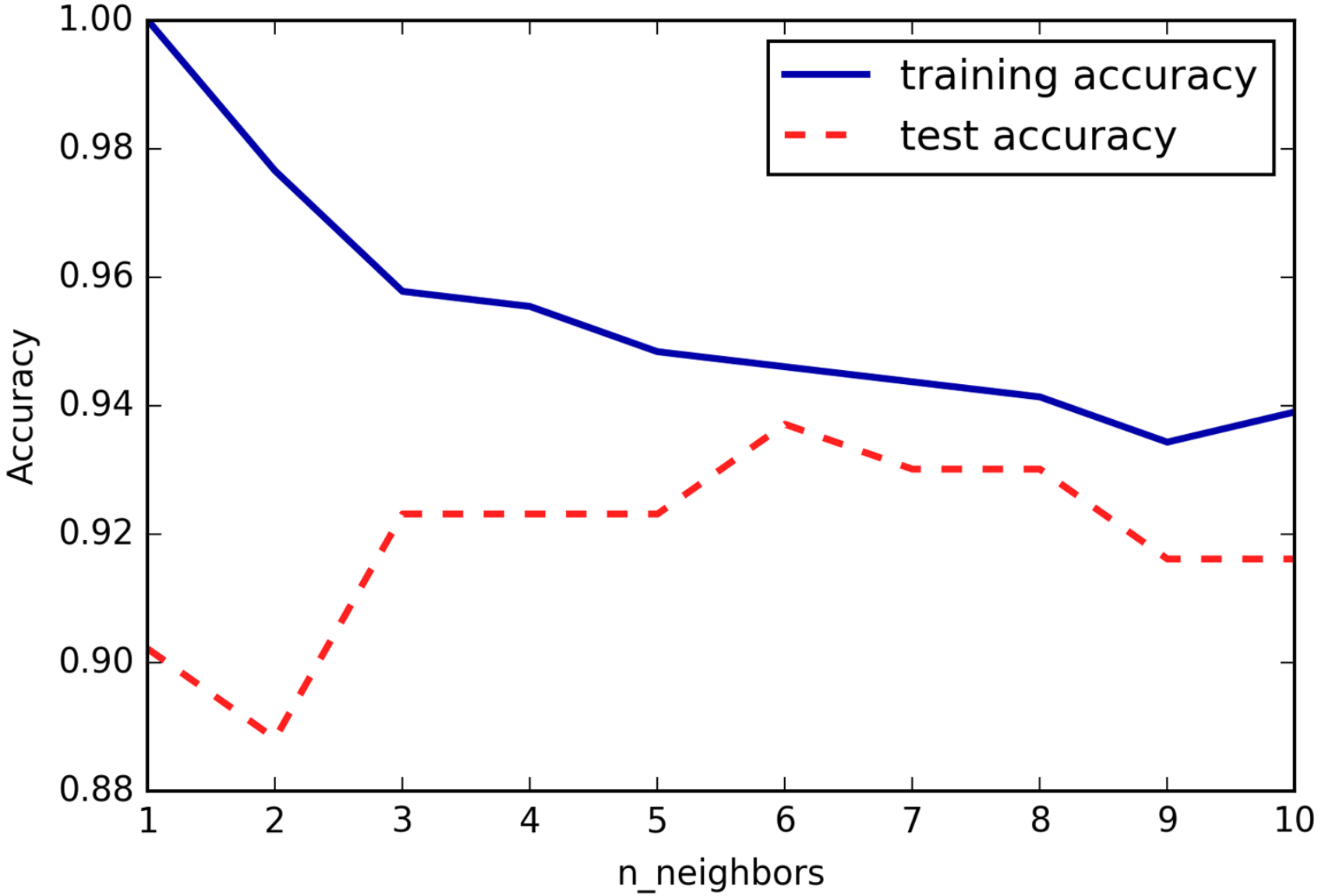
```
training_accuracy = []
test_accuracy = []
# try n_neighbors from 1 to 10
neighbors_settings = range(1, 11)

for n_neighbors in neighbors_settings:
    # build the model
    clf = KNeighborsClassifier(n_neighbors=n_neighbors)
    clf.fit(X_train, y_train)
    # record training set accuracy
    training_accuracy.append(clf.score(X_train, y_train))
    # record generalization accuracy
    test_accuracy.append(clf.score(X_test, y_test))
```

Notes taken from Introduction to Machine Learning with Python by Andreas C. Müller & Sarah Guido, O'Reilley Media, 2016

Effect of Number of Neighbors for a Real Dataset

```
plt.plot(neighbors_settings, training_accuracy, label="training accuracy")
plt.plot(neighbors_settings, test_accuracy, label="test accuracy")
plt.ylabel("Accuracy")
plt.xlabel("n_neighbors")
plt.legend()
```

Notes taken from Introduction to Machine Learning with Python by Andreas C. Müller & Sarah Guido, O'Reilley Media, 2016



Notes taken from Introduction to Machine Learning with Python by Andreas C. Müller & Sarah Guido, O'Reilley Media, 2016

Effect of Number of Neighbors for a Real Dataset

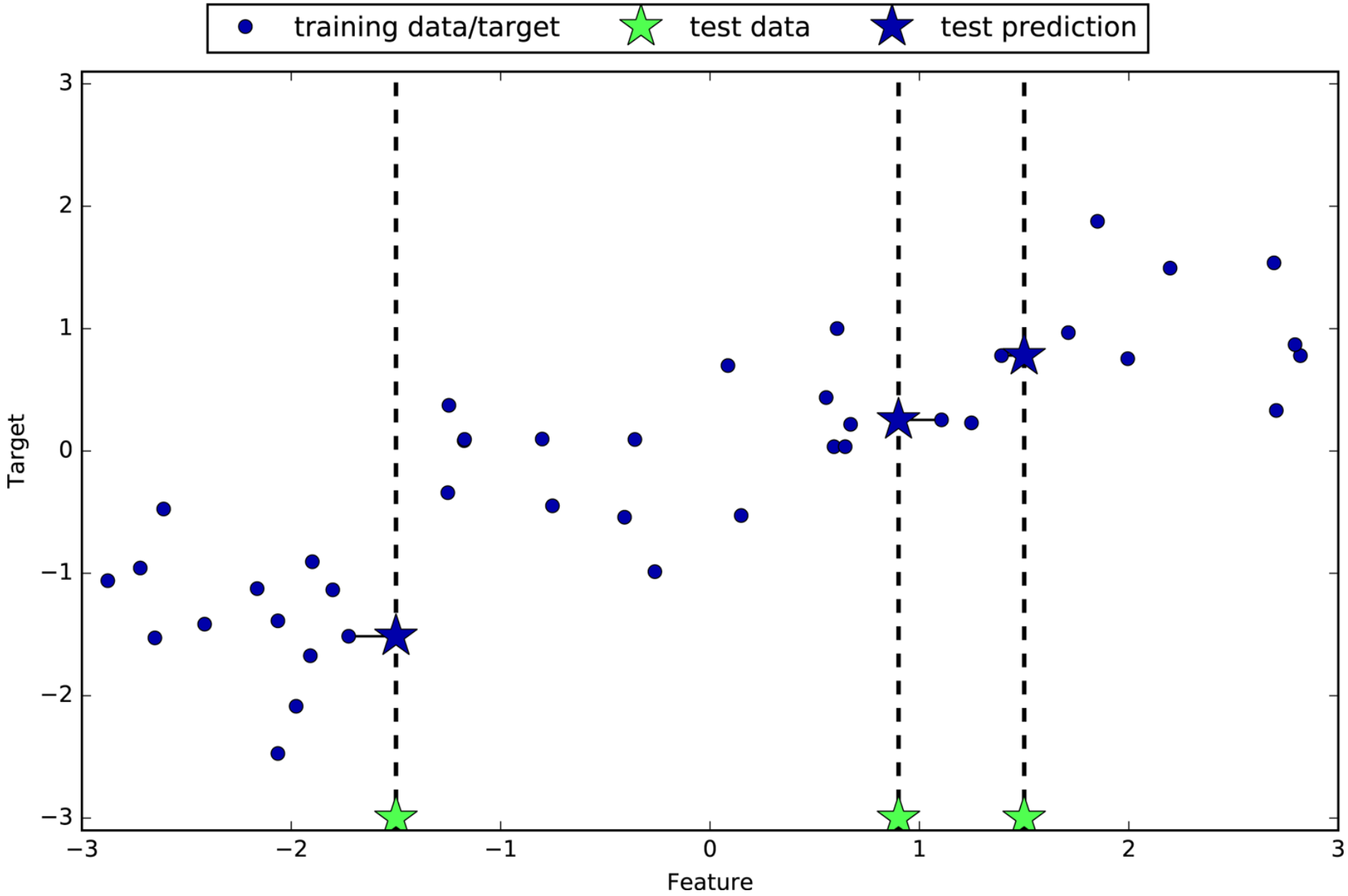
- The plot shows the training and test set accuracy on the y-axis against the number of neighbors on the x-axis.
- While real-world plots are rarely very smooth, we can still recognize some of the characteristics of overfitting and underfitting
- Considering a single nearest neighbor, the prediction on the training set is perfect.
- But when more neighbors are considered, the model becomes simpler and the training accuracy drops.

Effect of Number of Neighbors for a Real Dataset

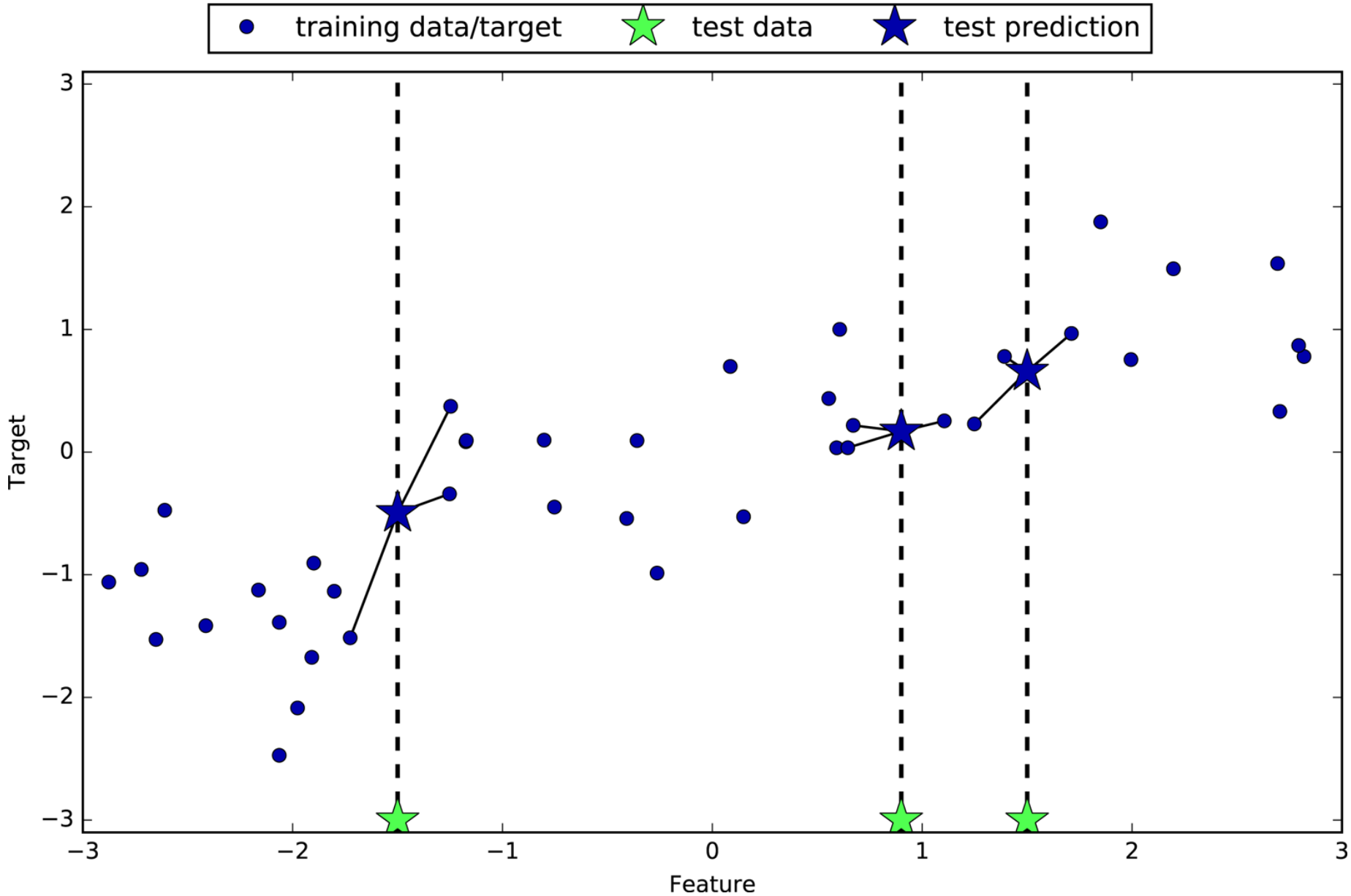
- The test set accuracy for using a single neighbor is lower than when using more neighbors, indicating that using the single nearest neighbor leads to a model that is too complex.
- On the other hand, when considering 10 neighbors, the model is too simple and performance is even worse.
- The best performance is somewhere in the middle, using around six neighbors.

Regression using kNN

- There is also a regression variant of the k-nearest neighbors algorithm.
- Three test data points are shown as green stars on the x-axis.
- The prediction using a single neighbor is just the target value of the nearest neighbor.
- These are shown as blue stars on the next slide.
- When using multiple nearest neighbors (e.g., 3), the prediction is the average, or mean, of the relevant neighbors.



Notes taken from Introduction to Machine Learning with Python by Andreas C. Müller & Sarah Guido, O'Reilley Media, 2016



Notes taken from Introduction to Machine Learning with Python by Andreas C. Müller & Sarah Guido, O'Reilley Media, 2016

Abalone Age Prediction Using kNN

```
import pandas as pd
url = "https://archive.ics.uci.edu/ml/"
"machine-learning-databases/abalone/abalone.data"
abalone = pd.read_csv(url, header=None)
```

```
abalone.head()
```

	0	1	2	3	4	5	6	7	8
0	M	0.455	0.365	0.095	0.5140	0.2245	0.1010	0.150	15
1	M	0.350	0.265	0.090	0.2255	0.0995	0.0485	0.070	7
2	F	0.530	0.420	0.135	0.6770	0.2565	0.1415	0.210	9
3	M	0.440	0.365	0.125	0.5160	0.2155	0.1140	0.155	10
4	I	0.330	0.255	0.080	0.2050	0.0895	0.0395	0.055	7

The info and describe methods are also very useful for learning more about the data.

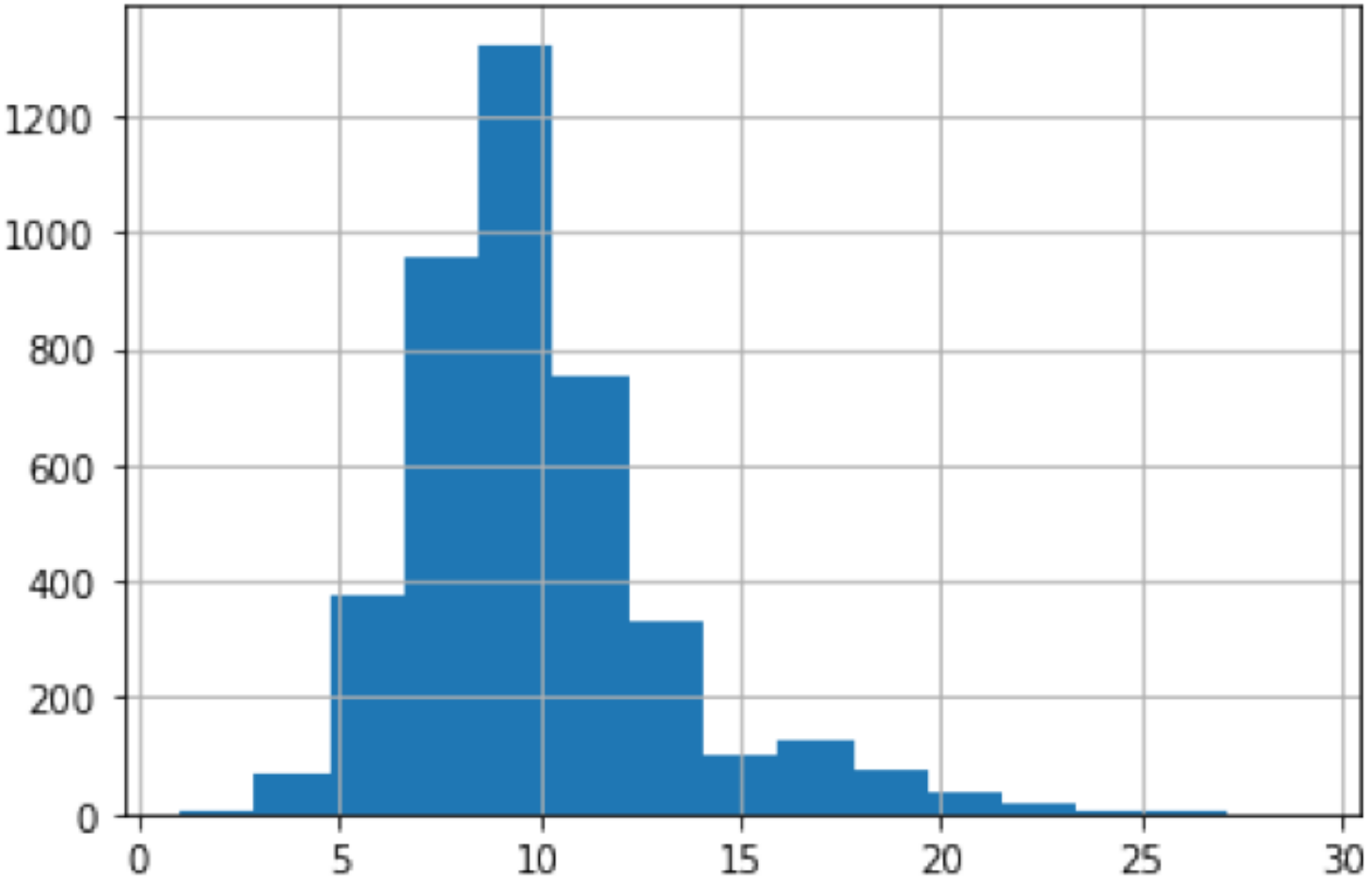
Notes taken from Introduction to Machine Learning with Python by Andreas C. Müller & Sarah Guido, O'Reilley Media, 2016

```
abalone.columns=["Sex", "Length", "Diameter", "Height", "Whole weight",
                 "Shucked weight", "Viscera weight", "Shell weight", "Rings" ]
```

```
abalone = abalone.drop("Sex", axis=1)
```

```
# Distribution of target variable
import matplotlib.pyplot as plt
abalone["Rings"].hist(bins=15)
```

Histogram of target feature



A long tail distribution is problematic for machine learning. It means there are many categories that are under-represented, which makes learning about them difficult.

Notes taken from Introduction to Machine Learning with Python by Andreas C. Müller & Sarah Guido, O'Reilley Media, 2016

```
# Find out aboutt correlations between the features and the target
correlation_matrix = abalone.corr()
correlation_matrix["Rings"]
```

```
Length            0.556720
Diameter          0.574660
Height            0.557467
Whole weight      0.540390
Shucked weight    0.420884
Viscera weight    0.503819
Shell weight      0.627574
Rings             1.000000
Name: Rings, dtype: float64
```

```
# Extract the data from the pandas dataframe
X = abalone.drop("Rings", axis=1)    # Exclude target
X = X.values
y = abalone["Rings"]    # Extract the target
y = y.values
```

```
# Split the data into training and test sets
from sklearn.model_selection import train_test_split
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2,␣
 ↪random_state=12345)
```

Notes taken from Introduction to Machine Learning with Python by Andreas C. Müller & Sarah Guido, O'Reilley Media, 2016

```
# Create a kNN regression model
from sklearn.neighbors import KNeighborsRegressor
knn_model = KNeighborsRegressor(n_neighbors=3)
knn_model.fit(X_train, y_train)
```

```
KNeighborsRegressor(n_neighbors=3)
```

```
# Predict for training data and calculate prediction error
from sklearn.metrics import mean_squared_error
from math import sqrt
train_preds = knn_model.predict(X_train)
mse = mean_squared_error(y_train, train_preds)
rmse = sqrt(mse)
print(rmse)
```

```
1.653705966446084
```

```
# Predict for test data and calsculate prediciton error
# If lower then the model can generalise well
# If higher then our model is overfitting
test_preds = knn_model.predict(X_test)
mse = mean_squared_error(y_test, test_preds)
rmse = sqrt(mse)
print(rmse)
```

```
2.375417924000521
```

Using Grid Search to Tune a Hyperparameter

- The error rate of our model is a lot lower on the training data than on the test data.
- This means that it is overfitting badly. We need to try and improve its performance.
- One parameter we can change is the number of neighbors. We can use grid search to try many different values of k and then choose the best k.

Grid Search

```
from sklearn.model_selection import GridSearchCV
parameters = {"n_neighbors": range(1, 50)}
gridsearch = GridSearchCV(KNeighborsRegressor(), parameters)
gridsearch.fit(X_train, y_train) # Train using the best k found
```

```
GridSearchCV(estimator=KNeighborsRegressor(),
             param_grid={'n_neighbors': range(1, 50)})
```

```
gridsearch.best_params_
```

```
{'n_neighbors': 25}
```

Notes taken from Introduction to Machine Learning with Python by Andreas C. Müller & Sarah Guido, O'Reilley Media, 2016

```
train_preds_grid = gridsearch.predict(X_train)
train_mse = mean_squared_error(y_train, train_preds_grid)
train_rmse = sqrt(train_mse)
test_preds_grid = gridsearch.predict(X_test)
test_mse = mean_squared_error(y_test, test_preds_grid)
test_rmse = sqrt(test_mse)
print("Train error:", train_rmse)
print("Test error:", test_rmse)
```

```
Train error: 2.0731180327543384
Test error: 2.1700197339962175
```

Root Mean Square Error

- Because regression predicts a number from a continuous distribution, it makes more sense to use root mean square error (RMSE) for evaluation than accuracy.
- RMSE takes the differences between the actual and the predicted, squares them, takes the average, and finally the square root. E.g.:
  - Actual: [1, 2, 3, 4, 5]
  - Predicted: [1, 1, 2, 5, 3]
  - RMSE = sqrt(0 + 1 + 1 + 1 + 4)/5