Supervised Learning – Part 2

ESM3081 Programming for Data Science

Seokho Kang

성균관대학교
SUNGKYUNKWAN UNIVERSITY(SKKU)

Learning algorithms covered in this course

- **Supervised Learning** (Classification/Regression)
  - **K-Nearest Neighbors**
  - Linear Models (Logistic/Linear Regression)
  - Decision Trees
  - Random Forests
  - Support Vector Machines
  - Neural Networks

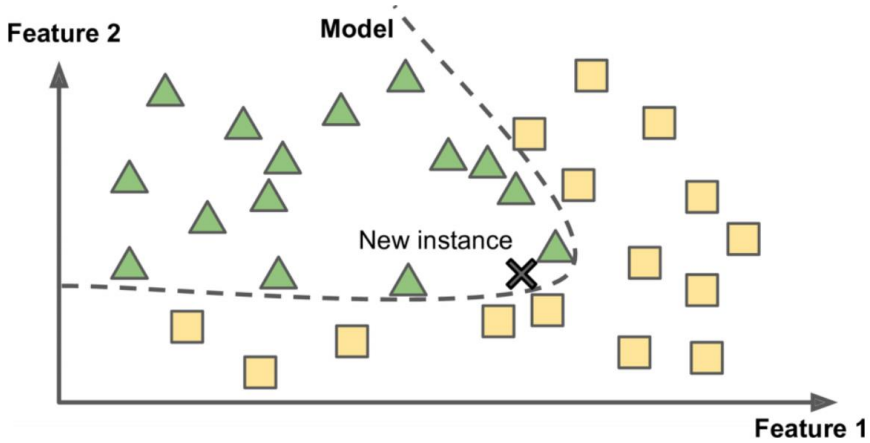
** Many algorithms have a classification and a regression variant, and we will describe both.*

** We will review the most popular machine learning algorithms, explain how they learn from data and how they make predictions, and examine the strengths and weaknesses of each algorithm.*

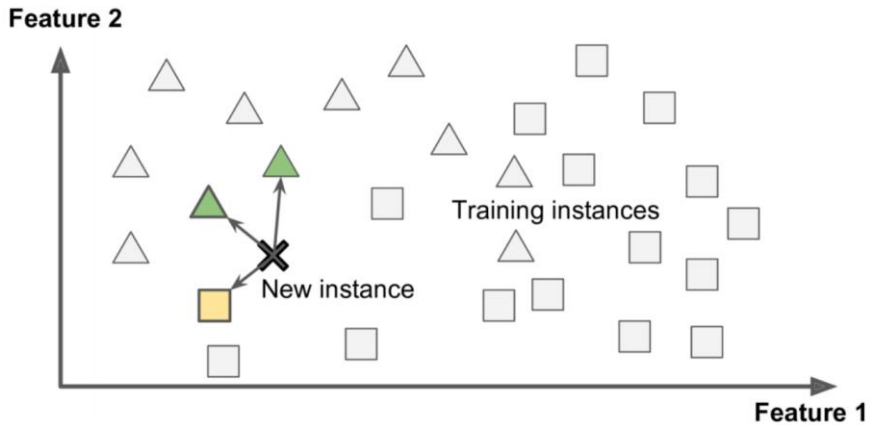
k-Nearest Neighbors

Model-Based vs. Instance-Based Learning

- **Model-Based Learning (Eager Learning)**
  - **Training phase:** Build a model using training data
  - **Prediction phase:** Use the model to make predictions



- **Instance-Based Learning (Lazy Learning)**
  - **Training phase:** Do nothing
  - **Prediction phase:** Compare new instances with training data to make predictions



- **Instance-based learning takes less time in training but more time in predicting, and is advantageous when training data becomes available gradually over time.**

k-Nearest Neighbors

- The *k*-NN algorithm simply stores 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."
- The prediction is an aggregation of the known outputs for the nearest neighbors.
  - Example:

    For *classification*, the prediction is the majority class among the relevant neighbors.

    For *regression*, the prediction is the average of the relevant neighbors' labels.

k-Nearest Neighbors

- Given a (training) dataset $D = \{(\boldsymbol{x}_1, y_1), (\boldsymbol{x}_2, y_2), \dots, (\boldsymbol{x}_n, y_n)\}$ such that $\boldsymbol{x}_i = (x_{i1}, \dots, x_{id}) \in \mathbb{R}^d$ is the *i*-th input vector of *d* features and y_i is the corresponding target label.

- For a query data point $\boldsymbol{x}_{\text{new}}$
  1. Compute distance from $\boldsymbol{x}_{\text{new}}$ to each data point in D (distance metric)
  2. Identify *k* nearest neighbors of $\boldsymbol{x}_{\text{new}}$, $kNN(\boldsymbol{x}_{\text{new}}) = \{(\boldsymbol{x}_{(1)}, y_{(1)}), \dots, (\boldsymbol{x}_{(k)}, y_{(k)})\} \subset D$ (k)
  3. Use labels of the nearest neighbors to predict y_{new} (weighting scheme)
     - e.g) voting or weighted voting for classification, averaging or weighted averaging for regression.

What are parameters?
What are hyperparameters?

k-Nearest Neighbors

- **For Classification,**

  Voting: $\hat{y} = \underset{j}{\operatorname{argmax}} \sum_{(\boldsymbol{x}_{(i)}, y_{(i)}) \in kNN(\boldsymbol{x})} I(y_{(i)} = j)$

  Weighted Voting: $\hat{y} = \underset{j}{\operatorname{argmax}} \sum_{(\boldsymbol{x}_{(i)}, y_{(i)}) \in kNN(\boldsymbol{x})} w(\boldsymbol{x}_{(i)}, \boldsymbol{x}) I(y_{(i)} = j)$

- **For Regression,**

  Averaging: $\hat{y} = \frac{1}{k} \sum_{(\boldsymbol{x}_{(i)}, y_{(i)}) \in kNN(\boldsymbol{x})} y_{(i)}$

  Weighted Averaging: $\hat{y} = \frac{1}{\sum_{(\boldsymbol{x}_{(i)}, y_{(i)}) \in kNN(\boldsymbol{x})} w(\boldsymbol{x}_{(i)}, \boldsymbol{x})} \sum_{(\boldsymbol{x}_{(i)}, y_{(i)}) \in kNN(\boldsymbol{x})} w(\boldsymbol{x}_{(i)}, \boldsymbol{x}) y_{(i)}$

$w(\boldsymbol{x}_{(i)}, \boldsymbol{x})$ is a weight function *(hyperparameter, not learned)*

e.g., inverse of Euclidean distance $\frac{1}{\|\boldsymbol{x}_{(i)} - \boldsymbol{x}\|_2}$

Hyperparameters

- **Distance metric**
  - Euclidean: $\left\|\boldsymbol{x}_{(i)} - \boldsymbol{x}\right\|_2$
  - Manhattan: $\left\|\boldsymbol{x}_{(i)} - \boldsymbol{x}\right\|_1$
  - Minkowski: $\left\|\boldsymbol{x}_{(i)} - \boldsymbol{x}\right\|_p$

  ** Distance Metrics available in scikit-learn*

  https://scikit-learn.org/stable/modules/generated/sklearn.neighbors.DistanceMetric.html

- **The number of nearest neighbors *k***
  - Smaller *k* → capture local structure in data (but also noise)
  - Larger *k* → provide more smoothing, less noise, but may miss local structure

Hyperparameters

- **Weight function**
  - Uniform: $w\left(\boldsymbol{x}_{(i)}, \boldsymbol{x}\right) = 1$
  - Distance Weight (Inverse of Euclidean): $w\left(\boldsymbol{x}_{(i)}, \boldsymbol{x}\right) = \frac{1}{\left\|\boldsymbol{x}_{(i)} - \boldsymbol{x}\right\|_2}$
  - Distance Weight (Inverse of Manhattan): $w\left(\boldsymbol{x}_{(i)}, \boldsymbol{x}\right) = \frac{1}{\left\|\boldsymbol{x}_{(i)} - \boldsymbol{x}\right\|_1}$
  - Distance Weight (Inverse of Minkowski (p)): $w\left(\boldsymbol{x}_{(i)}, \boldsymbol{x}\right) = \frac{1}{\left\|\boldsymbol{x}_{(i)} - \boldsymbol{x}\right\|_p}$

scikit-learn Practice: *KNeighborsClassifier*

https://scikit-learn.org/stable/modules/generated/sklearn.neighbors.KNeighborsClassifier.html

sklearn.neighbors.KNeighborsClassifier

class sklearn.neighbors.**KNeighborsClassifier**(*n_neighbors=5, *, weights='uniform', algorithm='auto', leaf_size=30, p=2, metric='minkowski', metric_params=None, n_jobs=None, **kwargs*) [source]

Classifier implementing the k-nearest neighbors vote.

Read more in the User Guide.

Parameters:

n_neighbors : *int, default=5*

Number of neighbors to use by default for `kneighbors` queries.

weights : *{'uniform', 'distance'} or callable, default='uniform'*

weight function used in prediction. Possible values:

- 'uniform' : uniform weights. All points in each neighborhood are weighted equally.
- 'distance' : weight points by the inverse of their distance. in this case, closer neighbors of a query point will have a greater influence than neighbors which are further away.
- [callable] : a user-defined function which accepts an array of distances, and returns an array of the same shape containing the weights.

algorithm : *{'auto', 'ball_tree', 'kd_tree', 'brute'}, default='auto'*

Algorithm used to compute the nearest neighbors:

- 'ball_tree' will use `BallTree`
- 'kd_tree' will use `KDTree`
- 'brute' will use a brute-force search.
- 'auto' will attempt to decide the most appropriate algorithm based on the values passed to `fit` method.

Note: fitting on sparse input will override the setting of this parameter, using brute force.

scikit-learn Practice: *KNeighborsClassifier*

https://scikit-learn.org/stable/modules/generated/sklearn.neighbors.KNeighborsClassifier.html

p : *int, default=2*
Power parameter for the Minkowski metric. When p = 1, this is equivalent to using manhattan_distance (l1), and euclidean_distance (l2) for p = 2. For arbitrary p, minkowski_distance (l_p) is used.

metric : *str or callable, default='minkowski'*
the distance metric to use for the tree. The default metric is minkowski, and with p=2 is equivalent to the standard Euclidean metric. See the documentation of `DistanceMetric` for a list of available metrics. If metric is "precomputed", X is assumed to be a distance matrix and must be square during fit. X may be a sparse graph, in which case only "nonzero" elements may be considered neighbors.

metric_params : *dict, default=None*
Additional keyword arguments for the metric function.

Methods

`fit`(X, y)	Fit the k-nearest neighbors classifier from the training dataset.
`get_params`([deep])	Get parameters for this estimator.
`kneighbors`([X, n_neighbors, return_distance])	Finds the K-neighbors of a point.
`kneighbors_graph`([X, n_neighbors, mode])	Computes the (weighted) graph of k-Neighbors for points in X
`predict`(X)	Predict the class labels for the provided data.
`predict_proba`(X)	Return probability estimates for the test data X.
`score`(X, y[, sample_weight])	Return the mean accuracy on the given test data and labels.
`set_params`(**params)	Set the parameters of this estimator.

scikit-learn Practice: *KNeighborsClassifier*

- **Example with the *forge* dataset**
  - The dataset consists of 26 data points with two classes (binary classification).

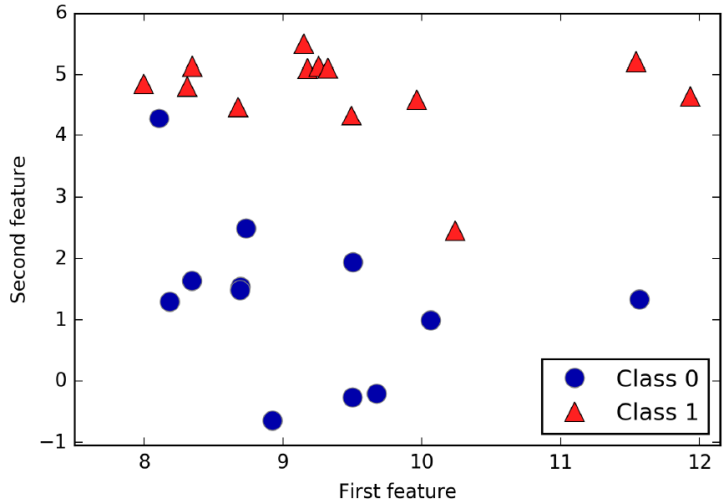
```
import mglearn
import matplotlib.pyplot as plt
```

In[1]:

```
# generate dataset
X, y = mglearn.datasets.make_forge()
# plot dataset
mglearn.discrete_scatter(X[:, 0], X[:, 1], y)
plt.legend(["Class 0", "Class 1"], loc=4)
plt.xlabel("First feature")
plt.ylabel("Second feature")
print("X.shape: {}".format(X.shape))
```

Out[1]:

```
X.shape: (26, 2)
```



	X1	X2	Y
0	9.96347	4.59677	1
1	11.03295	-0.16817	0
2	11.54156	5.21116	1
3	8.69289	1.54322	0
4	8.10623	4.28696	0
5	8.30989	4.80624	1
6	11.93027	4.64866	1
7	9.67285	-0.20283	0
8	8.34810	5.13416	1
9	8.67495	4.47573	1
10	9.17748	5.09283	1
11	10.24029	2.45544	1
12	8.68937	1.48710	0
13	8.92230	-0.63993	0
14	9.49123	4.33225	1
15	9.25694	5.13285	1
16	7.99815	4.85251	1
17	8.18378	1.29564	0
18	8.73371	2.49162	0
19	9.32298	5.09841	1
20	10.06394	0.99078	0
21	9.50049	-0.26430	0
22	8.34469	1.63824	0
23	9.50169	1.93825	0
24	9.15072	5.49832	1
25	11.56396	1.33894	0

scikit-learn Practice: *KNeighborsClassifier*

- **Example (*forge* dataset): *k*=3**

In [2]:
```
import mglearn
X, y = mglearn.datasets.make_forge()
```

In [3]:
```
from sklearn.model_selection import train_test_split
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.25, random_state=0)
```

In [4]:
```
from sklearn.neighbors import KNeighborsClassifier
clf = KNeighborsClassifier(n_neighbors=3)
clf.fit(X_train, y_train)
```

Out[4]:
```
KNeighborsClassifier
KNeighborsClassifier(n_neighbors=3)
```

In [5]:
```
y_test_hat = clf.predict(X_test)
print(y_test)
print(y_test_hat)
```

```
[1 0 1 0 1 1 0]
[1 0 1 0 1 0 0]
```

In [6]:
```
from sklearn.metrics import accuracy_score
y_train_hat = clf.predict(X_train)
print('train accuracy: %.5f'%accuracy_score(y_train, y_train_hat))
y_test_hat = clf.predict(X_test)
print('test accuracy: %.5f'%accuracy_score(y_test, y_test_hat))
```

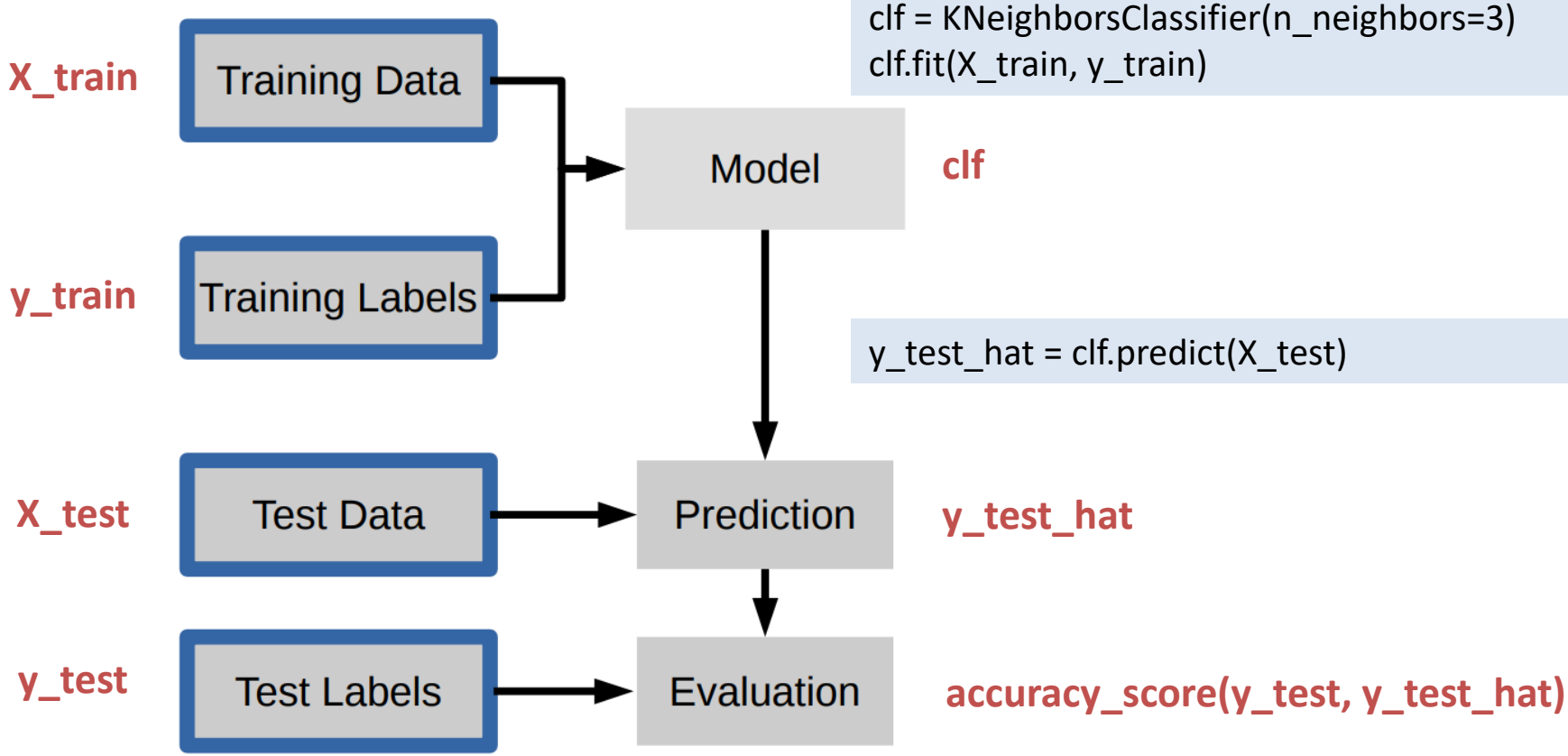
```
train accuracy: 0.94737
test accuracy: 0.85714
```

scikit-learn Practice: *KNeighborsClassifier*

- **Example (*forge* dataset): *k*=3**

```
X, y = mglearn.datasets.make_forge()
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.25, random_state=0)
```

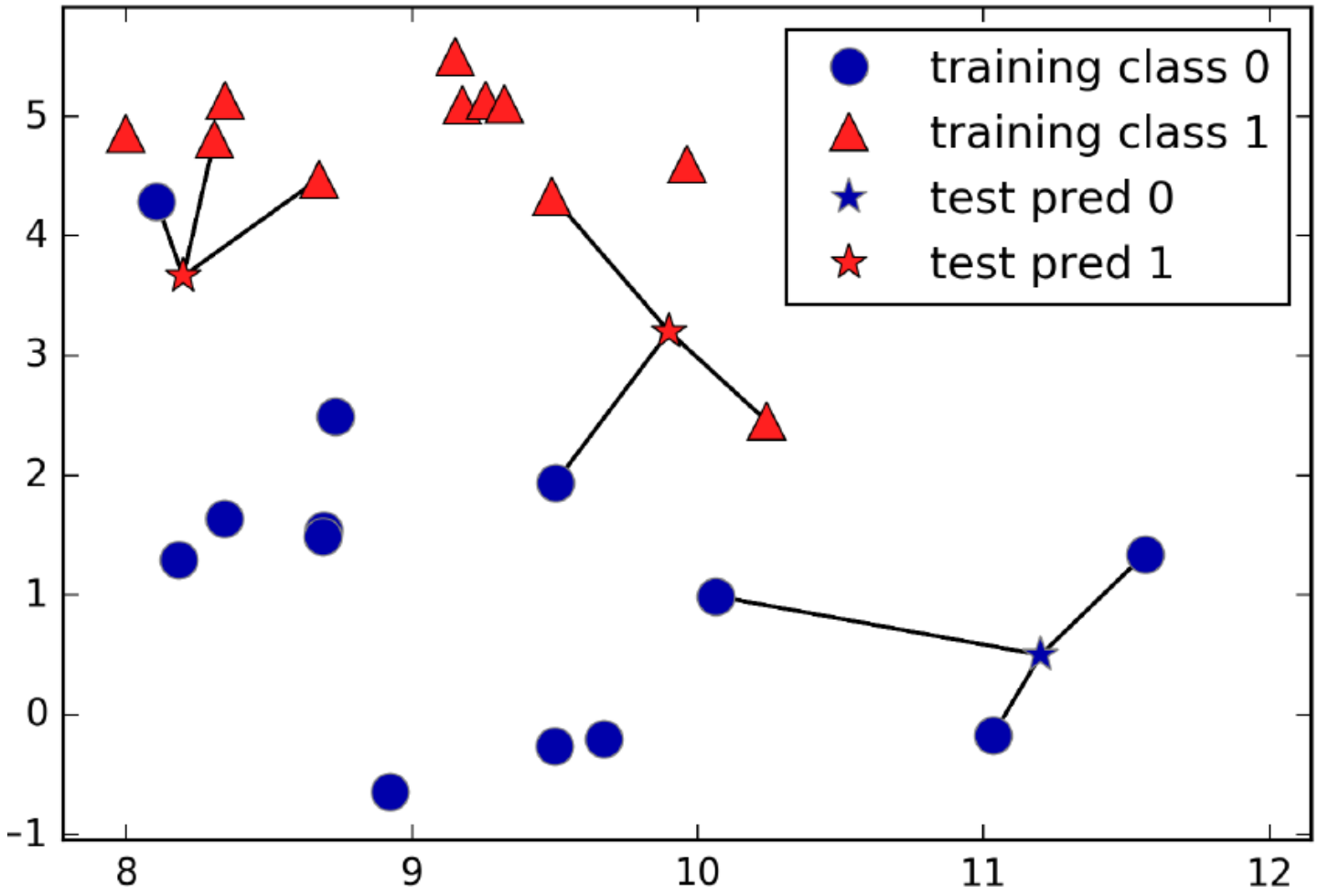
```
clf = KNeighborsClassifier(n_neighbors=3)
clf.fit(X_train, y_train)
```




```
y_test_hat = clf.predict(X_test)
```

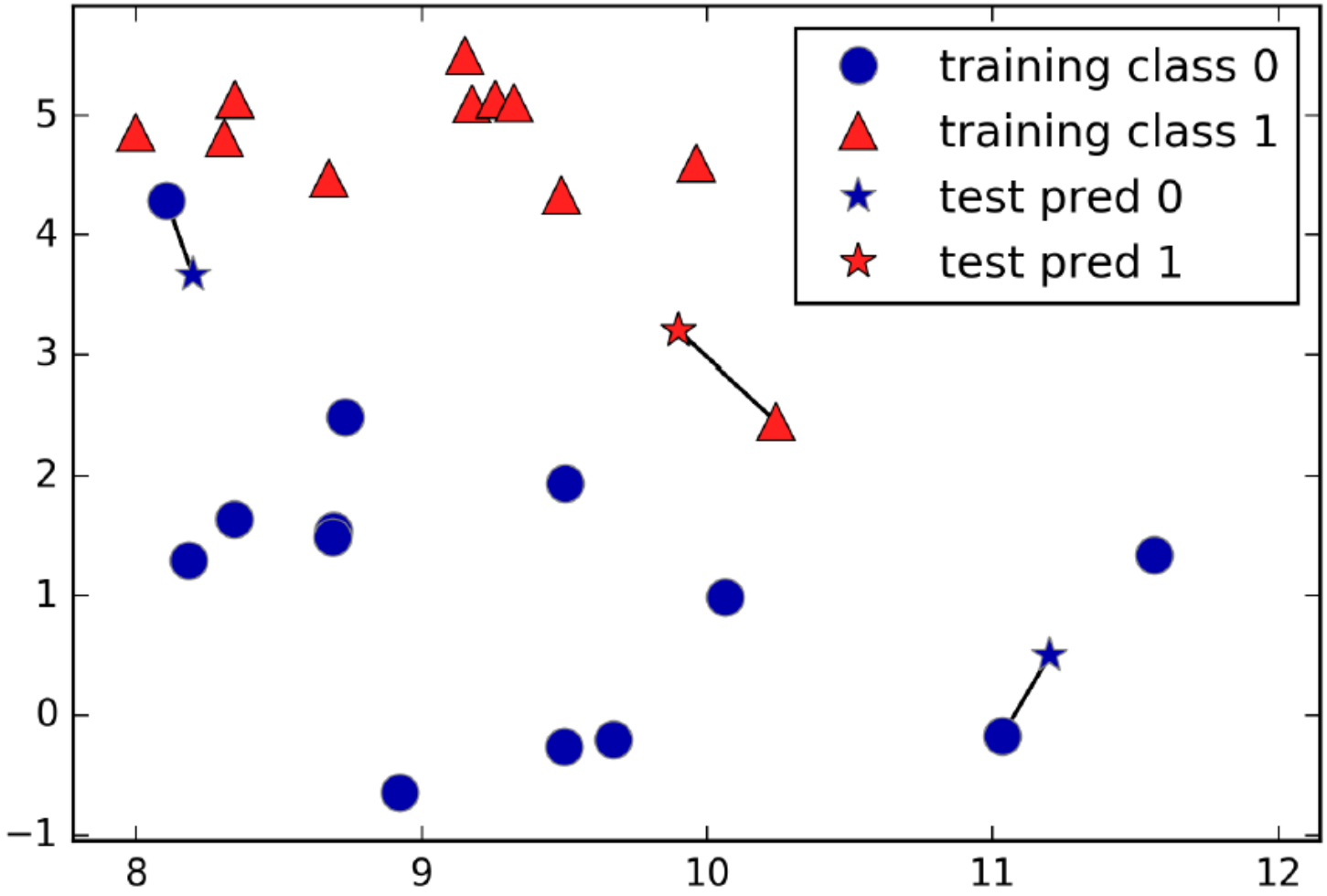
scikit-learn Practice: *KNeighborsClassifier*

- **Example (*forge* dataset): *k*=3**



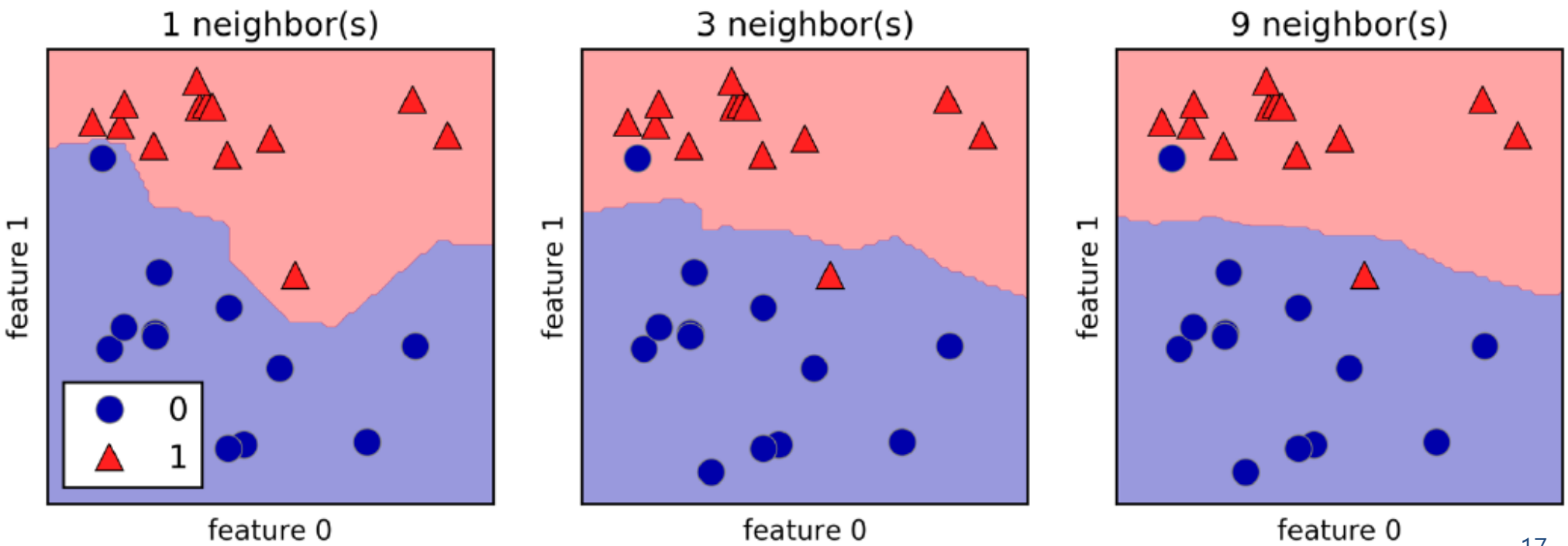
scikit-learn Practice: *KNeighborsClassifier*

- **Example (*forge* dataset): *k*=1**



scikit-learn Practice: *KNeighborsClassifier*

- **The effect of the hyperparameter *k* (n_neighbors)**
  - Decision boundaries created by *k*-NN
    - Using a single neighbor (*k*=1) results in a decision boundary that follows the training data closely. (corresponds to high model complexity)
    - Considering more neighbors leads to a smoother decision boundary. (corresponds to low model complexity)



scikit-learn Practice: *KNeighborsClassifier*

- **Example with the *breast_cancer* dataset**
  - The dataset consists of 569 data points with 30 features
  - Each data point (tumor) is labeled as “benign” (for harmless tumors) or “malignant” (for cancerous tumors) – *binary classification*
  - Of these 569 data points, 212 are labeled as malignant and 357 as benign.
  - The classification task is to learn to predict whether a tumor is malignant based on the measurements of the tissue.

scikit-learn Practice: *KNeighborsClassifier*

- **Example (*breast_cancer* dataset): varying the hyperparameter *k***

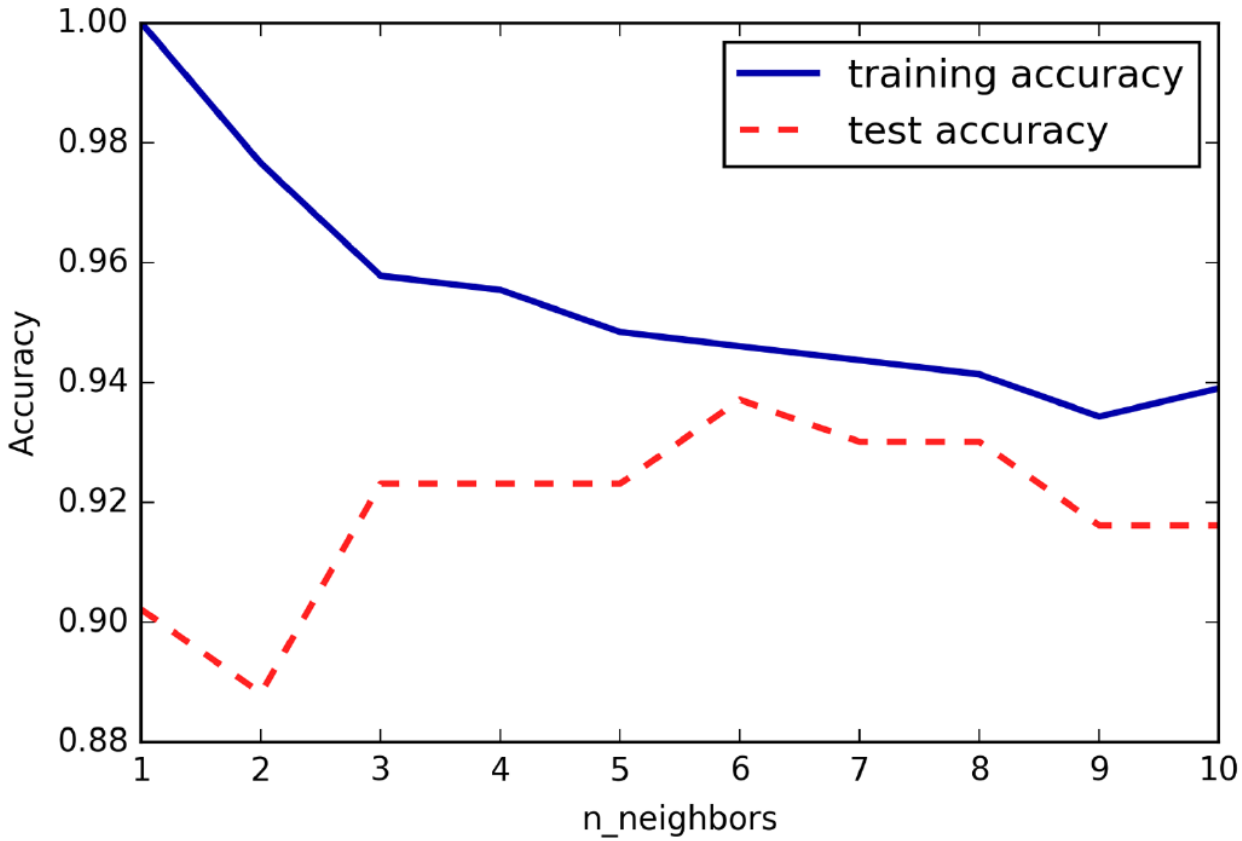
In [7]:
```
from sklearn.datasets import load_breast_cancer
from sklearn.model_selection import train_test_split
from sklearn.neighbors import KNeighborsClassifier
from sklearn.metrics import accuracy_score

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)
```

In [8]:
```
training_accuracy = []
test_accuracy = []

k_settings = range(1, 11) # try n_neighbors from 1 to 10
for k in k_settings:
    # build the model
    clf = KNeighborsClassifier(n_neighbors=k)
    clf.fit(X_train, y_train)

    # accuracy on the training set
    y_train_hat = clf.predict(X_train)
    training_accuracy.append(accuracy_score(y_train, y_train_hat))

    # accuracy on the test set (generalization)
    y_test_hat = clf.predict(X_test)
    test_accuracy.append(accuracy_score(y_test, y_test_hat))
```

	k	training accuracy	test accuracy
0	1	1.00000	0.90210
1	2	0.97653	0.88811
2	3	0.95775	0.92308
3	4	0.95540	0.92308
4	5	0.94836	0.92308
5	6	0.94601	0.93706
6	7	0.94366	0.93007
7	8	0.94131	0.93007
8	9	0.93427	0.91608
9	10	0.93897	0.91608

scikit-learn Practice: *KNeighborsClassifier*

- **Example (*breast_cancer* dataset): varying the hyperparameter *k***
  - *Comparison of training and test accuracy as a function of k (n_neighbors)*
    - Smaller *k* → overfitting
    - Larger *k* → underfitting



scikit-learn Practice: *KNeighborsClassifier*

- **Example (*breast_cancer* dataset): varying the distance metric**

In [10]:
```
from sklearn.datasets import load_breast_cancer
from sklearn.model_selection import train_test_split
from sklearn.neighbors import KNeighborsClassifier
from sklearn.metrics import accuracy_score

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)
```

In [11]:
```
training_accuracy = []
test_accuracy = []

p_settings = range(1, 6) # try minkowski p from 1 to 5
for p in p_settings:
    # build the model
    clf = KNeighborsClassifier(n_neighbors=5, metric='minkowski', p=p)
    clf.fit(X_train, y_train)

    # accuracy on the training set
    y_train_hat = clf.predict(X_train)
    training_accuracy.append(accuracy_score(y_train, y_train_hat))

    # accuracy on the test set (generalization)
    y_test_hat = clf.predict(X_test)
    test_accuracy.append(accuracy_score(y_test, y_test_hat))
```

Minkowski Distance (p)

$$\left\|\boldsymbol{x}_{(i)} - \boldsymbol{x}\right\|_p$$

	p	training accuracy	test accuracy
0	1	0.96479	0.93706
1	2	0.94836	0.92308
2	3	0.94366	0.93007
3	4	0.94366	0.92308
4	5	0.94366	0.92308

scikit-learn Practice: *KNeighborsRegressor*

https://scikit-learn.org/stable/modules/generated/sklearn.neighbors.KNeighborsRegressor.html

sklearn.neighbors.KNeighborsRegressor¶

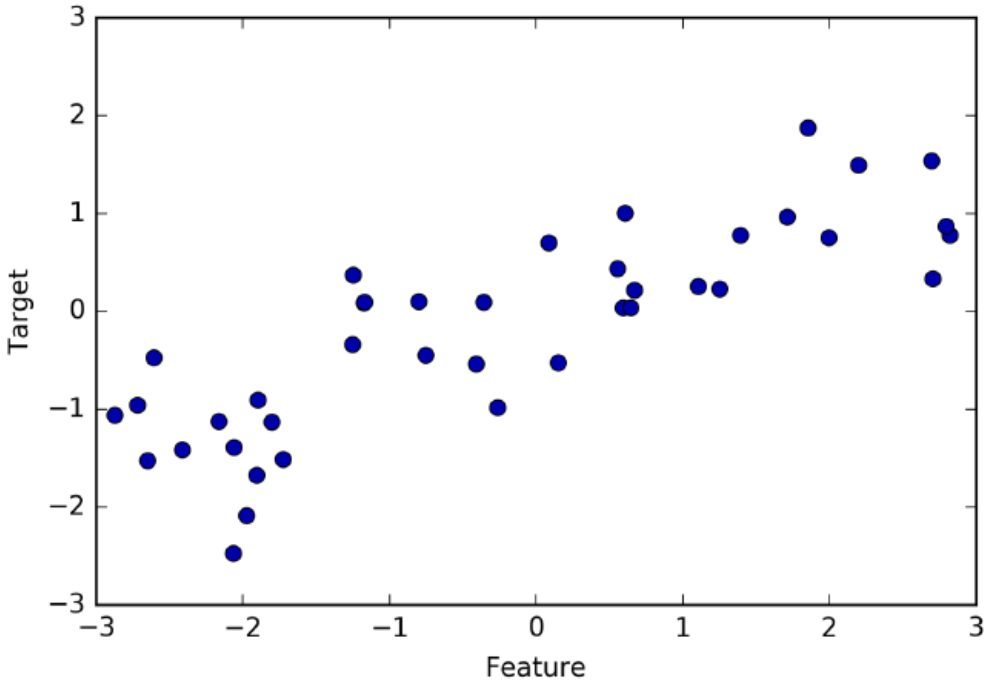
class `sklearn.neighbors.`**`KNeighborsRegressor`**(*n_neighbors=5, *, weights='uniform', algorithm='auto', leaf_size=30, p=2, metric='minkowski', metric_params=None, n_jobs=None, **kwargs*) [source]

Regression based on k-nearest neighbors.

The target is predicted by local interpolation of the targets associated of the nearest neighbors in the training set.

scikit-learn Practice: *KNeighborsRegressor*

- **Example with the *wave* dataset**
  - The wave dataset is a synthetic dataset that has a single feature and a continuous target.



scikit-learn Practice: *KNeighborsRegressor*

- **Example (*wave* dataset): *k*=3**

In [13]:
```
import mglearn
X, y = mglearn.datasets.make_wave(n_samples=40)
```

In [14]:
```
from sklearn.model_selection import train_test_split
X_train, X_test, y_train, y_test = train_test_split(X, y, random_state=0)
```

In [15]:
```
from sklearn.neighbors import KNeighborsRegressor
reg = KNeighborsRegressor(n_neighbors=3)
reg.fit(X_train, y_train)
```

Out[15]:
```
KNeighborsRegressor
KNeighborsRegressor(n_neighbors=3)
```

In [16]:
```
y_test_hat = reg.predict(X_test)
print(y_test)
print(y_test_hat)
```

```
[ 0.37299  0.21778  0.96695 -1.38774 -1.0598  -0.90497  0.43656  0.77896
 -0.54115 -0.95652]
[-0.05397  0.35686  1.13672 -1.89416 -1.13881 -1.63113  0.35686  0.91241
 -0.4468  -1.13881]
```

In [17]:
```
from sklearn.metrics import mean_absolute_error, mean_squared_error, r2_score
print('MAE: %.5f'%mean_absolute_error(y_test,y_test_hat))
print('RMSE: %.5f'%mean_squared_error(y_test,y_test_hat)**0.5)
print('R_square: %.5f'%r2_score(y_test,y_test_hat))
```

```
MAE: 0.25372
RMSE: 0.32966
R_square: 0.83442
```

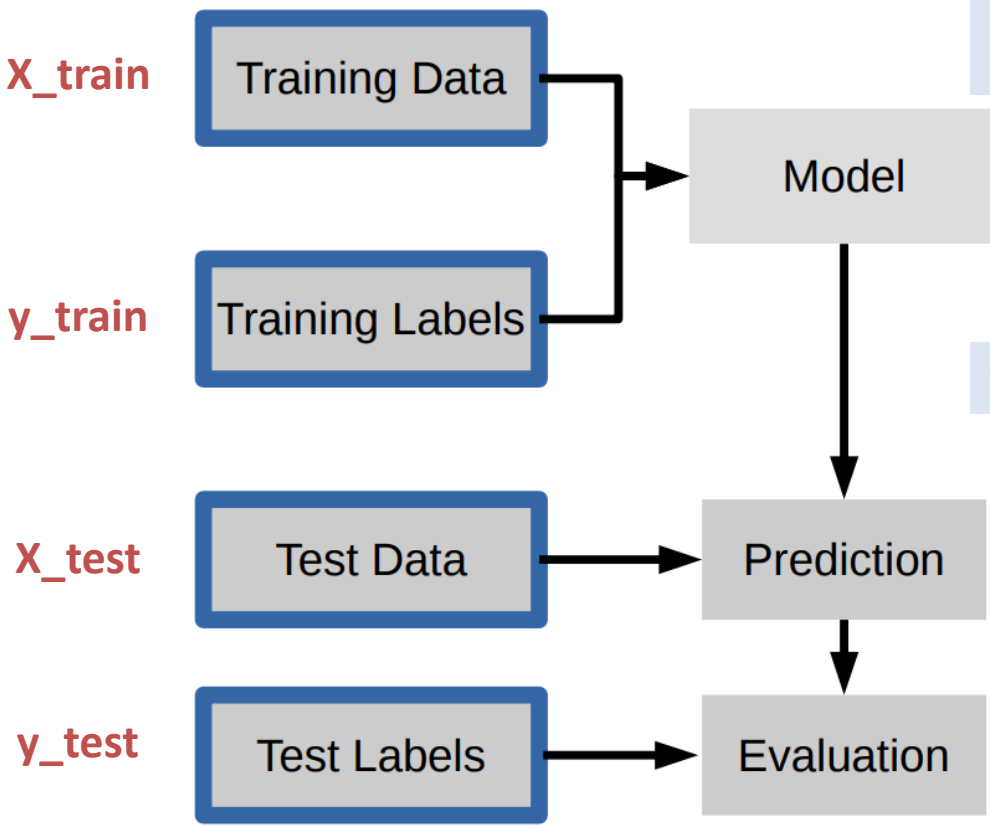
	X	Y
0	-0.75276	-0.44822
1	2.70429	0.33123
2	1.39196	0.77932
3	0.59195	0.03498
4	-2.06389	-1.38774
5	-2.06403	-2.47196
6	-2.65150	-1.52731
7	2.19706	1.49417
8	0.60669	1.00032
9	1.24844	0.22956
10	-2.87649	-1.05980
11	2.81946	0.77896
12	1.99466	0.75419
13	-1.72597	-1.51370
14	-1.90905	-1.67303
15	-1.89957	-0.90497
16	-1.17455	0.08449
17	0.14854	-0.52735
18	-0.40833	-0.54115
19	-1.25263	-0.34091
	⋮	⋮

scikit-learn Practice: *KNeighborsRegressor*

- **Example (*wave* dataset): *k*=3**

```
X, y = mglearn.datasets.make_wave(n_samples=40)
X_train, X_test, y_train, y_test = train_test_split(X, y, random_state=0)
```

```
reg = KNeighborsRegressor(n_neighbors=3)
reg.fit(X_train, y_train)
```



reg

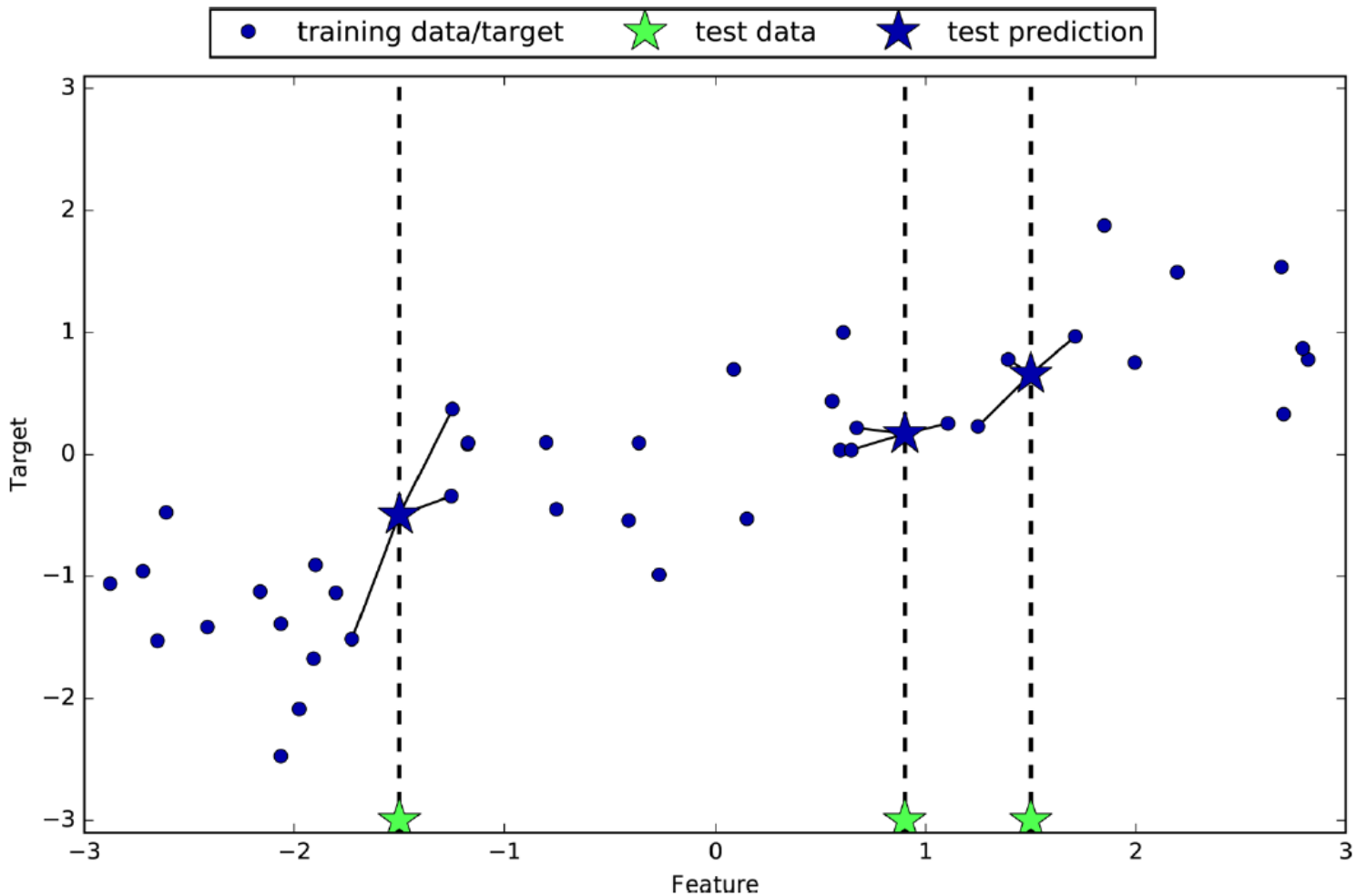
```
y_test_hat = reg.predict(X_test)
```

y_test_hat

```
mean_absolute_error(y_test,y_test_hat)
mean_squared_error(y_test,y_test_hat)**0.5
r2_score(y_test,y_test_hat)
```

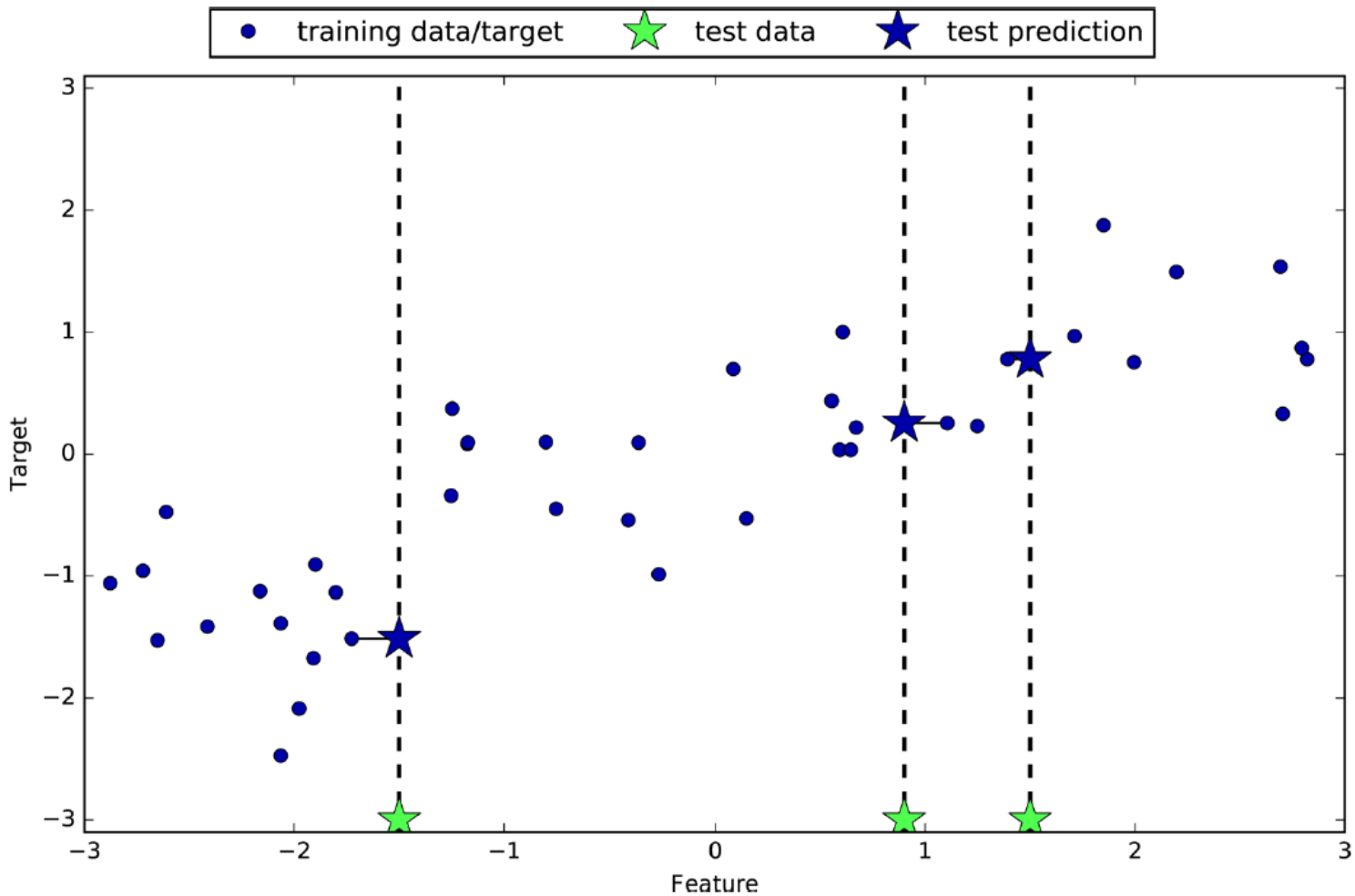
scikit-learn Practice: *KNeighborsRegressor*

- **Example (*wave* dataset): *k*=3**



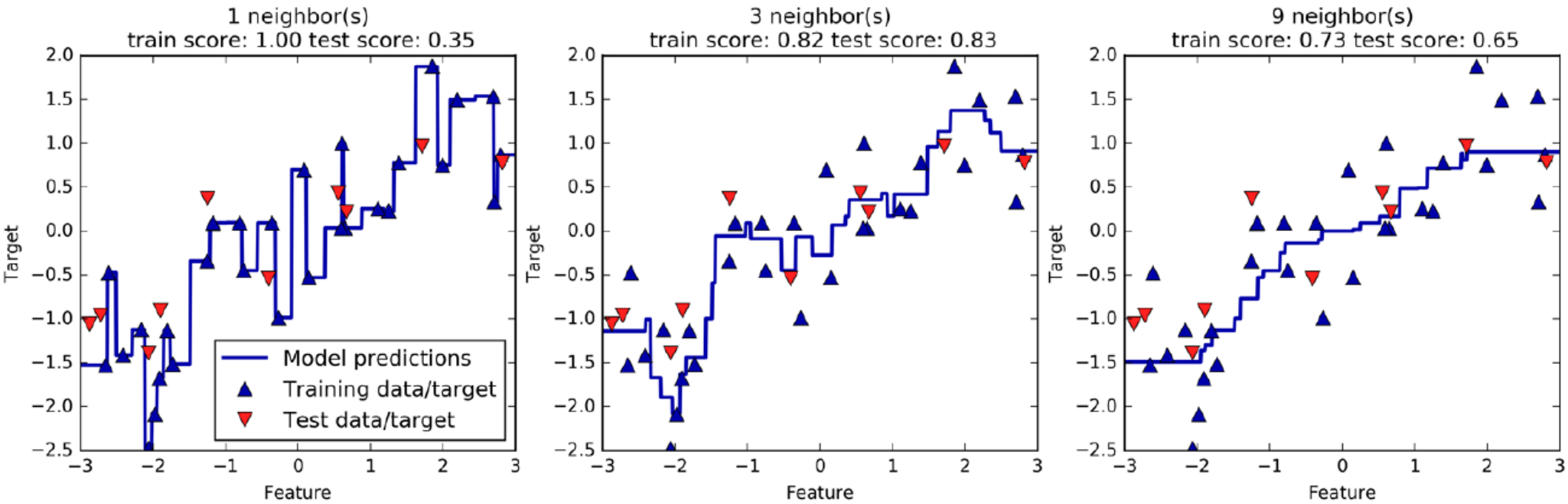
scikit-learn Practice: *KNeighborsRegressor*

- **Example (*wave* dataset): *k*=1**



scikit-learn Practice: *KNeighborsRegressor*

- **The effect of the hyperparameter *k* (n_neighbors)**
  - *Comparing predictions made by k*-NN
    - By using a single neighbor (*k*=1), each point in the training set has an obvious influence on the predictions. (high model complexity)
    - Considering more neighbors leads to smoother predictions. (low model complexity)



Discussion

- **The main hyperparameters to the *k*-NN algorithm**
  - *n_neighbor (*the number of neighbors k)
  - *metric* (distance metric)
  - *weights* (weighting scheme)

  ** Typically chosen to have the highest performance in validation data*

  * It's important to preprocess your data (including *data scaling* and *one-hot encoding*)

- **Strengths**
  - The algorithm is very easy to understand
  - The algorithm often gives reasonable performance without a lot of adjustments (good baseline method to try)

- **Weaknesses**
  - Prediction can be slow when your training set is very large (either in number of features or in number of data points)
  - The algorithm often does not perform well on datasets with many features (hundreds or more)

Data Scaling

- **Feature Scaling for Distance Metric**
  - Certain features are more important than others in many applications.
  - It is often sensible to scale the features differently.

Original Dataset

	x_1	x_2
data point a	3	200
data point b	10	100
data point c	11	200

$$\text{dist}(\boldsymbol{a}, \boldsymbol{b}) = \sqrt{10049} \simeq 100.2$$
$$\text{dist}(\boldsymbol{a}, \boldsymbol{c}) = \sqrt{64} = 8$$

$$\text{dist}(\boldsymbol{a}, \boldsymbol{b}) > \text{dist}(\boldsymbol{a}, \boldsymbol{c})$$

Same Dataset (but different feature scales)

	x_1	$x_2/100$
data point a	3	2
data point b	10	1
data point c	11	2

$$\text{dist}(\boldsymbol{a}, \boldsymbol{b}) = \sqrt{50} \simeq 7.1$$
$$\text{dist}(\boldsymbol{a}, \boldsymbol{c}) = \sqrt{64} = 8$$

$$\text{dist}(\boldsymbol{a}, \boldsymbol{b}) < \text{dist}(\boldsymbol{a}, \boldsymbol{c})$$

Data Scaling

- **Feature Scaling: Puts all numeric features on same scale**
  - It must be used when features with the largest scales would dominate and skew results
- **Scaling functions**
  - **Standard Scaling:** Transform to have a mean of 0 and a standard deviation of 1 by subtracting mean and dividing by standard deviation

```
class sklearn.preprocessing.StandardScaler(*, copy=True, with_mean=True, with_std=True)   [source]
```

  - **MinMax Scaling:** Scale to 0-1 by subtracting minimum and dividing by the range

```
class sklearn.preprocessing.MinMaxScaler(feature_range=(0, 1), *, copy=True, clip=False)   [source]
```

  - **Robust Scaling**: Transform by subtracting the median and then dividing by the interquartile range

```
class sklearn.preprocessing.RobustScaler(*, with_centering=True, with_scaling=True, quantile_range=(25.0, 75.0), copy=True,
unit_variance=False)   [source]
```

Data Scaling

- **Example (*forge* dataset) with *StandardScaler***

In [19]:
```
import mglearn
X, y = mglearn.datasets.make_forge()
```

In [20]:
```
from sklearn.model_selection import train_test_split
X_train, X_test, y_train, y_test = train_test_split(X, y, random_state=0)
```

In [21]:
```
from sklearn.preprocessing import StandardScaler
scaler = StandardScaler()
scaler.fit(X_train)
X_train_scaled = scaler.transform(X_train)
X_test_scaled = scaler.transform(X_test)
```

In [22]:
```
from sklearn.neighbors import KNeighborsClassifier
clf = KNeighborsClassifier(n_neighbors=3)
clf.fit(X_train_scaled, y_train)
```

Out[22]:
```
▾ KNeighborsClassifier
KNeighborsClassifier(n_neighbors=3)
```

	X1_train	X2_train	X1_train_scaled	X2_train_scaled
0	8.92230	-0.63993	-0.43383	-1.66209
1	8.73371	2.49162	-0.61218	-0.21838
2	9.32298	5.09841	-0.05489	0.98339
3	7.99815	4.85251	-1.30782	0.87003
4	11.03295	-0.16817	1.56227	-1.44460
5	9.17748	5.09283	-0.19250	0.98082
6	11.56396	1.33894	2.06445	-0.74979
7	9.15072	5.49832	-0.21780	1.16776
8	8.34810	5.13416	-0.97686	0.99988
9	11.93027	4.64866	2.41088	0.77605
	⋮		⋮	

In [23]:
```
from sklearn.metrics import accuracy_score
y_train_hat = clf.predict(X_train_scaled)
print('train accuracy: %.5f'%accuracy_score(y_train, y_train_hat))
y_test_hat = clf.predict(X_test_scaled)
print('test accuracy: %.5f'%accuracy_score(y_test, y_test_hat))
```

```
train accuracy: 0.94737
test accuracy: 0.85714
```

Results without Scaling

```
train accuracy: 0.94737
test accuracy: 0.85714
```

Data Scaling

- **Example (*forge* dataset) with *MinMaxScaler***

In [25]:
```
import mglearn
X, y = mglearn.datasets.make_forge()
```

In [26]:
```
from sklearn.model_selection import train_test_split
X_train, X_test, y_train, y_test = train_test_split(X, y, random_state=0)
```

In [27]:
```
from sklearn.preprocessing import MinMaxScaler
scaler = MinMaxScaler()
scaler.fit(X_train)
X_train_scaled = scaler.transform(X_train)
X_test_scaled = scaler.transform(X_test)
```

In [28]:
```
from sklearn.neighbors import KNeighborsClassifier
clf = KNeighborsClassifier(n_neighbors=3)
clf.fit(X_train_scaled, y_train)
```

Out[28]:
```
▾ KNeighborsClassifier
KNeighborsClassifier(n_neighbors=3)
```

In [29]:
```
from sklearn.metrics import accuracy_score
y_train_hat = clf.predict(X_train_scaled)
print('train accuracy: %.5f'%accuracy_score(y_train, y_train_hat))
y_test_hat = clf.predict(X_test_scaled)
print('test accuracy: %.5f'%accuracy_score(y_test, y_test_hat))
```

```
train accuracy: 0.94737
test accuracy: 0.85714
```

	X1_train	X2_train	X1_train_scaled	X2_train_scaled
0	8.92230	-0.63993	0.23502	0.00000
1	8.73371	2.49162	0.18706	0.51017
2	9.32298	5.09841	0.33693	0.93485
3	7.99815	4.85251	0.00000	0.89479
4	11.03295	-0.16817	0.77180	0.07686
5	9.17748	5.09283	0.29992	0.93394
6	11.56396	1.33894	0.90684	0.32238
7	9.15072	5.49832	0.29312	1.00000
8	8.34810	5.13416	0.08900	0.94067
9	11.93027	4.64866	1.00000	0.86158
	⋮		⋮	

Results without Scaling

```
train accuracy: 0.94737
test accuracy: 0.85714
```

One-Hot Encoding

- **Categorical features must be transformed into numeric features**
- **One-Hot Encoding:** replace a categorical feature with multiple new features that can have the values 0 and 1 (one new feature per category)

class `sklearn.preprocessing.`**`OneHotEncoder`**(*, *categories='auto', drop=None, sparse=True, dtype=<class 'numpy.float64'>, handle_unknown='error'*) [source]

- **Example: Color Categories {Red, Green, Blue}**

ID	Color
1	Red
2	Green
3	Green
4	Blue
5	Green
6	Red
7	Red
8	Red
9	Green
10	Blue

One-hot encoding

ID	Color_Red	Color_Green	Color_Blue
1	1	0	0
2	0	1	0
3	0	1	0
4	0	0	1
5	0	1	0
6	1	0	0
7	1	0	0
8	1	0	0
9	0	1	0
10	0	0	1