# CHAP II

# BOOK: introduction to machine learning with python

# Supervised learning

---

supervised learning is used whenever we want to predict a certain
outcome from a given input, and we have examples of input/output pairs.

## Classification vs regression

**Classification**: predict a class label

-binary:distinguishing between exactly 2 cases

-multiclass:classification  between more than 2 classes

Example:-yes/no questions

-spam/non spam

**Regression:** predict a continuous number or a floating number in programming term (or real number in math)

Example: -predict a person’s annual income from education ,age,where they live (the predicted value is an amount and can have any number in a given range)

-predict the yield of a corn farm given attributes such as yields, weather, and number of employees working on the farm

It can also be an arbitrary number

## Generalization, overfitting, undefitting

- if a model is able to make accurate predictions on unseen data, we say it is able to generalize from training set to the test set

**overfitting:** building the model that is too complex for the amount of data owned.

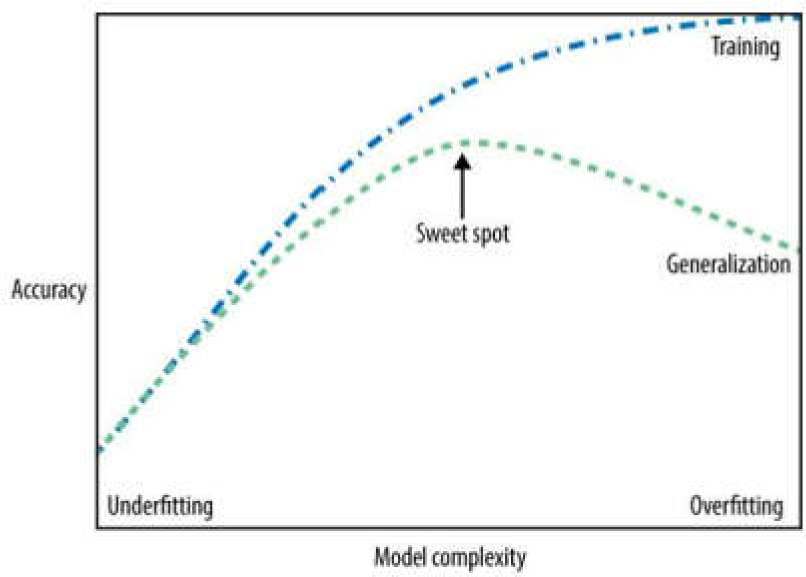
Occurs when you fit a model too closely to the training set but is not able to generalize to new data choosing too simple model is called **underfitting**

**-example**: "everybody who owns a house buys a boat"

- Does not capture all the aspects of and variability in the data and does bad on the training set

- The more complex the model, the better will be able to predict on the training data
- However, if the model becomes too complex, it fails to generalize well to new data
- The model needed is the one in between the two

The trade off between overfitting is illustrated below




## Model complexity vs dataset size

- The larger variety of data points your dataset contains, the more complex a model you can use without overfitting.
- Larger datasets allow building more complex models
- However, duplicating the same data point or collecting very similar data will not help

- Having more data and building more complex models can often work wonders for supervised learning tasks
- Never underestimate the power of more data

# Supervised learning algorithm

## k-Nearest Neighbors

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."

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.

### K-Nearest with scikit-learn

Splitting the data into training and tests ets

**In[12]:**
```
from sklearn.model_selection import train_test_split
X, y = mglearn.datasets.make_forge()
X_train, X_test, y_train, y_test = train_test_split(X, y, random_state=0)
```

import and instantiate the class and set parameters:the number of neighbors=3

**In[13]:**
```
from sklearn.neighbors import KNeighborsClassifier
clf = KNeighborsClassifier(n_neighbors=3)
```

fit the classifier using the training set. For KNeighborsClassifier this means storing the dataset, so we can compute neighbors during prediction:
**In[14]:**

```
clf.fit(X_train, y_train)
```

Make prediction by calling the predict method. For each data point in the test set, this computes its nearest neighbors in the training set and finds the most common class among these:
**In[15]:**

```
print("Test set predictions: {}".format(clf.predict(X_test)))
```

**Out[15]:**

Test set predictions: [1 0 1 0 1 0 0]

To evaluate how well our model generalizes, we can call the score method with the test data together with the test labels:

**In[16]:**

```
print("Test set accuracy: {:.2f}".format(clf.score(X_test, y_test)))
```

**Out[16]:**

Test set accuracy: 0.86

We see that our model is about 86% accurate, meaning the model predicted the class correctly for 86% of the samples in the test dataset.

## Analyzing KNeighborsClassifer

***decision boundary*****,**

**In[17]:**

```
fig, axes = plt.subplots(1, 3, figsize=(10, 3))
for n_neighbors, ax in zip([1, 3, 9], axes):
# the fit method returns the object self, so we can instantiate
# and fit in one line
clf = KNeighborsClassifier(n_neighbors=n_neighbors).fit(X, y)
mglearn.plots.plot_2d_separator(clf, X, fill=True, eps=0.5, ax=ax, alpha=.4)
mglearn.discrete_scatter(X[:, 0], X[:, 1], y, ax=ax)
ax.set_title("{} neighbor(s)".format(n_neighbors))
ax.set_xlabel("feature 0")
ax.set_ylabel("feature 1")
axes[0].legend(loc=3)
```

evaluate the model complexity vs generalization

we evaluate training and test set performance with different numbers of neighbors. The results are shown in Figure 2-7:

**In[18]:**

```
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)
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))
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()
```

# kneighbors regression

The *k*-nearest neighbors algorithm for regression is implemented in the KNeighbors Regressor class in scikit-learn. It's used similarly to KNeighborsClassifier:

**In[21]:**

```
from sklearn.neighbors import KNeighborsRegressor
X, y = mglearn.datasets.make_wave(n_samples=40)
# split the wave dataset into a training and a test set
X_train, X_test, y_train, y_test = train_test_split(X, y, random_state=0)
# instantiate the model and set the number of neighbors to consider to 3
reg = KNeighborsRegressor(n_neighbors=3)
# fit the model using the training data and training targets
reg.fit(X_train, y_train)
```

Now we can make predictions on the test set:

**In[22]:**

```
print("Test set predictions:\n{}".format(reg.predict(X_test)))
```

**Out[22]:**

```
Test set predictions:
[-0.054 0.357 1.137 -1.894 -1.139 -1.631 0.357 0.912 -0.447 -1.139]
```

**evaluate the model**

**In[23]:**

```
print("Test set R^2: {:.2f}".format(reg.score(X_test, y_test)))
```

**Out[23]:**

```
Test set R^2: 0.83
```

Here, the score is 0.83, which indicates a relatively good model fit.

### Analyzing KNeighborsRegressor

creating a test dataset consisting of
many points on the line:

**In[24]:**

```
fig, axes = plt.subplots(1, 3, figsize=(15, 4))
# create 1,000 data points, evenly spaced between -3 and 3
line = np.linspace(-3, 3, 1000).reshape(-1, 1)
for n_neighbors, ax in zip([1, 3, 9], axes):
# make predictions using 1, 3, or 9 neighbors
reg = KNeighborsRegressor(n_neighbors=n_neighbors)
reg.fit(X_train, y_train)
ax.plot(line, reg.predict(line))
ax.plot(X_train, y_train, '^', c=mglearn.cm2(0), markersize=8)
ax.plot(X_test, y_test, 'v', c=mglearn.cm2(1), markersize=8)
ax.set_title(
"{} neighbor(s)\n train score: {:.2f} test score: {:.2f}".format(
n_neighbors, reg.score(X_train, y_train),
reg.score(X_test, y_test)))
ax.set_xlabel("Feature")
ax.set_ylabel("Target")
axes[0].legend(["Model predictions", "Training data/target",
"Test data/target"], loc="best")
```

As we can see from the plot, using only a single neighbor, each point in the training set has an obvious influence on the predictions, and the predicted values go through all of the data points. This leads to a very unsteady prediction. Considering more neighbors leads to smoother predictions, but these do not fit the training data as well.

## Strengths, weaknesses, and parameters

### Parameters:

- The number of neighbors
- The distance between data point

### Streingth:

- Easy to understand
- Give reasonable performance without a lot of adjustments

**Weaknesses**

- slow prediction
- inability to handle many features

## Other algorithms

- Linear models for regression
- Linear regression
- Ridge regression
- Lasso
- Linear models for classification
- Logistic regression
- Linear support vector machines
- Linear models for multiclass classification
- Naïve bayes classifiers
- Decision trees
- Random forest

## Neural Networks (Deep Learning)

**Not understood**

## Reference

Introduction to machine learning with python A GUIDE FOR DATA SCIENTISTS: Andreas C. Müller & Sarah Guido