Chapter 3:
Scikit-Learn: Validasi dan Pemilihan Model, beserta Ukuran Kinerja

October 3, 2020

Pada *Chapter* 2 kita telah memulai menggunakan *library Scikit-Learn* saat membahas regresi linier dan regesi logistik. Untuk lebih memudahkan memahami cara menggunakan library *Scikit-Learn* pada *machine learning*, pada bagian ini akan dijelaskan secara singkat mengenai *library* tersebut. Dalam implementasi algoritma-algoritma *machine learning* dengan library *Scikit-Learn*, maka akan kita lihat langkah-langkah yang serupa tidak tergantung pada jenis algoritma yang kita gunakan.

Selanjutnya, melakukan validasi dan pemilihan model untuk *machine learning* sangat diperlukan untuk mendapatkan kinerja yang terbaik. Oleh sebab itu, akan dibahas cara validasi dan pemilihan model untuk data dengan menggunakan `Scikit-Learn`. Untuk menjelaskan kinerja dari *supervised learning*, sistem klasifikasi yang sederhana akan digunakan. Sedangkan, metode klasifikasi secara terperinci akan dibahas pada *chapter-chapter* selanjutnya.

1 Pengenalan *Scikit-Learn*

Ada beberapa *library* pada Python yang menyediakan implementasi algoritma *machine learning*. Salah satu yang paling baik adalah `Scikit-Learn`, yang merupakan *library open source* untuk *supervised* dan *unsupervised learning*. *Library* ini juga menyediakan tool untuk *model fitting*, *data preprocessing*, *model selection and evaluation* beserta utiliti-utiliti lainnya. Pada bagian ini akan dijelaskan secara singkat bagaimana menggunakan `Scikit-Learn` *library*. *Tutorial*, *User Guide* dan dokumentasi *online* yang sangat detail tentang `Scikit-Learn` bisa dilihat pada link berikut Scikit-Learn.

1.1 Representasi data pada *Scikit-Learn*

Machine learning dapat dipandang sebagai metode untuk membangun model berdasarkan data, sehingga diperlukan cara bagaimana data direpresentasikan sehingga lebih mudah diinterpretasikan oleh komputer. Cara terbaik untuk melihat data pada `Scikit-Learn` adalah dalam bentuk tabel.

- **Data sebagai tabel**

Tabel paling dasar adalah tabel dengan ukuran dua dimensi, dimana baris menyatakan sebuah **sampel** dan kolom menyatakan ***feature***. Contoh sederhana, seperti yang telah dijelaskan pada *Chapter* 2 adalah dataset bunga Iris. Pada bagian ini dataset tersebut akan didownload dalam bentuk Pandas `DataFrame` menggunakan libary `Seaborn` (lihat Pandas dan Seaborn).

[12]:
```
import seaborn as sns
iris = sns.load_dataset('iris')
iris.head()
```

[12]:

	sepal_length	sepal_width	petal_length	petal_width	species
0	5.1	3.5	1.4	0.2	setosa
1	4.9	3.0	1.4	0.2	setosa
2	4.7	3.2	1.3	0.2	setosa
3	4.6	3.1	1.5	0.2	setosa
4	5.0	3.6	1.4	0.2	setosa

Terlihat pada tabel di atas setiap baris merupakan satu observasi bunga (sampel bunga) dengan ukuran panjang/lebar sepal dan petal masing-masing, termasuk kolom terakhir menyatakan jenisnya. Jumlah baris ($n_{samples}$) menyatakan jumlah total sampel. Sedangkan setiap kolom menyatakan *feature-feature* bunga Iris, dengan jumlah kolom ($n_{features}$) menyatakan jumlah *feature*.

- **Matrix *features***

Tabel di atas menjelaskan bahwa informasi dapat dilihat sebagai larik (*array*) numerik dua dimensi atau matriks, yang disebut dengan matriks *feature*. Dengan konvensi, matriks *feature* sering disimbolkan dengan huruf X. Matriks *feature* sering diasumsikan sebagai matriks dua dimensi dengan ukuran $n_{samples} \times n_{features}$ yang biasanya disimpan dalam bentuk `NumPy` array maupun `Pandas DataFrame`. Tetapi, beberapa model `Scikit-Learn` ada juga yang menggunakan bentuk *SciPy sparse matrix*. Baris dari matriks *feature* merupakan sebuah objek individu pada dataset, contohnya bunga, seseorang, dokumen, gambar, file suara, video, dan apapun yang dapat dideskripsikan dengan satu set pengukuruan kuantitatif. Harga-harga *feature* secara umum berbentuk `real`, tetapi juga tidak jarang dalam bentuk `Boolean` atau harga-harga diskrit pada kasus-kasus tertentu.

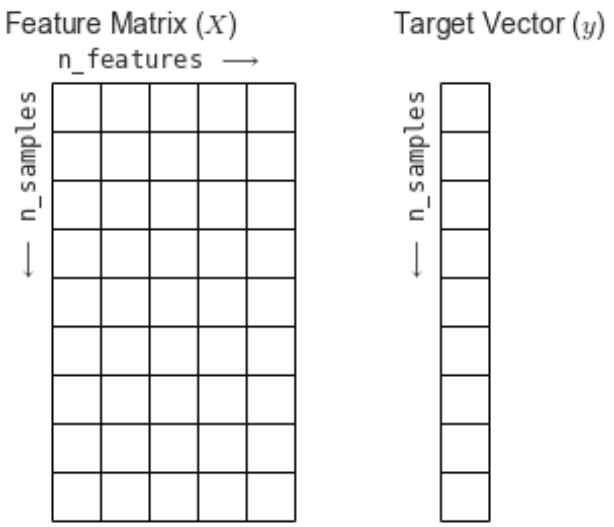
- **Array/Vektor target**

Selain menggunakan matriks *feature* X, pada *supervised learning* sering juga kita bekerja dengan array *label* atau array target, dan biasanya direpresentasikan secara konvensi dengan simbo y. Array target biasanya satu dimensi, dengan panjang $n_samples$ yang dapat berbentuk array `NumPy` atau `Pandas series`. Array target bisa bernilai numerik kontinyu, atau jenis *class* atau label diskrit. Beberapa estimator `Scikit-Learn` dapat menghandel harga-harga target jamak dua dimensi ($n_{samples} \times n_{targets}$). Tapi pada catatan ini hanya akan dibicarakan array target berdimensi satu ($n_{targets} = 1$) atau disebut juga vektor target. Pada tabel di atas, target adalah kolom terakhir sebelah kanan yang menyatakan tipe-tipe bunga Iris. Bagaimana membedakan kolom target dan kolom-kolom *features*? Kolom target merupakan kuantitas yang ingin kita prediksi dari data, atau secara statistik disebut dengan variabel tak bebas (*dependent variable*). Ilustrasi untuk matriks *feature* (*feature matrix*) dan vektor target (*target vector*) bisa dilihat pada Gambar 3.1

Agar data bunga Iris di atas (yang masih dalam bentuk gabungan matriks *features* dan array target) bisa digunakan pada `Scikit-Learn`, maka keduanya harus dipisahkan terlebih dahulu. Misalkan dalam kasus ini dengan menggunakan `Pandas DataFrame`.

[13]:
```
X_iris = iris.drop('species',axis =1) # ekstraksi Matriks Feature
```

[14]:
```
X_iris.shape # ukuruan matriks feature
```



Gambar 3.1: Ilustrasi matriks feature dan array/vektor target

[14]:
```
(150, 4)
```

[15]:
```
y_iris = iris['species'] # ekstraksi array target satu dimensi
```

[16]:
```
y_iris.shape # ukuran array target
```

[16]:
```
(150,)
```

1.2 Dasar-dasar *Applications Programming Interface* (API) dari *Scikit-Learn*

`Scikit-Learn` menyediakan banyak *built-in* algoritma dan model untuk *machine learning*, yang disebut dengan ***estimator***. Setiap estimator dapt di-*fit*-kan dengan data menggunakan metode `fit()`. **Dengan asumsi dataset yang akan kita gunakan telah tersedia** langkah-langkah yang umum untuk menggunakan estimator API pada `Scikit-Learn` adalah sebagai berikut:

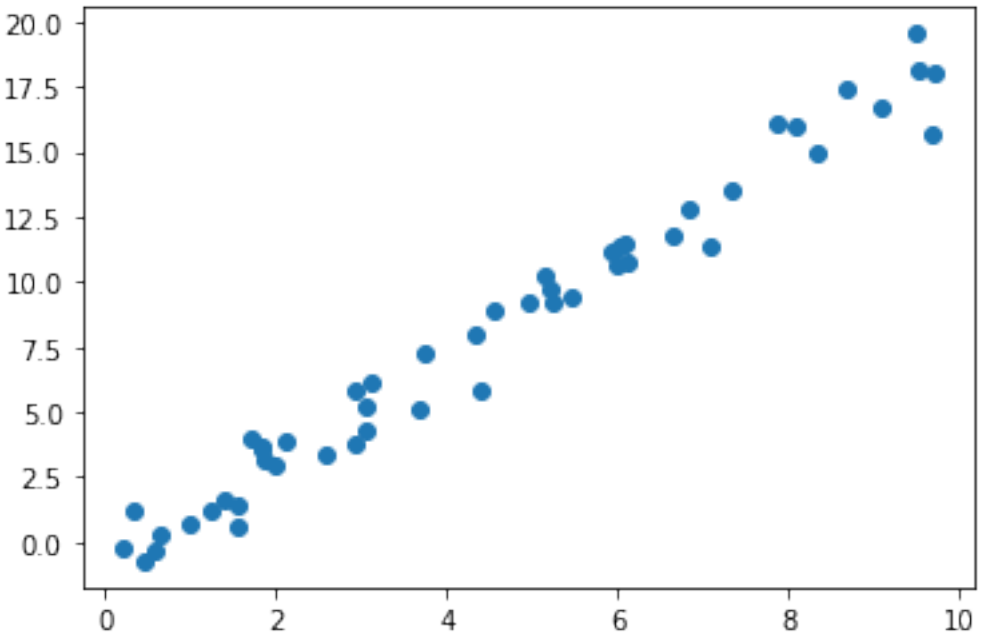
1. Memilih model *class* dengan melakukan impor *class* estimator yang cocok dari `Scikit-Learn`.
2. Memilih *hyperparameters* dengan *instantiating class* dengan harga-harga yang diinginkan.
3. Mengatur data kedalam matriks *feature* dan vektor target seperti yang telah dijelaskan sebelumnya, meskipun bisa jadi langkah ini dilakukan sebelumnya.
4. Melakukan *fitting* model terhadap data dengan fungsi `fit()` dari *instance* sebuah model.
5. Mengimplementasikan model terhadap data:
    - Pada *supervised learning*, dapat digunakan metode `predict()` untuk melakukan prediksi/klasifikasi data baru
    - Pada *unsupervised learning*, dapat digunakan metode `transform()` untuk mentransformasikan atau menyimpulkan sifat-sifat (*properties*) dari data dan juga dapat digunakan

`predict()` jikalau dibutuhkan

Contoh *supervised learning*: regresi linier sederhana

Sebagai contoh penerapan dari langkah-langkah di atas akan digunakan regeresi linier sederhana pada kasus *supervised learning*. Data yang digunakan dibangkingkan dengan program berikut.

[17]:
```
import matplotlib.pyplot as plt
import numpy as np

rng = np.random.RandomState(42)
x = 10*rng.rand(50)
y = 2*x-1+rng.randn(50)
plt.scatter(x,y);
```



Dengan data tersebut kita akan menggunakan lima langkah di atas secara berurutan:

1. Memilih model *class*

Pada `Scikit-Learn`, setiap model *class* direpresentasikan dengan Python *class*. Sehingga, untuk contoh di atas, kita akan gunakan model regresi linier dengan cara impor *class* regresi linier dengan cara berikut:

[18]:
```
from sklearn.linear_model import LinearRegression
```

Sebagai catatan, ada banyak model linier yang lain yang dapat dibaca pada dokumentasi modul `sklearn.linear_model`.

2. Pilih model *hyperparameter*

Kita harus mengetahui bahwa *class* sebuah model tidak sama dengan *instance* sebuah model. Setelah kita memutuskan model yang akan digunakan, masih terdapat beberapa pilihan yang masih terbuka untuk kita gunakan. Tergantung pada *class* model yang sedang digunakan, pertanyaan-pertanyaan berikut membantu menentukan perlu tidaknya menggunakan *hyperparameter*:

- Apakah kita akan melakukan *fitting* dengan *offset*-nya (*intercept*)?
- Apakah model akan dinormalisasi?
- Apakah kita akan melakukan *preprocessing* pada *feature* untuk membikan fleksibilitas pada model?
- Apakah kita akan menggunakan regularisasi? Sejauh apa?
- Berapa komponen-komponen model yang akan kita gunakan?

Pertanyaan-pertanyaan di atas merupakan contoh pilihan-pilihan penting yang harus dibuat setelah *class* model dipilih. Pilihan tersebut biasanya direpresentasikan dengan ***hyperparameter***, yaitu parameter-parameter yang harus di-*set* sebelum melakukan *fitting* model terhadap data.

Untuk contoh regresi linier di atas, kita dapat melakukan *instantiating* (melahirkan contoh/jenis) dari *class* `LinierRegression` dan menspesifikasikan bahwa kita akan melakukan *fitting intercept* menggunakan *hyperparameter* `fit_intercept`.

[19]:
```
model = LinearRegression(fit_intercept=True)
model
```

[19]:
```
LinearRegression(copy_X=True, fit_intercept=True, n_jobs=None, normalize=False)
```

Catatan. Ketika kita sudah melakukan *instantiating* sebuah model, langkah yang dilakukan hanya menyimpan harga-harga *hyperparameter* ini ke dalam sebuah memori, dan kita belum mengaplikasikan model tersebut terhadap data. API dari `Scikit-Learn` membuat perbedaan yang jelas antara pemilihan model dan aplikasi model terhadap data.

3. Mengatur data kedalam matriks *feature* dan vektor target

Seperti yang telah dijelaskan sebelumnya, `Scikit-Learn` merepresentasikan data dalam bentuk matriks *feature* dan array terget berdimensi satu (vektor target). Dalam contoh yang sedang kita kerjakan, vektor target sudah dalam bentuk yang benar (array dengan panjang n sampel), tetapi kita perlu membentuk matriks X menjadi berukuran `[n_samples,n_features]`. Hal ini bisa kita lakukan dengan *reshaping* dari yang sebelumnya berbentuk array 1 dimensi.

Data sebelumnya x berukuran:

[20]:
```
x.shape
```

[20]:
```
(50,)
```

dan diubah menjadi matriks X dengan cara

[21]:
```
X = x[:,np.newaxis]
X.shape
```

[21]:
```
(50, 1)
```

Terlihat bahwa X sekarang adalah matriks dengan ukuran 50×1 (jumlah sampel 50 dan jumlah *feature* 1).

4. Melakukan *fitting* model terhadap data (training)

Tahap ini mengimplementasikan model terhadap data (*fitting model* terhadap data), dengan menggunakan metode `fit()` dari model

[22]:
```
model.fit(X,y)
```

[22]:
```
LinearRegression(copy_X=True, fit_intercept=True, n_jobs=None, normalize=False)
```

Perintah `fit()` mengakibatkan terjadinya banyak komputasi internal yang tergantung pada model, dan hasil komputasi ini akan tersimpan pada atribut-atribut spesifik dari model tersebut yang dapat dieksplor oleh user. Pada `Scikit-Learn`, semua nama parameter model yang diperoleh (dipelajari) pada proses `fit()` mempunyai *trailing* berupa garis bawa (*underscore*). Sebagai contoh, pada model linier yang sedang kita kerjakan akan kita peroleh:

[23]:
```
model.coef_
```

[23]:
```
array([1.9776566])
```

[24]:
```
model.intercept_
```

[24]:
```
-0.9033107255311164
```

Kedua parameter di atas berturut-turut menunjukan gradien $\approx 1,97$ atau kemiringan (*slope*) dan juga *intercept* ≈ -0.903 dari garis hasil *fitting* terhadap data. Terlihat bahwa hasil estimasi parameter dekat dengan parameter yang digunakan untuk membangkitkan data sebelum langkah ke-1, dengan menggunakan persamaan $y = 2x - 1 +$ noise, dimana gradien adalah 2 dan *intercept* -1.

5. Memprediksi data baru

Setelah model ditraining dengan menggunakan training dataset, tugas utama dari *supervised learning* adalah melakukan evaluasi berdasarkan prediksi data baru yang bukan bagian dari training dataset. Pada `Scikit-Learn`, dapat digunakan metode `predict()`. Pada contoh ini, data baru dibuat berupa beberapa harga x dan dilihat hasil prediksi harga y untuk masing-masing harga x tersebut.

[25]:
```
x_new = np.linspace(-1,11)
```

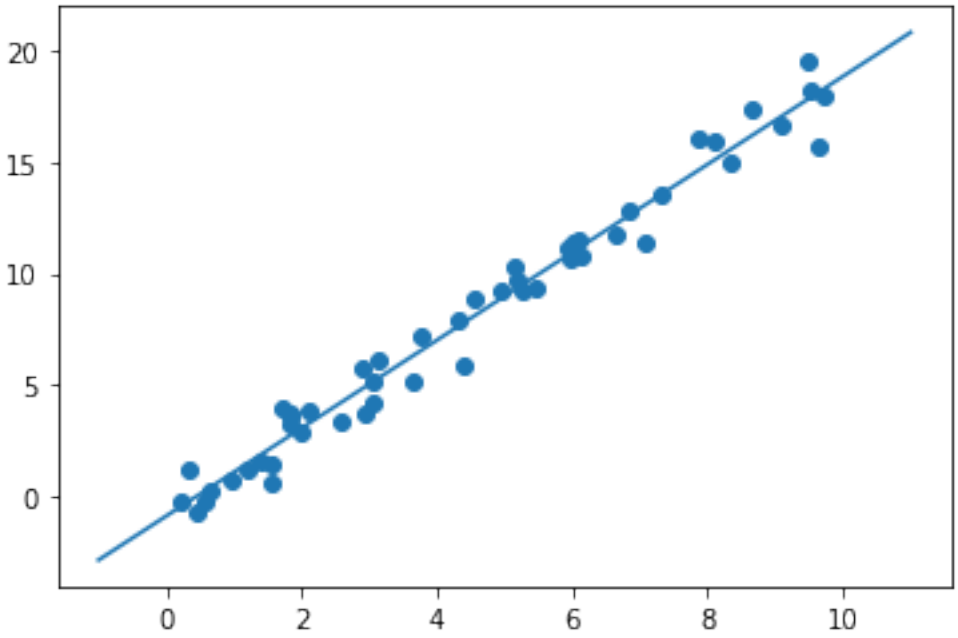
seperti sebelumnya kita harus merubah x_new ke dalam bentuk `[n_samples,n_features]` dari matriks *feature*, kemudian dijadikan input pada model

[26]:
```
X_new = x_new[:,np.newaxis]
y_fit_new = model.predict(X_new)
```

Untuk lebih jelas, kita dapat memvisualisasikan dataset training sebelumnya dan data hasil prediksi sebagai berikut:

[27]:
```
plt.scatter(x,y)
plt.plot(x_new,y_fit_new)
```

[27]:
```
[<matplotlib.lines.Line2D at 0x1a22fba7d0>]
```



Sebagai catatan, langkah ke-3 di atas bisa jadi dilakukan langsung setelah dataset training kita peroleh, seperti pada contoh berikut.

2 Dataset MNIST

Jika contoh pada section sebelumnya data dibangkitkan sendiri, pada bagian ini akan digunakan dataset MNIST sehingga lebih realistis. Jika contoh sebelumnya memberikan contoh kasus regresi, pada bagian ini akan dijelaskan contoh kasus klasifikasi. Selain itu, contoh sebelumnya hanya menggunakan satu *feature*, pada contoh ini kita akan menggunakan banyak *feature* yang menyatakan intensitas setiap piksel pada gambar hasil tulisan tangan.

Dataset MINST terdiri dari 70000 gambar ukuran kecil dari dijit tulisan tangan anak-anak sekolah menengah dan para pegawai biro sensus di Amerika. Setiap gambar dilabeli dengan dijit yang sesuai dengan tulisan tangannya. Dataset ini sering digunakan oleh para peneliti yang mempunyai ide atau metode klasifikasi baru, dan juga sering disebut "Hello World" dari ML.

Banyak dataset-dataset yang dapat diakses secara gratis di Internet. Beberapa repository dari yang menyediakan dataset gratis untuk *machine learning* adalah:

- Repositori open data yang populer
  - UC Irvine Machine Learning Repository

  - Kaggle Datasets
  - Amazon's AWS Datasets
- Meta portals (list dari repositori open data)
  - Data Portals
  - OpenDataMonitor
  - Quandl
- Alamat-alamat web lain untuk listing repositori data populer
  - Wikipedia'Listing of machine learning datasets
  - Quora.com
  - The datasets subreddit

`Scikit-Learn` menyediakan banyak fungsi *helper* untuk mendownload dataset-dataset yang populer, dan MINST salah satunya. Kode berikut melakukan penarikan (*fetching*) dataset MINST (detail untuk pengambilan data lihat di openml)

[28]:
```
from sklearn.datasets import fetch_openml
mnist = fetch_openml('mnist_784', version=1)
mnist.keys()
```

[28]:
```
dict_keys(['data', 'target', 'frame', 'feature_names', 'target_names', 'DESCR',
'details', 'categories', 'url'])
```

Dataset yang di-*load* dengan `Scikit-Learn` biasanya mempunyai struktur *dictionary* yang serupa, termasuk diantaranya: * Key `DESCR` menjelaskan dataset * Key `data` berisi array dengan baris menyatakan sampel (*instance*) dan kolom menyatakan *feature* * Key `target` berisi array dari label

Mengatur data kedalam matriks *feature* dan vektor target

Isi dari masing-masing array, yang merupakan matriks *feature* X dan vektor target y, dapat diakses dengan perintah berikut:

[29]:
```
X, y = mnist["data"], mnist["target"]
X.shape
```

[29]:
```
(70000, 784)
```

[30]:
```
y.shape
```

[30]:
```
(70000,)
```

Terlihat bahwa jumlah elemen pada vektor target y (label) sebanyak 70000, sesuai dengan jumlah baris pada `data` X yang menyatakan banyaknya sampel training. Dataset MINST terdiri dari 70000 gambar dan setiap gambar mempunyai 784 *features*. Jumlah *feature* sebanyak 784 diperoleh dari setiap gambar yang berukuran 28×28 pixels, dan setiap *feature* merepresentasikan intensitas

sebuah pixel (dari 0 untuk warna putih dan 255 untuk warna hitam). Untuk mengambil contoh satu dijit dari dataset dapat digunakan program berikut:

[31]:
```
%matplotlib inline
import numpy as np
import matplotlib as mpl
import matplotlib.pyplot as plt

some_digit = X[0]
some_digit_image = some_digit.reshape(28, 28)
plt.imshow(some_digit_image, cmap=mpl.cm.binary)
plt.axis("off")
plt.show()
```

Terlihat seperti angka '5', dan memang sama dengan label yang tersimpan untuk data tersebut:

[32]:
```
y[0]
```

[32]:
```
'5'
```

Luaran dari label adalah bertipe `string`, sedangkan kebanyakan algoritma ML menggunakan angka. Oleh sebab itu diperlukan perubahan tipe data (*type casting*):

[33]:
```
type(y[0])
```

[33]:
```
str
```

[34]:
```
y = y.astype(np.uint8)
```

[35]:
```
type(y[0])
```

[35]:
```
numpy.uint8
```

Untuk memberikan ilustrasi menyangkut kompleksitas dari proses pengklasifikasian, gambar 3.2 berikut menunjukan beberapa gambar dari tulisan tangan pada dataset MINST.

3.2: Contoh-contoh dijit dari dataset MINST

Selain data training, kita juga harus menyediakan data test. Dataset MINST sebetulnya sudah terbagi menjadi training set (60000 gambar pertama) dan test set (10000 gambar terakhir), yang bisa dipisahkan dengan cara berikut.

[36]:
```
X_train, X_test, y_train, y_test = X[:60000], X[60000:], y[:60000], y[60000:]
```

Untuk sementara kita akan melakukan simplifikasi masalah dengan hanya mempertimbangkan satu angka saja, misalkan nomor 5. Sistem ini dirancang hanya untuk membedakan angka 5 dan bukan 5. Sehingga kita akan melakukan modifikasi dari vektor target menjadi 2 *class*, yaitu *class* angka 5 dan bukan *class* angka lima dengan cara sebagai berikut:

[37]:
```
y_train_5 = (y_train == 5)
y_test_5 = (y_test == 5)
```

Kode diatas menyatakan jika nilai target untuk training dan test sama dengan 5 maka akan meng-

hasilkan `True`, dan yang bukan akan menghasilkan `False` yang disimpan di variable `y_train_5` dan `y_test_5`.

[38]:
```
y_train_5
```

[38]:
```
array([ True, False, False, ...,  True, False, False])
```

Memilih model class *classifier*

Pada kasus ini kita akan menggunakan *classifier Stochastic Gradient Descent* (SGD) di `Scikit-Learn`, yaitu *class* `SGDClassifier`. *Classifier* ini mempunyai kelebihan dapat melakukan *handling* dataset yang sangat besar secara efisien. Hal ini karena SGD berhubungan dengan sampel-sampel training secara independen (satu persatu), sehingga SGD juga cocok untuk *online learning*. Cara membuat *classifier* dan melakukan training adalah sebagai berikut (Detail keterangan untuk penggunaan `SGDClassifier` pada model linier dapat dilihat di SGD-Classifier).

[39]:
```
from sklearn.linear_model import SGDClassifier
```

Pilih model *hyperparameter*

Karena SGD digunakan pada kasus ini, maka *hyperparameters* yang akan kita pilih pada contoh ini adalah banyaknya iterasi maksimum (`max_iter`) untuk mendapatkan parameter yang dapat meminimalkan *cost function*, dan toleransi perbedaan parameter selanjutnya dan yang sekarang (`tol`).

[40]:
```
sgd_clf = SGDClassifier(max_iter=1000, tol=1e-3, random_state=42)
```

Melakukan *fitting* model terhadap data (melakukan training)

Untuk melakukan training seperti biasa, bisa digunakan metode `fit()` dengan matriks *feature* dan vektor target sebagai argumen.

[41]:
```
sgd_clf.fit(X_train, y_train_5)
```

[41]:
```
SGDClassifier(alpha=0.0001, average=False, class_weight=None,
              early_stopping=False, epsilon=0.1, eta0=0.0, fit_intercept=True,
              l1_ratio=0.15, learning_rate='optimal', loss='hinge',
              max_iter=1000, n_iter_no_change=5, n_jobs=None, penalty='l2',
              power_t=0.5, random_state=42, shuffle=True, tol=0.001,
              validation_fraction=0.1, verbose=0, warm_start=False)
```

Memprediksi data baru

[42]:
```
sgd_clf.predict([some_digit])
```

[42]:
```
array([ True])
```

`some_digit` dari kode sebelumnya berisi angka lima, dapat terlihat bahwa data tersebut dikenali sebagai angka 5 juga. Apabila kita coba angka lain misalkan 0 yang disimpan pada variabel `X[1]`

[43]:
```
sgd_clf.predict([X[1]])
```

[43]:
```
array([False])
```

Maka dikenali sebagai bukan angka 5.

3 Validasi Model

Pada bagian sebelumnya telah dijelaskan langkah-langkah dasar untuk mengimplementasikan supervised learning. Langkah memilih model maupun *hyperparameter* bisa jadi merupakan bagian terpenting dalam keberhasilan menggunakan teknik ini secara efektif. Untuk membuat pilihan yang tepat, kita memerlukan cara untuk memvalidasi model dan *hyperparameter* tersebut apakah telah menghasilkan *fitting* data dengan baik.

Secara prinsip, validasi model merupakan hal yang sederhana. Setelah memilih model dan *hyperparameter*, kita dapat melakukan estimasi seberapa efektif implementasinya pada data training dan membandingkan hasil prediksi dengan harga-harga yang telah diketahui. Pada bagian selanjutnya akan kita lihat contoh yang salah dan benar dalam melakukan validasi model.

Validasi model dengan cara yang salah

Kita akan mendemonstrasikan pendekatan yang salah dalam validasi model menggunakan data bunga Iris, yang sudah kita lihat di contoh-contoh sebelumnya. Kita mulai dengan melakukan *data loading*

[44]:
```
from sklearn.datasets import load_iris
iris = load_iris()
X = iris.data
y = iris.target
```

Kemudian kita pilih model dan *hyperparameter*. Dalam contoh ini kita akan menggunakan model yang sederhana dan intuitif yaitu *K*-nearest *classifier* dengan `n_neighbors=1`. Hasil klasifikasi yaitu (*label*) dari data baru (yang akan diklasifikasikan) adalah label data training terdekat.

[45]:
```
from sklearn.neighbors import KNeighborsClassifier
model = KNeighborsClassifier(n_neighbors=1)
```

Kemudian kita training model dan menggunakannya untuk memprediksikan label dari data yang telah diketahui (data training):

[46]:
```
model.fit(X, y)
y_model = model.predict(X)
```

Akhirnya kita menghitung fraksi (persentase) data point yang diklasifikasikan secara benar:

[47]:
```
from sklearn.metrics import accuracy_score
accuracy_score(y, y_model)
```

[47]:
```
1.0
```

Kita lihat bahwa skor untuk akurasi adalah 1.0, yang mengindikasikan bahwa 100% data dilabeli atau diklasifikasikan secara benar oleh model yang kita gunakan. Tetapi apakah hasil pengukuran akurasinya benar? Apakah kita benar-benar menemukan model yang kita harapkan betul 100% dalam keseluruhan waktu? Tentu saja jawabannya tidak. Sebetulnya, pendekatan ini mempunyai kesalahan fatal, yaitu **melakukan training dan evaluasi model menggunakan data yang sama**. Lebih jauh, model *nearest-neighbor* merupakan model estimator *instance-based* (lihat modul 1) yang hanya menyimpan data training dan memprediksi label dengan membandingkan data baru dengan data-data yang disimpan tadi. Sehingga akan selalu didapatkan akurasi 100%.

Validasi model dengan cara yang benar: *holdout set*

Cara yang lebih baik untuk memvalidasi model dibandingkan dengan cara sebelumnya adalah dengan *holdout set*, yaitu, kita pisahkan beberapa subset dari data training, yang kemudian akan digunakan untuk evaluasi. Pemisahan (*splitting*) ini dapat dilakukan dengan menggunakan metode `train_test_split` pada *Scikit-Learn*:

[48]:
```
from sklearn.model_selection import train_test_split
# split the data with 50% in each set
X1, X2, y1, y2 = train_test_split(X, y, random_state=0,
                                  train_size=0.5)

# fit the model on one set of data
model.fit(X1, y1)

# evaluate the model on the second set of data
y2_model = model.predict(X2)
accuracy_score(y2, y2_model)
```

[48]:
```
0.9066666666666666
```

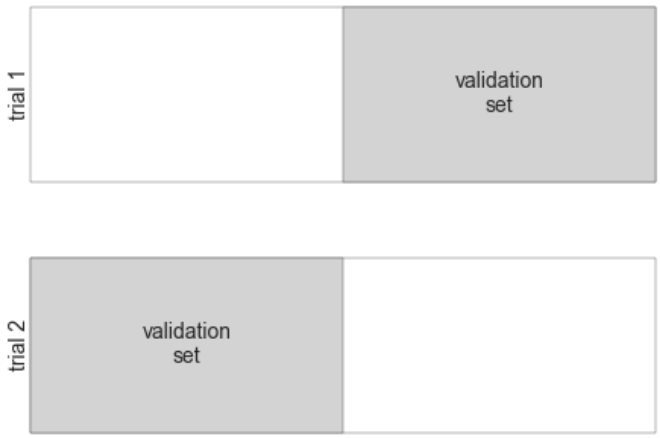
Kita mendapatkan nilai yang lebih valid, yaitu *classifier* dengan *nearest neighbor* mempunyai akurasi 90% pada set *holdout* ini. Set *holdout* serupa dengan data yang tidak diketahui (*unknown data*), karena model belum pernah mendapatkannya sebelumnya.

Validasi model dengan cara yang benar: *cross validation*

Kerugian dalam penggunaan set *holdout* untuk validasi model adalah kita kehilangan porsi data training yang cukup banyak. Pada kasus diatas, setengah dari dataset tidak digunakan untuk mentraining model. Hal ini membuat tidak optimal, dan dapat menyebabkan masalah terutama jumlah data training sedikit.

Salah satu cara untuk mengatasi hal tersebut adalah dengan *cross validation*, yaitu melakukan urutan *fitting* dimana setiap subset dari data dapat digunakan untuk training maupun untuk validasi. Gambar 3.3 mengilustrasikan hal tersebut secara visual.

Pada contoh ini kita mempunyai 2 percobaan validasi, menggunakan masing-masing setengah dari data sebagai set *holdout* secara bergantian. Berikut implementasi *cross validation* menggunakan data yang sudah dipisah sebelumnya:



Gambar 3.3: Ilustrasi konsep cross validation

[49]:
```
y2_model = model.fit(X1, y1).predict(X2)
y1_model = model.fit(X2, y2).predict(X1)
accuracy_score(y1, y1_model), accuracy_score(y2, y2_model)
```

[49]:
```
(0.96, 0.9066666666666666)
```

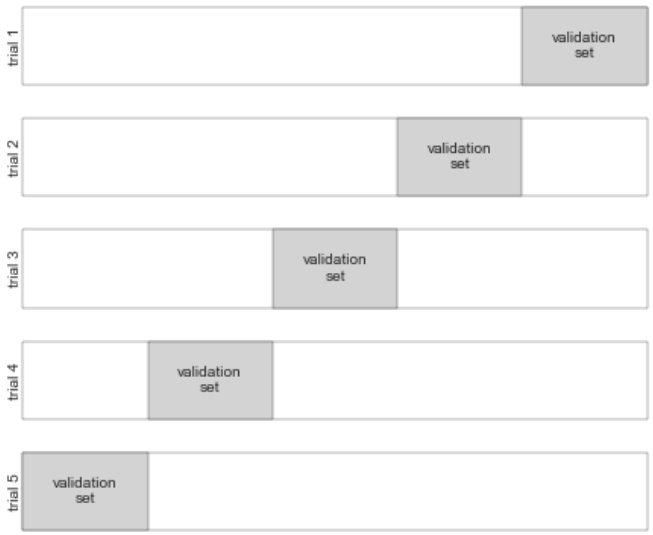
Terlihat bahwa terdapat 2 skor akurasi, yang dapat dikombinasikan, misalkan dengan mengambil rata-rata untuk mendapatkan ukuran kinerja model global secara lebih baik. Bentuk *cross-validation* ini disebut *two-fold cross-validation*, dimana kita telah membagi kedalam dua set data dan mengunakannya secara bergantian untuk set validasi. Kita dapat mengembangkan ide ini dengan membuat lebih banyak *trials* dan membuat lebih banyak lagi *fold* pada data. Sebagai contoh Gambar 3.4 mengilustrasikan *five-fold cross-validation*:

Pada kasus ini kita memisahkan data kedalam 5 grup, dan menggunakannya masing-masing secara bergantian untuk evaluasi model yang ditraining dengan 4/5 data yang lain. Untuk memudahkan, kita bisa gunakan `cross_val_score` pada *library* `Scikit-Learn`:

[50]:
```
from sklearn.model_selection import cross_val_score
cross_val_score(model, X, y, cv=5)
```

[50]:
```
array([0.96666667, 0.96666667, 0.93333333, 0.93333333, 1.        ])
```

Dengan mengulangi validasi terhadap subset-subset data yang berbeda akan memberikan gambaran yang lebih baik dari kinerja algoritma. `Scikit-Learn` mengimplementasikan beberapa skema *cross validation* yang berguna dalam situasi tertentu dengan menggunakan modul `cross_validation`. Sebagai contoh, mungkin kita menginginkan kasus ekstrim dimana jumlah *fold* sama dengan jumlah data, yang artinya melakukan training untuk keselurahan data kecuali satu data yang akan digunakan untuk validasi, dan hal tersebut dilakukan secara bergiliran untuk keseluruhan data training. Cara *cross validation* seperti ini disebut dengan *leave-one-out cross*



Gambar 3.4: Ilustrasi konsep cross validation dengan menggunakan 5-fold cross validation

validation, dan dapat dilakukan sebagai berikut:

[51]:
```
from sklearn.model_selection import LeaveOneOut
scores = cross_val_score(model, X, y, cv=LeaveOneOut())
scores
```

[51]:
```
array([1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1.,
       1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1.,
       1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1.,
       1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1.,
       1., 1., 0., 1., 0., 1., 1., 1., 1., 1., 1., 1., 1., 1., 0., 1.,
       1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1.,
       1., 1., 1., 1., 0., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1.,
       0., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 0., 1., 1.,
       1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1.])
```

Karena kita mempunyai 150 sampel, *leave-one-out cross validation* akan menghasilkan skor2 untuk 150 trials, dan skor tersebut menunjukan prediksi yang sukses (1.0) atau gagal (0.0). Dengan mengambil rata-ratanya akan diperoleh estimasi dari laju error (*error rate*):

[52]:
```
scores.mean()
```

[52]:
```
0.96
```

Metode-metode *cross validation* yang lain dapat digunakan secara serupa, yang dapat dilihat secara detail di Scikit-Learn's online dokumentasi cross-validation.

4 Memilih Model Terbaik

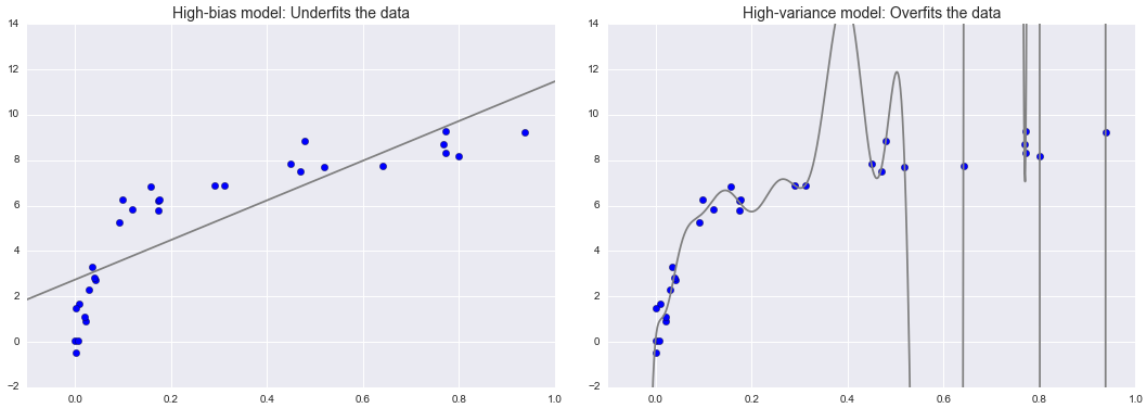
Sampai sejauh ini kita telah mengetahui dasar-dasar validasi dan *cross-validation*, pada bagian ini kita akan menggali lebih jauh menyangkut pemilihan model (*model selection*) dan pemilihan *hyperparameter*. Masalah pemilihan ini merupakan salah satu aspek praktis yang sangat penting pada *machine learning*, tetapi pada buku-buku *machine learning* seringkali hanya dibahas sepintas di bagian perkenalan. Pertanyaan pentingnya adalah **jika estimator kita mempunyai kinerja rendah (*underperforming*), apa yang akan kita lakukan?**. Beberapa jawaban yang mungkin diantaranya adalah:

- Gunakan model yang lebih kompleks atau model yang lebih fleksibel
- Gunakan model yang lebih sederhana atau model yang lebih kaku
- Gunakan lebih banyak data training
- Gunakan lebih banyak data untuk menambahkan fitur pada setiap sampel

Jawaban-jawaban pada pertanyaan tersebut terkadang *counter-intuitive*. Model yang lebih kompleks boleh jadi akan memberikan hasil yang lebih buruk, dan menambahkan data training tidak akan juga memperbaiki hasil. Kemampuan untuk menentukan apa langkah terbaik untuk memperbaiki model merupakan aspek yang dapat membedakan praktisi *machine learning* yang berhasil dan yang tidak berhasil.

4.1 *Trade-off* Bias dan Variansi

Pertanyaan **model terbaik** adalah tentang bagaimana mencari posisi terbaik pada *tradeoff* antara **bias dan variansi**. Gambar 3.5 merepresentasikan dua cara *fitting* terhadap data yang sama dengan model regresi yang berbeda. Gambar sebelah kiri digunakan model regresi linier (orde polinomial 1), sedangkan sebelah kanan menggunakan model regresi polinomial yang lebih tinggi. Terlihat bahwa kedua model gagal melakukan *fitting* yang baik dengan cara yang berbeda.



Gambar 3.5: Trade-off antara bias dan variansi

Model sebelah kiri berusaha untuk melakukan *fitting* dengan menggunakan garis lurus melalui data. Model garis lurus tidak akan bisa mendeskirpsikan atau merepresentasikan data dengan baik, karena data tersebut secara intrinsik lebih kompleks dibandingkan dengan garis lurus. Sehingga model biasanya disebut ***Underfitting*** terhadap data. Dengan kata lain model tidak mempunyai cukup fleksibilitas untuk memperhitungkan secara tepat semua *feature-feature* yang ada

Gambar 3.6: Hasil prediksi y dari data-data baru (poin-poin warna merah) dan penentuan coefficient of determination

pada data, atau dapat dikatakan juga bahwa model memiliki **bias** yang tinggi.

Model sebelah kanan berusaha untuk melakukan *fitting* dengan polinomial dengan orde tinggi yaitu 20. Model ini terlihat mempunyai fleksibilitas tinggi sehingga bisa akurat merepresentasikan dataset. Tetapi meskipun demikian keakuratan tersebut lebih cenderung lebih merefleksian sifat-sifat noise tertentu dari data, daripada merepresentasikan sifat intrinsik proses apapun yang membangkitkan data. Model dengan kondisi ini disebut ***overfitting*** terhadap data. Dengan kata lain model terlalu fleksibel sehingga model juga memperhitungkan error-error acak selain distribusi data yang seharusnya. Dengan kata lain model mempunyai **variansi** yang tinggi.

Untuk melihat masalah ini dari sisi lain, dimisalkan kita akan menggunakan kedua model tersebut untuk memprediksi harga-harga y dari beberapa data baru yang ditunjukan dengan poin-poin merah pada Gambar 3.6. Sebelumnya data-data baru tersebut tidak digunakan untuk mentraining kedua model.

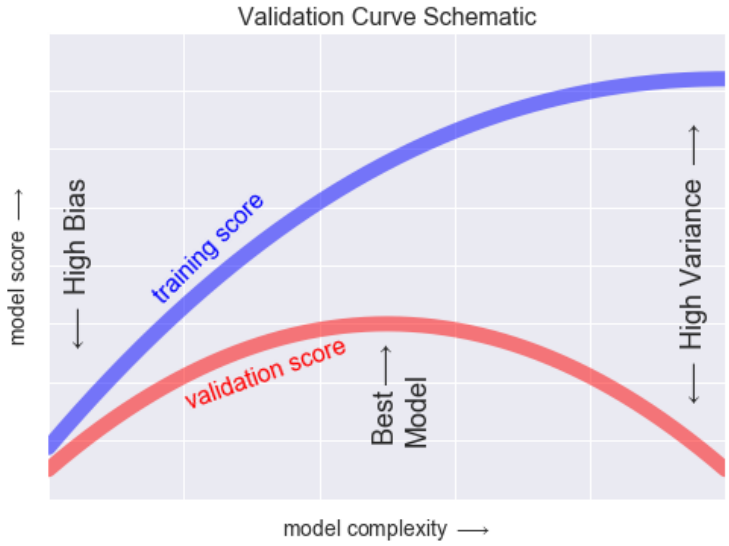
Skor yang digunakan adalah R^2, atau coefficient of determination yang mengukur seberapa baik kinerja model relatif terhadap mean dari harga-harga target. $R^2 = 1$ menunjukan sangat cocok, sedangkan $R^2 = 0$ menunjukan model tidak lebih baik dari hanya mengambil rata-rata dari data, dan harga negatif menunjukan model yang jauh lebih buruk lagi.

Dari skor-skor yang dihasilkan oleh kedua model, dapat membuat kesimpulan secara umum sebagai berikut:

- Untuk model dengan bias yang tinggi, kinerja model pada dataset validasi serupa dengan kinerja dari model pada dataset training.
- Untuk model dengan variansi yang tinggi, kinerja model pada dataset validasi jauh lebih buruk daripada kinerja model pada dataset training.

Jika kita mempunyai kemampuan untuk melakukan pengaturan (*tuning*) kompleksitas model, kita akan peroleh kurva skor untuk training dan validasi sebagai fungsi dari kompleksitas model seperti yang ditunjukan pada Gambar 3.7.

Kurva pada Gambar 3.7 biasanya disebut juga dengan kurva validasi (*validation curve*) dengan hasil simpulan berikut:



Gambar 3.7: Kurva validasi

- Skor training selalu lebih tinggi dibandingkan dengan skor validasi. Hal ini terjadi secara umum, model akan *fitting* lebih baik terhadap data yang telah dilihat sebelumnya (melalui training) dibandingkan dengan data yang belum dilihat sebelumnya(termasuk data validasi/test).
- Untuk model dengan kompleksitas rendah (model bias tinggi), data training mengalami *underfitting* yang berarti model merupakan prediktor yang buruk baik untuk data training maupun data-data yang belum terlihat sebelumnya.
- Untuk model dengan kompleksitas tinggi (model variansi tinggi), data training mengalami *overfitting* yang berarti model memprediksi data training sangat baik, tetapi gagal menunjukan kinerja yang sama untuk data-data yang belum terlihat sebelumnya.
- Untuk model dengan level kompleksitas diantara keduanya, kurva validasi akan mencapai maksimum. Level kompleksitas tersebut mengindikasikan *trade-off* yang cocok antara bias dan variansi.

Sebagai catatan, cara untuk melakukan *tuning* model bervariasi dari satu model ke model yang lain.

4.2 Kurva validasi pada `Scikit-Learn`

Berikut akan diberikan contoh menggunakan *cross validation* untuk menghitung kurva validasi dari model sebuah *class*. Kita akan menggunakan regresi polinomial sebagai *generalized linear model* dimana orde polinomial merupakan parameter yang dapat diatur. Misalkan, orde polinomial 1 merupakan garis lurus melalui data, untuk parameter model a dan b dapat dituliskan sebagai berikut:

$y = ax + b$

Sedangkan polinomial orde 3 dengan parameter model a, b, c, d:

$y = ax^3 + bx^2 + cx + d$

Kita dapat melakukan generalisasi pada sembarang jumlah *feature* polinomial. Pada `Scikit-learn`, kita dapat mengimplementasikan model linier ini dikombinasikan dengan *preprocessor* polinomial dengan menggunakan `pipeline` (detail menyangkut`pipeline` dapat dilihat di pipeline dan komposit estimator).

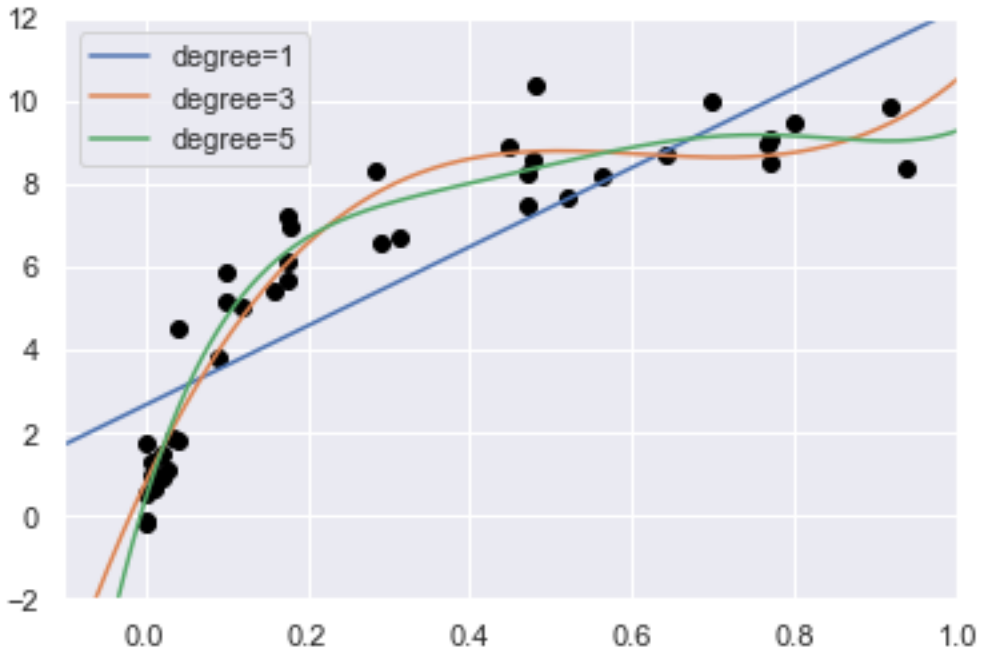
[53]:
```
from sklearn.preprocessing import PolynomialFeatures
from sklearn.linear_model import LinearRegression
from sklearn.pipeline import make_pipeline

def PolynomialRegression(degree=2, **kwargs):
    return make_pipeline(PolynomialFeatures(degree),
                         LinearRegression(**kwargs))
```

Kita bangkitkan data sintetik yang akan di-*fitting* dengan model yang telah dibuat:

[54]:
```
import numpy as np

def make_data(N, err=1.0, rseed=1):
    # randomly sample the data
    rng = np.random.RandomState(rseed)
    X = rng.rand(N, 1) ** 2
    y = 10 - 1. / (X.ravel() + 0.1)
    if err > 0:
        y += err * rng.randn(N)
    return X, y

X, y = make_data(40)
```

Visualisasi dari data yang telah dibangkitkan, dengan hasil *fitting* model polinomial dengan beberapa orde (1, 3 dan 5):

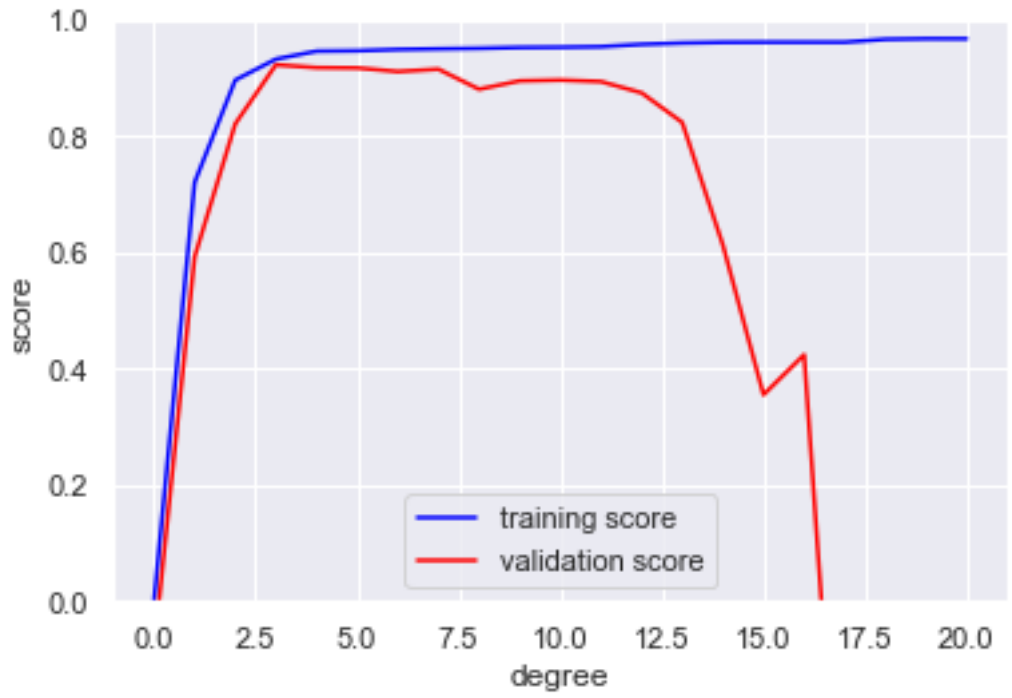
[55]:
```
%matplotlib inline
import matplotlib.pyplot as plt
import seaborn; seaborn.set()  # plot formatting

X_test = np.linspace(-0.1, 1.1, 500)[:, None]

plt.scatter(X.ravel(), y, color='black')
axis = plt.axis()
for degree in [1, 3, 5]:
    y_test = PolynomialRegression(degree).fit(X, y).predict(X_test)
    plt.plot(X_test.ravel(), y_test, label='degree={0}'.format(degree))
plt.xlim(-0.1, 1.0)
plt.ylim(-2, 12)
plt.legend(loc='best');
```



Untuk pengontrolan kompleksitas model pada kasus ini adalah orde polinomial, berupa integer *non-negative*. Pertanyaan selanjutnya adalah: **Berapa orde polinomial yang menghasilkan trade-off paling baik antara bias (*underfitting*) dan variansi (*overfitting*)?**

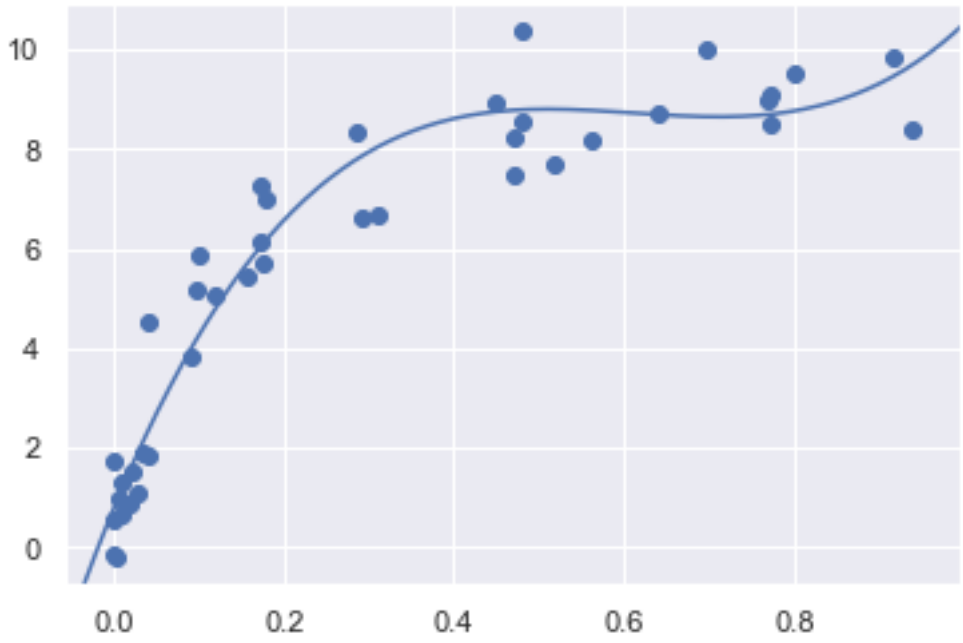
Kita dapat menjawab pertanyaan ini dengan memvisualisasikan kurva validasi untuk data dan model ini, yang bisa dilakukan secara langsung dengan menggunakan rutin `validation_curve` dari `Scikit-Learn`. Dengan diberikan sebuah model, data, nama parameter, dan sebuah range untuk dieksplorasi, rutin ini akan menghitung secara otomatis skor training dan validasi pada range tersebut.

[56]:
```
from sklearn.model_selection import validation_curve
degree = np.arange(0, 21)
train_score, val_score = validation_curve(PolynomialRegression(), X, y,
                                          'polynomialfeatures__degree', degree,␣
 ↪cv=7)

plt.plot(degree, np.median(train_score, 1), color='blue', label='training score')
plt.plot(degree, np.median(val_score, 1), color='red', label='validation score')
plt.legend(loc='best')
plt.ylim(0, 1)
plt.xlabel('degree')
plt.ylabel('score');
```



Grafik di atas menunjukan sifat kualitatif yang kita harapkan, yaitu skor training lebih tinggi dibandingkan dengan skor validasi. Skor training secara monoton membaik dengan bertambahnya kompleksitas model dan skor validasi mencapai maksimum sebelum akhirnya turun secara drastis karena model mulai *overfitting*. Dari kurva validasi, kita dapat melihat bahwa trade-off optimal antara bias dan variansi ditemukan pada polinomial orde ketiga. Hasil *fitting* model polinomial orde ketiga pada data dapat dilihat pada gambar berikut.

[57]:
```
plt.scatter(X.ravel(), y)
lim = plt.axis()
y_test = PolynomialRegression(3).fit(X, y).predict(X_test)
plt.plot(X_test.ravel(), y_test);
plt.axis(lim);
```

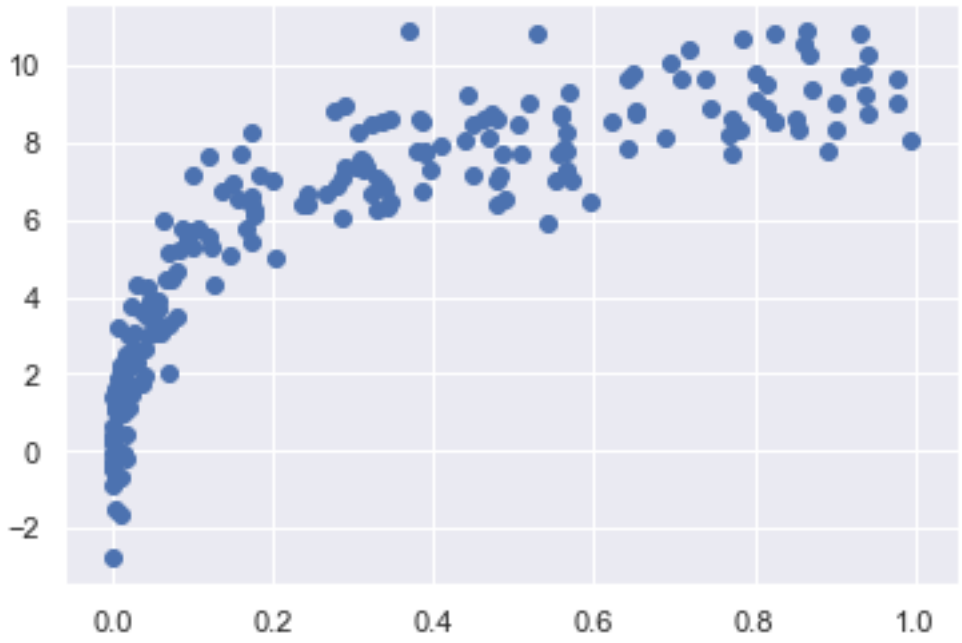


Harap diperhatikan bahwa menemukan model optimal tidak mengharuskan kita untuk menghitung skor training, tetapi dengan memeriksa hubungan antara skor training dan skor validasi bisa memberikan *insight* yang baik menyangkut kinerja model.

5 Kurva *Learning*

Salah satu masalah penting dari aspek kompleksitas model adalah adanya ketergantungan model yang optimal terhadap ukuran data training. Dimisalkan kita membangkitkan dataset baru dengan jumlah data yang lebih banyak dari yang sebelumnya.

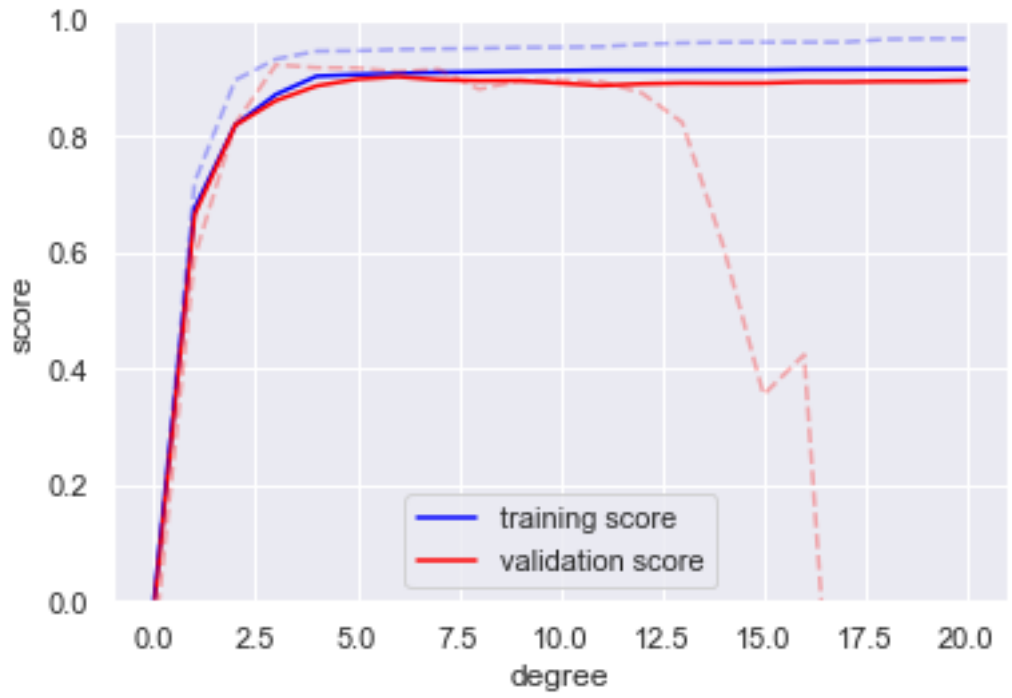
[58]:
```
X2, y2 = make_data(200)
plt.scatter(X2.ravel(), y2);
```



5.1 Pengertian dan perilaku kurva *learning*

Kita akan menduplikasi kode program sebelumnya yaitu *plotting* kurva validasi dan digunakan untuk dataset yang lebih besar seperti yang sudah dibangkitkan di atas (kurva dengan garis tebal). Sebagai pebanding, hasil kurva validasi sebelumnya dengan jumlah dataset yang kecil tetap ditampilkan (kurva dengan garis putus-putus).

[59]:
```
degree = np.arange(21)
train_score2, val_score2 = validation_curve(PolynomialRegression(), X2, y2,
                                            'polynomialfeatures__degree',␣
 ↪degree, cv=7)

plt.plot(degree, np.median(train_score2, 1), color='blue', label='training␣
 ↪score')
plt.plot(degree, np.median(val_score2, 1), color='red', label='validation score')
plt.plot(degree, np.median(train_score, 1), color='blue', alpha=0.3,␣
 ↪linestyle='dashed')
plt.plot(degree, np.median(val_score, 1), color='red', alpha=0.3,␣
 ↪linestyle='dashed')
plt.legend(loc='lower center')
plt.ylim(0, 1)
plt.xlabel('degree')
plt.ylabel('score');
```



Dari gambar di atas, garis tebal menunjukan hasil yang baru sedangkan garis samar putus-putus menunjukan hasil dari dataset yang lebih kecil. Berdasarkan perbandingan kurva validasi di atas, dataset yang lebih besar dapat mendukung model yang lebih kompleks, dimana puncaknya di sekitar orde 6 (sebelumnya orde 3 dengan dataset yang lebih kecil). Bahkan lebih jauh lagi polinomial orde 20 belum bisa dikatakan terjadi *overfitting*, karena skor training dan skor validasi masih berdekatan.

Sehingga dapat kita simpulkan bahwa perilaku kurva validasi tidak hanya tergantung pada kompleksita model, tetapi juga jumlah data training. Kadang-kadang, kita membutuhkan eksplorasi perilaku model sebagai fungsi dari jumlah data training, yaitu dengan cara meningkatkan jumlah data untuk *fitting* terhadap model. Kurva yang menunjukkan skor training/validasi terhadap ukuran dataset training disebut dengan **kurva *learning***.

Perilaku umum yang dapat kita harapkan dari kurva *learning* adalah sebagai berikut:

- Model dengan kompleksitas tertentu akan akan menghasilkan *overfitting* pada dataset yang kecil, artinya skor training akan relatif tinggi, sedangkan skor validasi akan relatif rendah.
- Model dengan kompleksitas tertentu akan menghasilkan *underfitting* pada dataset yang besar, artinya skor training akan turun, tetapi skor validasi akan bertambah.
- Model tidak akan memberikan skor yang lebih tinggi terhadap set validasi dibandingkan set training kecuali secara kebetulan, artinya kurva bisa saling mendekat tetapi tidak pernah bersilangan.

Dengan sifat-sifat ini, kita dapat simpulkan bahwa kurva *learning* secara kualitatif akan terlihat seperti pada Gambar 3.8.

Sifat yang paling menonjol pada kurva *learning* adalah konvergensi ke skor tertentu ketika

jumlah sampel training membesar. Tetapi, ketika jumlah sampel training yang kita gunakan sudah dikatakan cukup, maka penambahan jumlah sampel training tidak akan membantu meningkatkan performansi lebih jauh. Salah satu cara yang mungkin untuk meningkatkan kinerja model pada kasus ini adalah menggunakan model lain yang bisa jadi lebih kompleks.

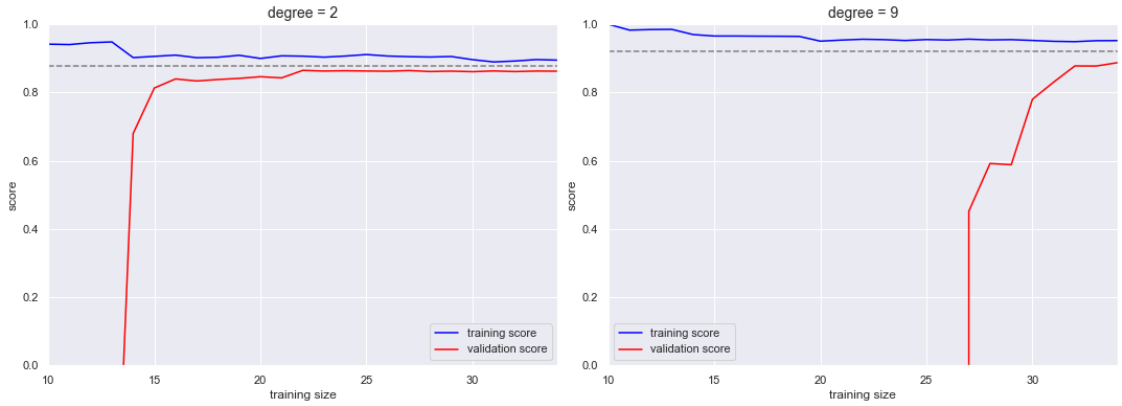
Gambar 3.8: Bentuk kurva learning secara umum

5.2 Kurva *learning* pada `Scikit-Learn`

`Scikit-Learn` menawarkan utilitas yang sangat baik untuk membuat kurva *learning* dari sebuah model. Pada bagian ini kita akan menghitung kurva *learning* dari dataset dengan polinomial orde 2 dan orde 9.

[60]:
```
from sklearn.model_selection import learning_curve

fig, ax = plt.subplots(1, 2, figsize=(16, 6))
fig.subplots_adjust(left=0.0625, right=0.95, wspace=0.1)

for i, degree in enumerate([2, 9]):
    N, train_lc, val_lc = learning_curve(PolynomialRegression(degree),
                                         X, y, cv=7,
                                         train_sizes=np.linspace(0.3, 1, 25))

    ax[i].plot(N, np.mean(train_lc, 1), color='blue', label='training score')
```

```
    ax[i].plot(N, np.mean(val_lc, 1), color='red', label='validation score')
    ax[i].hlines(np.mean([train_lc[-1], val_lc[-1]]), N[0], N[-1],
                 color='gray', linestyle='dashed')

    ax[i].set_ylim(0, 1)
    ax[i].set_xlim(N[0], N[-1])
    ax[i].set_xlabel('training size')
    ax[i].set_ylabel('score')
    ax[i].set_title('degree = {0}'.format(degree), size=14)
    ax[i].legend(loc='best')
```



Metode ini merupakan metode diagnostik yang sangat berharga, karena akan memberikan gambaran visual bagaimana model merespon jika data training diperbanyak. Khususnya, ketika kurva *learning* telah konvergen (yaitu, ketika kurva training dan validasi sudah mendekat satu sama lain), maka penambahan jumlah data training tidak akan memperbaiki kinerja secara signifikan. Situasi ini terlihat pada gambar sebelah kiri, yang merupakan kurva *learning* untuk model polinomial orde 2.

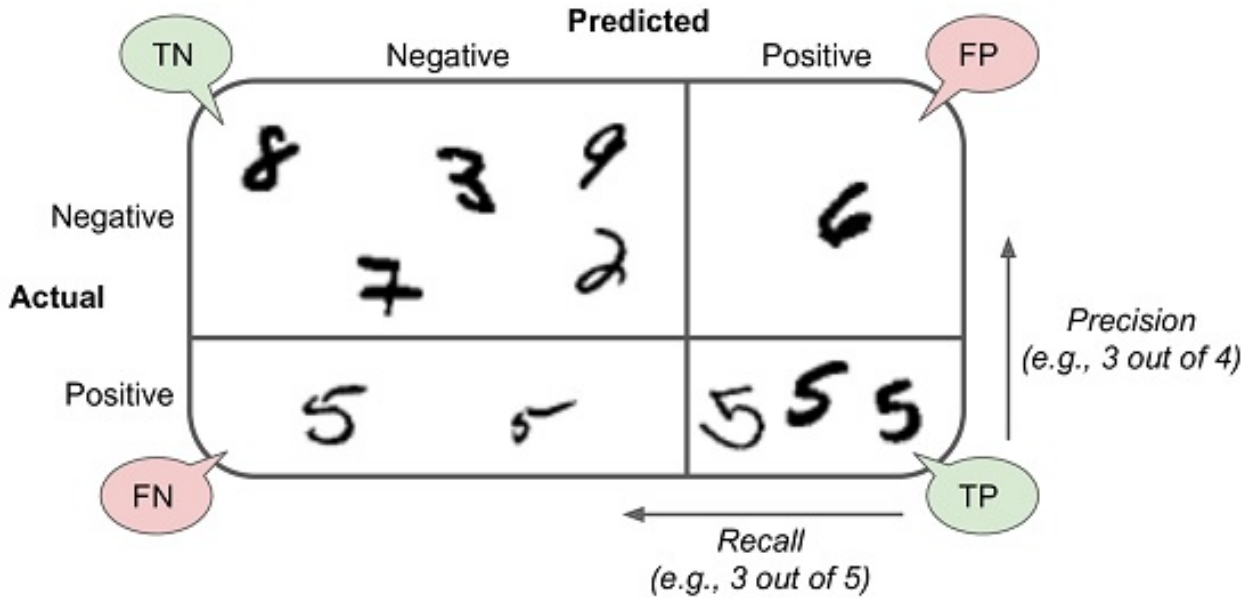
Satu-satunya cara untuk meningkatkan skor konvergensi adalah menggunakan model yang berbeda dan bisa jadi lebih kompleks. Kita lihat masalah ini pada gambar sebelah kanan, dimana ketika kita menggunakan model yang lebih kompleks, skor konvergensi akan membaik (ditunjukkan oleh garis abu putus-putus), tetapi harus dibayar dengan hasil variansi model yang lebih tinggi (diindikasikan dengan perbedaan antara skor training dan skor validasi yang lebih besar). Jika kita terus menambahkan data training lebih banyak, maka kurva *learning* untuk model yang lebih kompleks pada akhirnya akan konvergen. Melakukan *plotting* kurva *learning* pada model dan dataset yang kita pilih dapat menolong untuk membuat keputusan menyangkut bagaimana cara untuk melanjutkan sehingga menghasilkan analisa yang lebih baik.

6 Ukuran Kinerja (*Performance Measure*)

Section ini akan menjelaskan beberapa parameter kinerja yang biasanya digunakan untuk klasifikasi. Diantaranya adalah *confusion matrix*, *precision* dan *recall*, serta kurva *receiver operating characteristics* (ROC).

6.1 *Confusion Matrix*

Cara yang paling baik untuk mengukur kinerja sebuah *classifier* adalah dengan melihat ***confussion matrix***. Idenya adalah menghitung berapa kali sebuah contoh (*instances*) dari *class* A diklasifikasikan sebagai *class* yang lain, misalkan *class* B, begitu juga sebaliknya. Contoh pada kasus klasifikasi tulisan tangan, untuk mengetahui berapa kali *classifier* bingung (*confused*) gambar dari angka 5 menjadi diputuskan angka 3, kita dapat melihat pada baris ke-5 dan kolum ke-3 dari *confussion matrix*. Ilustrasi *confussion matrix* untuk klasifikasi biner (*binary classificaion*) ditunjukan pada gambar 3.9.



Gambar 3.9: Ilustrasi Confussion Matrix menunjukan contoh dari true negatives (TN), false positives (FP), false negatives (FN) dan true positives (TP)

Dalam menentukan *confussion matrix* maka diperlukan satu set hasil prediksi sehingga dapat dibandingkan dengan target aktual. Untuk mendapatkan hasil prediksi dengan *K-fold cross validation* kita bisa gunakan fungsi `cross_val_predict()`. Kode berikut akan melakukan pemisahan training set menjadi 3 subset berbeda secara random yang disebut dengan *fold*, menggunakan *3-fold cross validation* (cv =3). Data tersebut akan digunakan untuk melakukan evaluasi prediksi dengan sebuah *fold* berbeda sebanyak tiga kali, dimana masing-masing setelah dilakukan training oleh 2 *fold* yang lain. Detail untuk `cross_val_predict()` pada `model_selection` bisa dilihat di cross validation prediction.

[61]:
```
from sklearn.model_selection import cross_val_predict

y_train_pred = cross_val_predict(sgd_clf, X_train, y_train_5, cv=3)
```

Variabel `y_train_pred` menyimpan hasil prediksi berdasarkan 3 subset di atas, dan bisa digunakan untuk menentukan *confussion matrix* menggunakan fungsi `confusion_matrix()` dengan argumen target *class* masing-masing (`y_train_5`) dan *class* hasi prediksi (`y_train_pred`).

[62]:
```
from sklearn.metrics import confusion_matrix
confusion_matrix(y_train_5, y_train_pred)
```

[62]:
```
array([[53892,   687],
       [ 1891,  3530]])
```

Setiap baris pada *confusion matrix* menyatakan *actual class*, sementara setiap kolom menyatakan *predicted class*. Pada contoh di atas, baris pertama menyatakan jumlah gambar yang bukan angka 5 (*non-5 images*), atau *negative class*, sehingga baris pertama kolom pertama menyatakan banyaknya gambar yang bukan angka 5 diklasifikasikan betul sebagai bukan angka 5 (*true negatives*), sedangkan baris pertama kolom kedua menyatakan jumlah yang diklasifikasikan salah sebagai angka 5 (*false positives*). Baris kedua menyatakan jumlah gambar angka 5, atau *positive class*, sehingga baris kedua kolom pertama menyatakan banyaknya gambar angka 5 yang diklasifikasikan salah sebagai bukan angka 5 (*false negatives*), sementara baris kedua kolom kedua menyatakan jumlah yang diklasifikasikan betul sebagai angka 5 (*true positives*). *Classifier* yang sempurna hanya akan mengklasifikasikan secara sempurna tanap salah, sehingga hanya akan mempunyai *true positives* dan *true negatives*, atau hanya terdapat harga tidak nol (*non-zero*) pada diagonal-nya saja (atas kiri ke bawah kanan).

6.2 *Precision* dan *Recall*

Confusion matrix memberikan banyak informasi, tetapi kadang-kadang kita lebih membutuhkan ukuran yang lebih ringkas atau sederhana. Salah satu yang sering digunakan adalah keakuratan dari prediksi positif (*positive prediction*), yang disebut dengan *Precision* pada persamaan (3.1).

Persamaan (3.1). *Precision*

$$\text{precision} = \frac{TP}{TP + FP}$$

dimana TP menyatakan jumlah *true positives* dan FP menyatakan jumlah *false positives*.

Pengukuran kinerja dengan menggunakan metrik *precision* kadang tidak cukup karena bisa mengakibatkan salah pengertian. Sebagai contoh, kita bisa mendapatkan *precision* 100% dengan hanya dengan melakukan prediksi benar dengan satu sampel benar, dengan kata lain $TP = 1$ dan $FP = 0$. Oleh sebab itu metrik *precision* sering disandingkan dengan metrik lain yaitu *recall* yang menyatakan sensitivitas atau disebut juga *true positive rate* (TPR). *Recall* merupakan rasio antara *positive instances* yang dikategorikan sebagai benar oleh *classifier*, yang ditunjukan dengan persamaan (3.2). Kita dapat melihat Gambar 3.9 untuk lebih memahami pengertian *precision* dan *recall*.

Persamaan (3.2). *Recall*

$$\text{recall} = \frac{TP}{TP + FN}$$

dimana FN menyatakan jumlah *false positives*.

Perhitungan metrik *precision* dan *recall* dapat dilakukan pada `Scikit-Learn`, seperti kode di bawah.

[63]:
```
from sklearn.metrics import precision_score, recall_score
precision_score(y_train_5, y_train_pred)
```

[63]:
```
0.8370879772350012
```

Hasil diatas menunjukan metrik *precision* dalam desimal (atau dapat juga dalam prosentase jika dikalikan 100%). Sedangkan untuk memperoleh nilai *recall* dengan program berikut.

[64]:
```
recall_score(y_train_5, y_train_pred)
```

[64]:
```
0.6511713705958311
```

> Coba anda cocokan hasil perhitungan *precision* dan *recall* menggunakan angka pada *confussion matrix* hasil eksekusi `confusion_matrix(y_train_5, y_train_pred)` pada program di atas. Apakah cocok?

Metrik lain yang sering digunakan adalah F_1 *score*, dimana metrik ini menggabungkan metrik *precision* dan *recall* menjadi satu angka. F_1 *score* khususnya digunakan untuk membandingkan dua *classifier* secara sederhana. F_1 *score* merupakan rata-rata harmonik (*harmonic mean*) dari *precision* dan *recall*, seperti terlihat pada persamaan (3.3). Rata-rata bisa memberikan bobot sama untuk semua harga, sedangkan rata-rata harmonik akan memberikan bobot yang lebih tinggi untuk harga yang kecil. Sehingga, sebuah *classifier* hanya akan memperoleh nilai F_1 *score* yang tinggi jika *precision* dan *recall* keduanya bernilai besar.

Persamaan (3.3). *Recall*

$$F_1 = \frac{2}{\frac{1}{\text{precision}} + \frac{1}{\text{recall}}} = 2 \times \frac{\text{precision} \times \text{recall}}{\text{precision} + \text{recall}} = \frac{TP}{TP + \frac{FN+FP}{2}}$$

Untuk menghitung F_1 *score* pada `Scikit-Learn` maka bisa dengan cara memanggil fungsi `f1_score()`.

[65]:
```
from sklearn.metrics import f1_score
f1_score(y_train_5, y_train_pred)
```

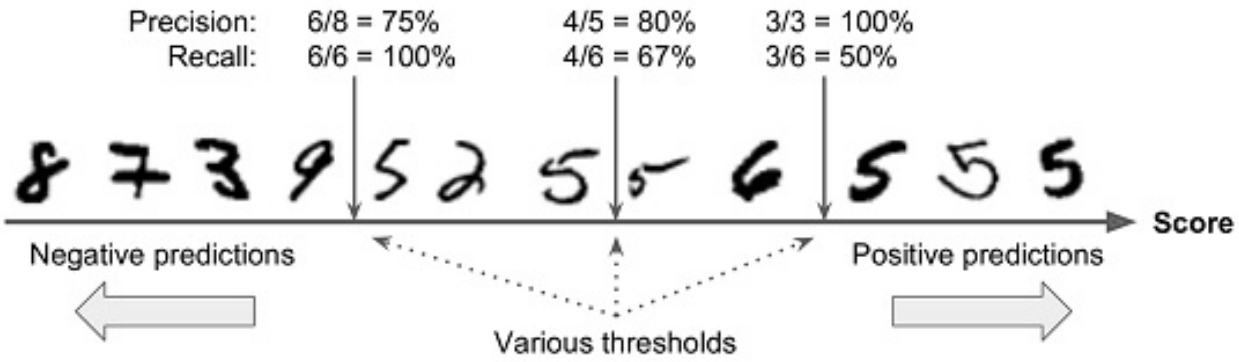
[65]:
```
0.7325171197343846
```

F_1 *score* lebih cenderung berpihak pada *classifier* dengan *precision* dan *recall* yang mirip dan tidak jauh berbeda nilainya. Pada konteks tertentu kita cenderung lebih memperhatikan *precision*, dan pada konteks lain kita cenderung lebih memperhatikan *recall*. Sebagai contoh, jika kita melakukan training untuk mendeteksi video yang aman untuk anak-anak kecil, maka kemungkinan kita lebih cenderung memilih *classifier* yang menolak (*reject*) *good videos* (*low recall*) tetapi hanya menyimpan yang betul-betul aman (*high precision*), dibandingkan dengan *classifier* yang mempunyai *high recall* tetapi membiarkan beberapa video yang tidak aman diklasifikasikan sebagai aman bagi anak-anak. Tetapi dalam kasus lain, misalkan *classifier* untuk mendeteksi penjarah menggunakan gambar dari kamera *surveillence*, maka kemungkinan tidak jadi masalah jika *precision* hanya 30% selama *recall* mempunyai rate 99% (artinya, *security* akan mendapatkan banyak *alert* pencurian yang salah, tetapi hampir semua penjarah akan terdeteksi dengan benar.

> **Catatan**. Kita tidak bisa memperbesar *recall* dan *precision* secara berbarengan, selalu ada *trade-off*. Ketika memperbesar *recall* maka secara bersamaan memperkecil *precision* dan sebaliknya.

6.3 *Trade-off* antara *Precision* dan *Recall*

Untuk memahami *trade-off*, kita akan melihat bagaimana `SGDClassifier` membuat keputusan klasifikasinya. Sebagai contoh, *classifier* menghitung *score* berdasarkan *decision function*. Jika *score* tersebut lebih besar dari sebuah *threshold*, maka akan memberikan keputusan sebagai *positive class*, dan sebaliknya sebagai *negative class*. Gambar 3.10 menunjukan beberapa dijit yang mempunyai nilai *score* terendah sebelah kiri dan tertinggi di sebelah kanan.



Gambar 3.10: Pada trade-off precision dan recall ini gambar disusun berdasarkan *score* klasifikasinya, dimana yang lebih besar dari threshold akan diklasfikasikan sebagai positive. Semakin besar threshold, semakin rendah recall, tapi (secara umum) precision lebih tinggi

Gambar 3.10 menunjukan jika *threshold* keputusan berada pada posisi panah di tengah (antara dua angka 5), maka kita akan menemukan bahwa 4 *true positive* (aktual berangka 5) pada sebelah kanan *threshold* tersebut, dan hanya 1 *false positive* (aktual berangka 6). Sehingga dengan *threshold* tersebut maka dihasilkan *precision* 80% (4 dari 5). Tetapi dari 6 buah yang berangka 5, hanya dapat dideteksi 4 saja, sehingga *recall* menjadi 67% (4 dari 6). Jika kita menaikan nilai *threshold* ke sebelah kanan sehingga pada posisi panah terkanan, maka akan diperoleh kenaikan *precision* menjadi 100% (3 dari 3 berangka 5). Tetapi sebuah *true poisive* berangka 5 akan menjadi sebuah *false negative* sehingga *recall* menurun menjadi 50% (3 dari 6 berangka 5). Demikian pula sebaliknya jika nilai *threshold* diperkecil sehingga berada pada posisi panah terkiri,maka *recall* bertambah dan *precision* berkurang.

`Scikit-Learn` tidak memperkenankan kita untuk melakukan setting *threshold* secara langsung, tetai memberikan kita akses ke *decision score* yang digunakan untuk membuat prediksi. Kita dapat memanggil fungsi `decision_function()` yang mengembalikan luaran berupa *score* dari setiap *instance*, kemudian bisa dipilih *threshold* untuk membuat prediksi berdasarkan *score-score* tersebut

[66]:
```
y_scores = sgd_clf.decision_function([some_digit])
y_scores
```

[66]:
```
array([2164.22030239])
```

[67]:
```
threshold = 0
y_some_digit_pred = (y_scores > threshold)
```

[68]:
```
y_some_digit_pred
```

[68]:
```
array([ True])
```

[69]:
```
threshold = 8000
y_some_digit_pred = (y_scores > threshold)
y_some_digit_pred
```

[69]:
```
array([False])
```

Hal di atas memberikan konfirmasi bahwa menaikan nilai *threshold* mengurangi *recall*. Gambar sebetulnya merepresentasikan angka 5, dan *classifier* mendeteksinya ketika *threshold* sama dengan nol, tetapi membuat klasifikasi salah ketika *threshold* dinaikan ke 8000.

Kemudian pertanyaannya adalah bagaimana menentukan *threshold* yang akan dipakai? Maka dapat digunakan fungsi `cross_val_predict()` untuk mendapatkan *score* dari semua *instance* pada training set, tetapi kita menspesifikasikan untuk memperoleh *decision score* bukan hasil prediksi.

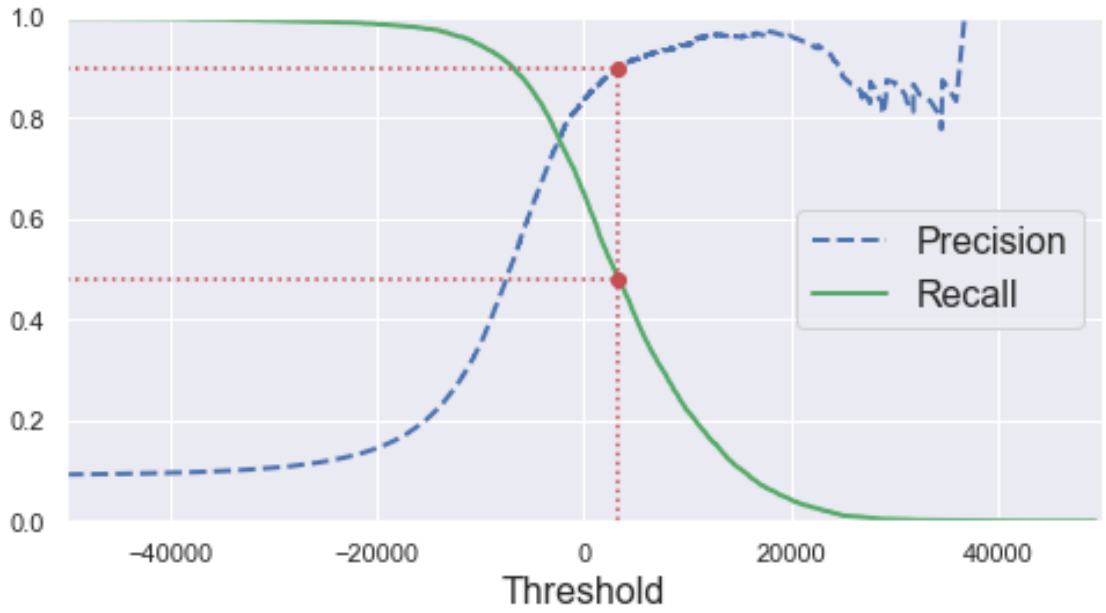
[70]:
```
y_scores = cross_val_predict(sgd_clf, X_train, y_train_5, cv=3,
                             method="decision_function")
```

Dengan *score* ini kita bisa menghitung *precision* dan *recall* untuk semua kemungkinan threshold dengan menggunakan fungsi `precision_recall_curve()`.

[71]:
```
from sklearn.metrics import precision_recall_curve
precisions, recalls, thresholds = precision_recall_curve(y_train_5, y_scores)
```

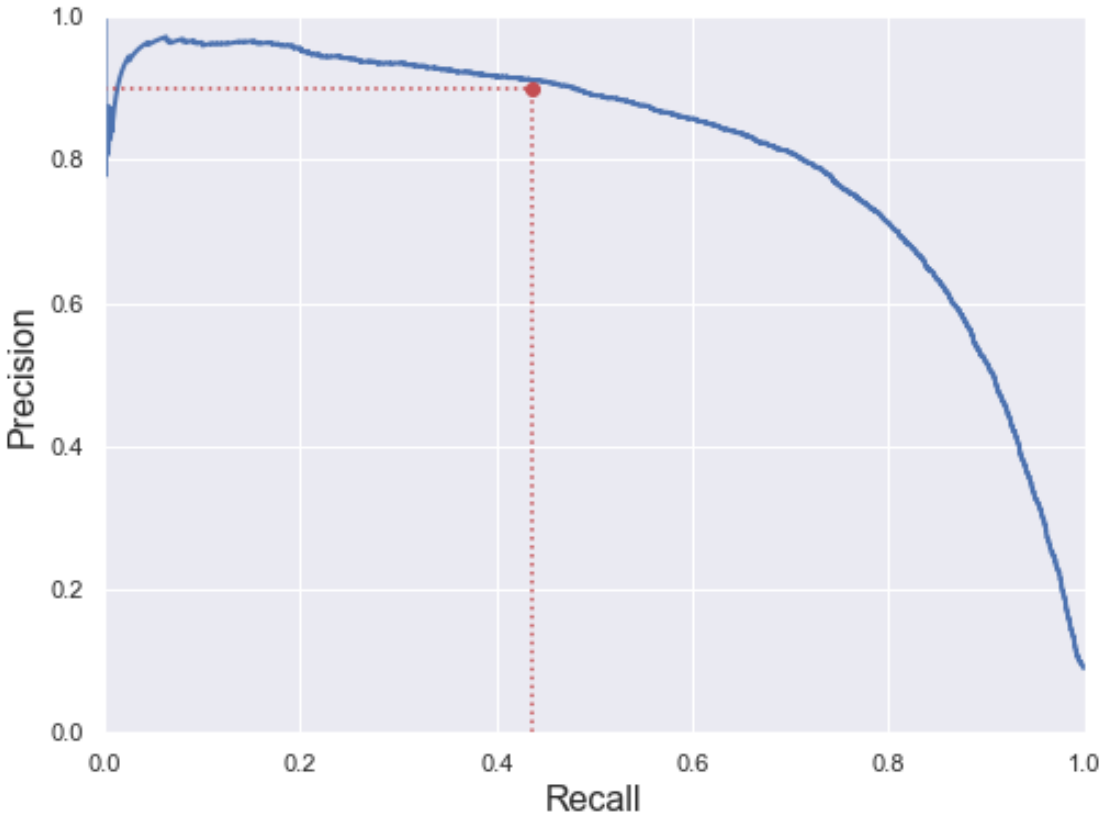
Kemudian bisa digunakan `matplotlib` untuk memploting *precision* dan *recall* sebagai fungsi nilai-nilai *threshold*.

[72]:
```
def plot_precision_recall_vs_threshold(precisions, recalls, thresholds):
    plt.plot(thresholds, precisions[:-1], "b--", label="Precision", linewidth=2)
    plt.plot(thresholds, recalls[:-1], "g-", label="Recall", linewidth=2)
    plt.legend(loc="center right", fontsize=16)
    plt.xlabel("Threshold", fontsize=16)
    plt.grid(True)
    plt.axis([-50000, 50000, 0, 1])

recall_90_precision = recalls[np.argmax(precisions >= 0.90)]
threshold_90_precision = thresholds[np.argmax(precisions >= 0.90)]

plt.figure(figsize=(8, 4))
plot_precision_recall_vs_threshold(precisions, recalls, thresholds)
plt.plot([threshold_90_precision, threshold_90_precision], [0., 0.9], "r:")
plt.plot([-50000, threshold_90_precision], [0.9, 0.9], "r:")
plt.plot([-50000, threshold_90_precision], [recall_90_precision,␣
 ↪recall_90_precision], "r:")
plt.plot([threshold_90_precision], [0.9], "ro")
plt.plot([threshold_90_precision], [recall_90_precision], "ro")
plt.show()
```



Cara lain untuk memilih *trade-off precision/recall* yang baik adalah dengan melakukan plotting *precision* sebagai fungsi *recall* secara langung.

[73]:
```
def plot_precision_vs_recall(precisions, recalls):
    plt.plot(recalls, precisions, "b-", linewidth=2)
    plt.xlabel("Recall", fontsize=16)
    plt.ylabel("Precision", fontsize=16)
    plt.axis([0, 1, 0, 1])
    plt.grid(True)

plt.figure(figsize=(8, 6))
plot_precision_vs_recall(precisions, recalls)
plt.plot([0.4368, 0.4368], [0., 0.9], "r:")
plt.plot([0.0, 0.4368], [0.9, 0.9], "r:")
plt.plot([0.4368], [0.9], "ro")
plt.show()
```



Dengan menggunakan kurva *precision* sebagai fungsi *threshold* di atas, jika diinginkan *precision* 90% maka akan diperoleh *threshold di sekitar 8000, atau bisa digunakan fungsi `np.argmax()` yang menghasilkan indeks pertama dari harga maksimum.

[74]:
```
threshold_90_precision = thresholds[np.argmax(precisions >= 0.90)]
```

[75]:
```
threshold_90_precision
```

[75]:
```
3370.0194991439557
```

Untuk melakukan prediksi (pada training set), maka dapat digunakan kode berikut sebagi pengganti fungsi `predict()` seperti sebelumnya.

[76]:
```
y_train_pred_90 = (y_scores >= threshold_90_precision)
```

Hasil *precision* dan *recall* dari prediksi ini adalah sebagai berikut.

[77]:
```
precision_score(y_train_5, y_train_pred_90)
```

[77]:
```
0.9000345901072293
```

[78]:
```
recall_score(y_train_5, y_train_pred_90)
```

[78]:
```
0.4799852425751706
```

Terlihat kita sudah memperoleh nilai *precision* 90% seperti yang kita inginkan. Sangat mudah untuk mendapatkan *precision* cukup besar, yaitu dengan melakukan setting *threshold* cukup tinggi. Tetapi hati-hati, nilai *recall* akan menjadi sangat kecil, dan *classifier* kita menjadi tidak banyak berarti.

> Jika seseorang berkata kepada anda, "Ayo buat *classifier* dengan *precision* 99%," maka pertanyaan yang paling tepat anda lontarkan adalah, "Pada *recall* berapa?"

6.4 *Receiver Operating Characterisitic* (ROC)

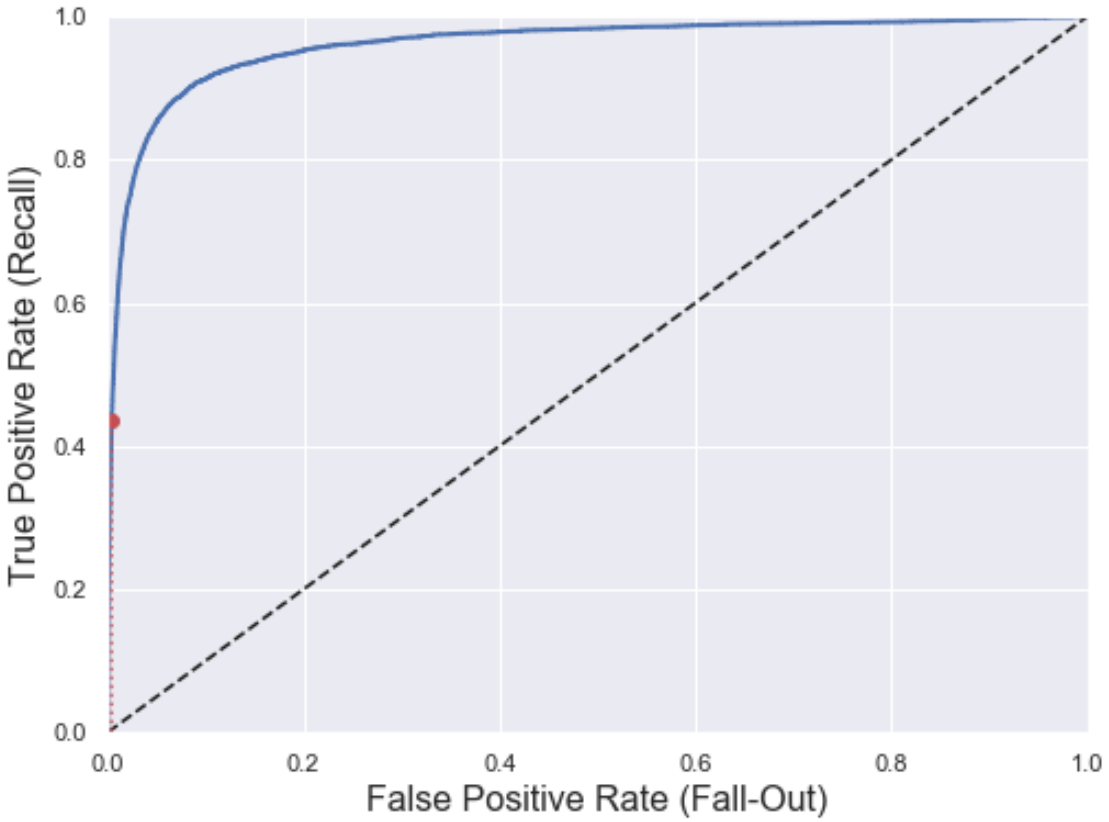
Kurva ROC adalah cara lain sering digunakan untuk mengukur kinerja *binary classifier*. Mirip dengan kurva *precision/recall*, tetapi pada ROC kurva menunjukan hubungan antara *true positive rate* (TPR), yang merupakan nama lain *recall*, terhadap *false positive rate* (FPR). FPR adalah rasio dari *negative instance* yang diklasifikasikan salah sebagai *positive*. FPR sama dengan 1-*true negative rate* (TNR) yang merupakan rasio dari *negative instance* yang diklasifikasikan benar sebagai *negative*. TNR disebut juga sebagai *specificity*. Sehingga ROC adalah kurva yang menggambarkan hubungan antara *recall* dengan 1-*specificity*.

Kita gunakan fungsi `roc_curve()` untuk menghitung TPR dan FPR pada harga-harga *threshold* beragam, sebelum bisa plotting kurva ROC.

[79]:
```
from sklearn.metrics import roc_curve
fpr, tpr, thresholds = roc_curve(y_train_5, y_scores)
```

Kemudian kita bisa plotting FPR terhadap TPR dengan `matplotlib`.

[80]:
```
def plot_roc_curve(fpr, tpr, label=None):
    plt.plot(fpr, tpr, linewidth=2, label=label)
    plt.plot([0, 1], [0, 1], 'k--') # dashed diagonal
    plt.axis([0, 1, 0, 1])
    plt.xlabel('False Positive Rate (Fall-Out)', fontsize=16)
    plt.ylabel('True Positive Rate (Recall)', fontsize=16)
    plt.grid(True)

plt.figure(figsize=(8, 6))
plot_roc_curve(fpr, tpr)
plt.plot([4.837e-3, 4.837e-3], [0., 0.4368], "r:")
plt.plot([0.0, 4.837e-3], [0.4368, 0.4368], "r:")
plt.plot([4.837e-3], [0.4368], "ro")
plt.show()
```



Jika kita lihat kurva ROC maka terdapat *trade-off* juga, semakin besar TPR maka semakin besar juga FPR. Garis titik-titik menunjukan kurva ROC dari *classifier* yang betul-betul random (*purely random classifier*), seperti melemparkan koin. *Classifier* yang baik akan mempunyai kurva ROC yang menjauh dari garis titik-titik tersebut sebisa mungkin (menuju sudut kiri) seperti pada kurva bergaris biru.

Cara untuk membandingkan kinerja *classifier* adalah menghitung area dibawah kurva, *area under the curve* (AUC). *Classifier* yang ideal (*perfect*) akan mempunyai ROC-AUC sama dengan 1 (1×1), sedangkan *purely random classifier* mempunyai ROC-AUC $0,5$ ($0,5 \times 1 \times 1$). `Scikit-Learn` menyediakan fungsi untuk menghitung ROC-AUC.

[81]:
```
from sklearn.metrics import roc_auc_score
roc_auc_score(y_train_5, y_scores)
```

[81]:
```
0.9604938554008616
```

Karena kurva ROC mirip dengan kurva *precision/recall* (PR), mana yang sebaiknya digunakan pada kasus yang sedang ditangani? Sebagai *Rule of Thumb*, kita gunakan kurva PR kalau *positive class* jumlahnya jarang atau kita lebih *concern* terhadap *false positive* dibandingkan dengan *false negative*. Sebaliknya, kita gunakan kurva ROC. Pada kasus sebelumnya, karena jumlah *positive class* (gambar angka 5) lebih sedikit dibandingkan dengan *negative class* (gambar bukan angka 5), terlihat pada kurva ROC

kinerja dari *classifier* bisa dikatakan sangat baik. Tetapi kalau kita lahit kurva PR maka kita simpulkan masih banyak ruang untuk memperbaiki kinerja *classifier* untuk kasus tersebut.

7 *Multiclass Classifier*

Jika *binary classifier* dapat membedakan antara dua *class*, *multiclass classifier* atau disebut juga *multinomial classifier* dapat membedakan lebih dari dua *class*. Beberapa algoritma diantaranya *SGD classifier*, *Random Forest classifier*, dan *Naive Bayes classifier* dapat digunakan secara langsung untuk melakukan *multiclass classifier*. Sedangkan algoritma lain, misalkan *Logistic regression* dan *Support Vector Machine*, pada dasarnya merupakan *binary classifier*. Tetapi, ada beberapa strategi yang dapat dipakai untuk melakukan *multiclass classification* dengan mengkombinasikan beberapa *binary classifier*.

Salah satu cara untuk membuat sistem dapat mengklasifikasikan gambar-gambar dijit kedalam 10 *class* (dijit 0 sampai 9) adalah dengan melakukan training 10 *binary classifier*, satu untuk setiap dijit (sebuah detektor untuk angka 0, detektor untuk angka 1, detektor untuk angka 2, dsb). Kemudian ketika kita ingin mengklasifikasikan gambar, kita akan mendapatkan *decision score* untuk setiap *classifier* tersebut dan memilih *class* hasil *classifier* dengan *score* tertinggi. Cara ini disebut dengan strategi *one-versus-the-rest* (OvR) atau disebut juga *one-versus-all*.

Strategi lain adalah melatih *binary classifier* untuk setiap pasangan dijit. Satu *classifier* untuk membedakan 0s dan 1s, yang lain untuk membedakan 0s dan 2s, yang lain untuk 1s dan 2s, dan seterusnya. Cara ini disebut dengan strategi *one-versus-one* (OvO). Jika terdapat N *class* maka dibutuhkan sebanyak $N \times (N-1)/2$ *classifier*. Untuk dataset MNIST, untuk membedakan 10 *class*, maka dibutuhkan 45 *classifier* dan diambil *class* yang paling banyak diputuskan dari 45 *binary classifier* tersebut. Keuntungan utama dari OvO adalah setiap *classifier* hanya dilatih dengan sebagian data dari training set untuk masing-masing pasangan *class* yang harus dibedakan.

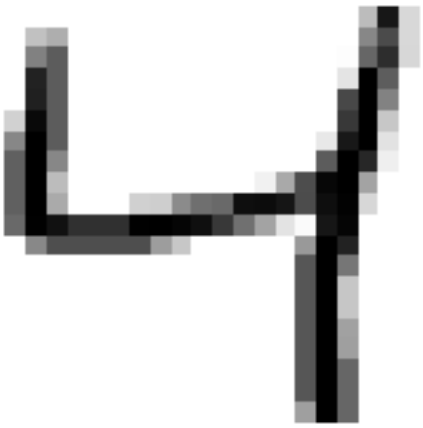
Beberapa algoritma misalkan SVM jika di-*scaling* dengan ukuran training set tidak akan mempunyai kinerja yang baik. Untuk algoritma-algoritma semacam ini maka strategi OvO lebih baik untuk dipilih karena akan lebih cepat melatih banyak *classifier* dengan training set yang sedikit dibandingkan melatih sedikit *classifier* dengan ukuran training set yang besar. Untuk sebagian besar algoritma *binary classifier* lebih baik menggunakan strategi OvR.

`Scikit-Learn` akan mendeteksi jika digunakan *binary classifier* untuk pekerjaan *multiclass classification*, dan secara otomatis akan memilih strategi OvR atau OvO, tergantung pada algoritma yang digunakan. Sebagai contoh, kita akan menggunakan SVM yang pada dasarnya untuk *binary classification*, tetapi untuk melakukan *multiclass classification* dengan *class* `sklearn.svm.SVC`.

[82]:
```
from sklearn.datasets import fetch_openml
mnist = fetch_openml('mnist_784', version=1)
X, y = mnist["data"], mnist["target"]
X_train, X_test, y_train, y_test = X[:60000], X[60000:], y[:60000], y[60000:]
```

[83]:
```
%matplotlib inline


import numpy as np
```

```
import matplotlib as mpl
import matplotlib.pyplot as plt

some_digit = X[2]
some_digit_image = some_digit.reshape(28, 28)
plt.imshow(some_digit_image, cmap=mpl.cm.binary)
plt.axis("off")
plt.show()
```



[84]:
```
from sklearn.svm import SVC

svm_clf = SVC(gamma="auto", random_state=42)
svm_clf.fit(X_train[:1000], y_train[:1000]) # y_train, bukan y_train_5
svm_clf.predict([some_digit])
```

[84]:
```
array(['4'], dtype=object)
```

Kode di atas melatih SVC dengan training set menggunakan *class* target dari 0 s/d 9 (`y_train`), kemudian SVC digunakan untuk melakukan *multiclass classification* untuk gambar dari angka tersebut. Di dalamnya `Scikit-Learng` menggunakan OvO strategi meskipun tidak tampak, yaitu dengan cara melatih 45 *binary classifier*, kemudian diperoleh *decision score* dari masing-masing *classfier* dan dipilih *class* yang paling banyak diputuskan dari keseluruhan *binary classifier*.

Jika kita memanggil fungsi `decision_function()` maka akan diperoleh 10 *score* tiap *instance* yang merupakan masing-masing *score* tiap *class*.

[85]:
```
some_digit_scores = svm_clf.decision_function([some_digit])
some_digit_scores
```

[85]:
```
array([[ 3.82111996,  7.09167958,  4.83444983,  1.79943469,  9.29932174,
         0.79485736,  2.80437474,  8.10392157, -0.22417259,  5.84182891]])
```

Terlihat *score* tertinggi merupakan *class* 4.

[86]:
```
np.argmax(some_digit_scores)
```

[86]:
```
4
```

Jika kita ingin memaksa `Scikit-Learn` menggunakan *one-versus-one* atau *one-versus-the-rest* maka kita bisa menggunakan `OneVsOneClassifier` atau `OneVsRestClassifier`, yaitu dengan cara membuat sebuah *instance* dan melewatkan *classifier* pada konstruktornya. Sebagai contoh, SVM digunakan untuk *multiclass classifier* menggunakan strategi OvR, berdasarkan SVC.

[87]:
```
from sklearn.multiclass import OneVsRestClassifier
ovr_clf = OneVsRestClassifier(SVC(gamma="auto", random_state=42))
ovr_clf.fit(X_train[:1000], y_train[:1000])
ovr_clf.predict([some_digit])
```

[87]:
```
array(['4'], dtype='<U1')
```

Berbeda dengan SVM, `SGDClassifier` atau `RandomForestClassifier` dapat digunakan secara langsung untuk melakukan *multiclass classifier* tanpa menggunakan strategi OvR maupun OvO, sehingga lebih mudah untuk melakukan training.

[88]:
```
from sklearn.linear_model import SGDClassifier
sgd_clf = SGDClassifier(max_iter=1000, tol=1e-3, random_state=42)
sgd_clf.fit(X_train, y_train)
sgd_clf.predict([some_digit])
```

[88]:
```
array(['4'], dtype='<U1')
```

Fungsi `decision_function()` menghasilkan satu harga tiap *class*.

[89]:
```
sgd_clf.decision_function([some_digit])
```

[89]:
```
array([[-34143.40703505, -21942.13780869,  -4018.29275037,
         -2239.19313075,     43.09419826, -15058.88052383,
        -33653.31059893,  -8277.80610963,  -7460.52016321,
        -14180.15338984]])
```

Dari hasil di atas kita bisa melihat bahwa *classifier* sangat *confident* untuk melakukan klasifikasi, dimana hanya *score* untuk angka 4 yang mempunyai nilai positif, sedangkan *score* untuk *class* lain bernilai negatif.