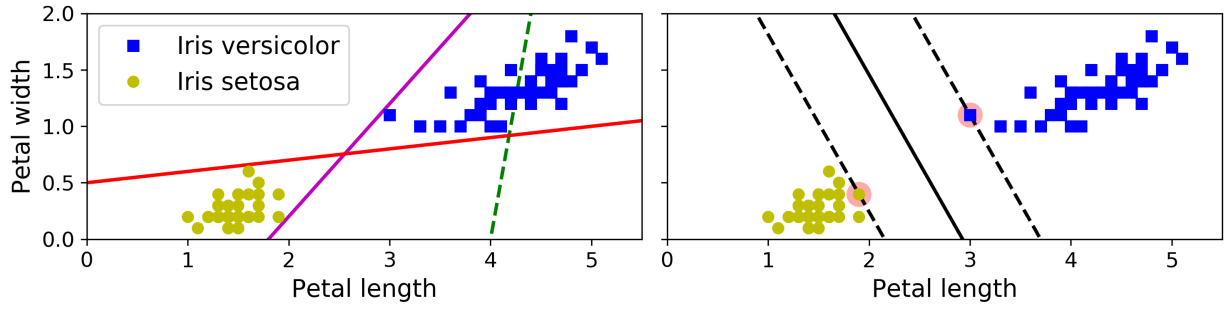
Chapter 5:
Support Vector Machine (SVM)

October 31, 2020

Support Vector Machine (SVM) merupakan model *machine learning* yang sangat powerful dan juga beragam, mampu melakukan klasifikasi linier atau non linier, regresi, dan bahkan deteksi pencilan (*outlier detection*). SVM merupakan model yang paling populer, sehingga untuk orang-orang yang tertarik pada *machine learning* SVM merupakan algoritma yang harus diketahui. SVM khususnya cocok untuk klasfikasi dataset kompleks dengan ukuran kecil atau menengah. Pada bagian ini akan dijelaskan konsep dari SVM, bagaimana menggunakannya dan bagaimana SVM bekerja.

1 Klasifikasi SVM Linier

Ide dasar dari SVM dapat dijelaskan dengan Gambar 5.1 Gambar tersebut menunjukan sebagian dataset iris yang telah kita gunakan di bab-bab sebelumnya. Dua *class* dapat dipisahkan dengan mudah menggunakan garis lurus (*linearly separable*). Gambar sebelah kiri menunjukan batas keputusan (*decision boundary*) dari 3 *classifier* linier yang mungkin. Batas keputusan yang direpresentasikan dengan garis putus-putus mempunyai performansi sangat buruk, bahkan tidak bisa memisahkan kedua *class* dengan baik. Dua model yang lain dapat bekerja dengan baik untuk training set ini, tetapi batas keputusan tersebut sangat dekat dengan *instance-instance* dari data. Sehingga dapat disimpulkan jika terdapat data baru, model-model ini tidak akan memiliki kinerja yang baik. Sebaliknya, garis tebal pada gambar sebelah kanan merupakan batas keputusan dari *classifier* SVM, dimana garis tersebut bukan hanya memisahkan kedua *class* tetapi berjarak cukup jauh dari training *instances* yang terdekat. Kita dapat melihat *classifer* SVM sebagai cara untuk *fitting* lebar jalan terbesar yang memisahkan kedua *class* yang ada, seperti yang ditunjukan pada gambar sebelah kanan. Metode ini disebut dengan *large margin classification*.



Gambar 5.1. Ilustrasi klasifikasi dengan margin yang besar

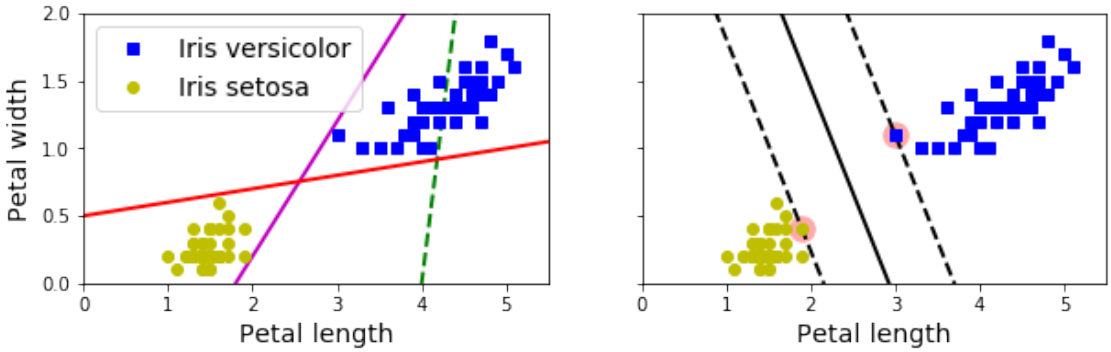
Berikut kode program yang menghasilkan Gambar 5.1.

[1]:
```
from sklearn.svm import SVC
from sklearn import datasets

iris = datasets.load_iris()
X = iris["data"][:, (2, 3)]  # petal length, petal width
y = iris["target"]

setosa_or_versicolor = (y == 0) | (y == 1)
X = X[setosa_or_versicolor]
y = y[setosa_or_versicolor]

# SVM Classifier model
svm_clf = SVC(kernel="linear", C=float("inf"))
svm_clf.fit(X, y)
```

[1]:
```
SVC(C=inf, break_ties=False, cache_size=200, class_weight=None, coef0=0.0,
    decision_function_shape='ovr', degree=3, gamma='scale', kernel='linear',
    max_iter=-1, probability=False, random_state=None, shrinking=True,
    tol=0.001, verbose=False)
```

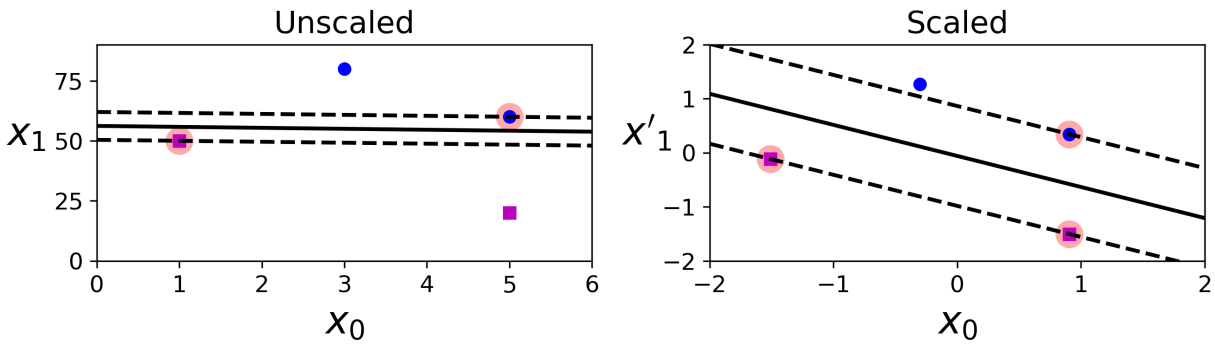
[4]:
```
# Bad models
import numpy as np
import matplotlib.pyplot as plt

x0 = np.linspace(0, 5.5, 200)
pred_1 = 5*x0 - 20
pred_2 = x0 - 1.8
pred_3 = 0.1 * x0 + 0.5

def plot_svc_decision_boundary(svm_clf, xmin, xmax):
    w = svm_clf.coef_[0]
    b = svm_clf.intercept_[0]

    # At the decision boundary, w0*x0 + w1*x1 + b = 0
    # => x1 = -w0/w1 * x0 - b/w1
    x0 = np.linspace(xmin, xmax, 200)
    decision_boundary = -w[0]/w[1] * x0 - b/w[1]

    margin = 1/w[1]
    gutter_up = decision_boundary + margin
    gutter_down = decision_boundary - margin

    svs = svm_clf.support_vectors_
    plt.scatter(svs[:, 0], svs[:, 1], s=180, facecolors='#FFAAAA')
    plt.plot(x0, decision_boundary, "k-", linewidth=2)
```

```
    plt.plot(x0, gutter_up, "k--", linewidth=2)
    plt.plot(x0, gutter_down, "k--", linewidth=2)

fig, axes = plt.subplots(ncols=2, figsize=(10,2.7), sharey=True)

plt.sca(axes[0])
plt.plot(x0, pred_1, "g--", linewidth=2)
plt.plot(x0, pred_2, "m-", linewidth=2)
plt.plot(x0, pred_3, "r-", linewidth=2)
plt.plot(X[:, 0][y==1], X[:, 1][y==1], "bs", label="Iris versicolor")
plt.plot(X[:, 0][y==0], X[:, 1][y==0], "yo", label="Iris setosa")
plt.xlabel("Petal length", fontsize=14)
plt.ylabel("Petal width", fontsize=14)
plt.legend(loc="upper left", fontsize=14)
plt.axis([0, 5.5, 0, 2])

plt.sca(axes[1])
plot_svc_decision_boundary(svm_clf, 0, 5.5)
plt.plot(X[:, 0][y==1], X[:, 1][y==1], "bs")
plt.plot(X[:, 0][y==0], X[:, 1][y==0], "yo")
plt.xlabel("Petal length", fontsize=14)
plt.axis([0, 5.5, 0, 2])
plt.show()
```



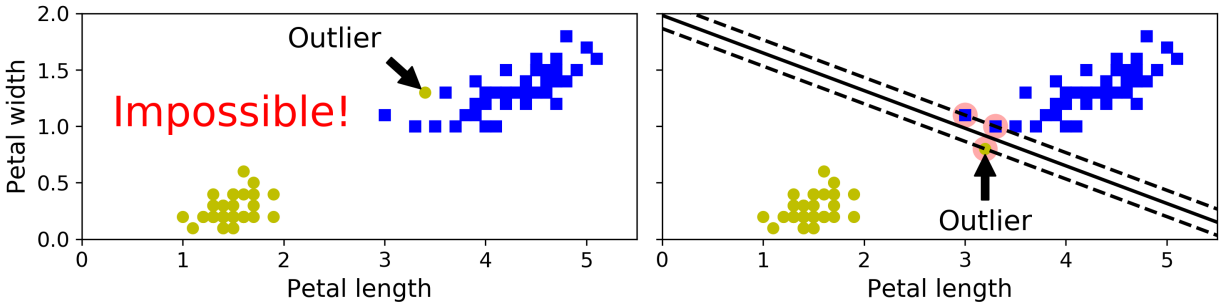
Jika kita perhatikan, apabila kita tambah training *instances* di luar jalan (*off the street*) tidak akan mempengaruhi batas keputusan sama sekali. Batas keputusan sangat ditentukan (didukung) oleh *instances* pada ujung-ujung (*edges*) dari jalan tersebut. *Instances* ini disebut dengan ***support vectors*** (diberi bulatan pada Gambar 5.1.).

SVM sangat sensitif terhadap penskalaan (*scaling*) dari *feature* seperti terlihat pada Gambar 5.2. Pada gambar sebelah kiri, skala vertikal lebih lebar dibandingkan dengan skala horisontal. Sehingga jalan pemisah terlebar lebih cenderung untuk dipilih pada arah horisontal. Setelah penskalaan *feature* (mis. dengan `StandardScaler` pada `Scikit Learn`), batas keputusan pada gambar sebelah kanan akan terlihat lebih baik.



Gambar 5.2. Ilustrasi SVM sensitif terhadap penskalaan

Jika kita membatasi bahwa semua *instances* berada di luar jalan pemisah dan berada pada sisi (*class*) yang benar, maka disebut dengan ***hard margin classification***. Terdapat dua isu utama dengan *hard margin classification*. Pertama, *hard margin classification* hanya berkerja jika data *linearly separable*. Kedua, metode ini sangat sensitif terhadap pencilan data (*outliers*). Gambar 5.3 menunjukan dataset Iris dengan penambahan sebuah data pencilan. Pada gambar sebelah kiri, menunjukan kasus dimana tidak mungkinan untuk mendapatkan *hard margin*. Sedangkan pada gambar sebelah kanan, batas keputusan yang diperoleh ketika ada pencilan berbeda dengan gambar sebelah kanan di Gambar 5.1 ketika tanpa pencilan. Sehingga batas keputusan yang dibuat saat ada pencilan tidak akan menghasilkan generalisasi yang baik.

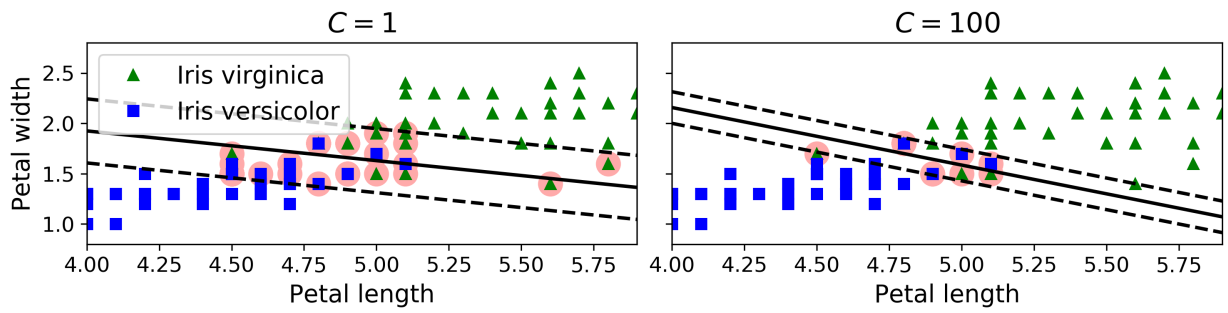


Gambar 5.3. Ilustrasi hard margin sensitif terhadap pencilan

Untuk menghindari isu-isu tersebut, bisa menggunakan model yang lebih fleksibel. Objektifnya adalah menemukan keseimbangan antara membuat jalan pemisah sebesar mungkin, dan membatasi adanya *instances* yang terletak pada tengah-tengah jalan pemisah (*margin violations*) atau bahkan pada sisi yang salah. Cara ini disebut dengan *soft margin classification*.

Ketika membuat model SVM menggunakan `Scikit-Learn`, kita dapat menspesifikasi jumlah *hyperparameter*, misalkan *hyperparameter* `C`. Jika kita setting `C` dengan harga rendah, maka kita akan menghasilkan model seperti pada Gambar 5.4 sebelah kiri, sedangkan untuk harga tinggi kita akan mendapatkan model sebelah kanan. *Margin violations* merupakan hal yang buruk, sehingga akan lebih baik jika tidak banyak jumlahnya. Meskipun demikian, pada kasus ini model pada sebelah kiri mempunyai *margin violations* yang banyak, tetapi menghasilkan generalisasi yang lebih baik.

Catatan. Jika model SVM yang kita gunakan mengalami *overfitting*, maka bisa dicoba untuk



Gambar 5.4. Margin violations yang banyak (kiri) dan sedikit (kanan)

melakukan regularisasi dengan mengurangi *hyperparameter* `C`.

Kode `Scikit-Learn` berikut menggunakan dataset Iris, melakukan penskalaan pada *feature*, dan digunakan untuk mentraining model SVM linier (menggunakan *class* `LinierSVC` dengan `C=1` dan fungsi ***hinge loss***) untuk mendeteksi bunga Iris virginica. Dengan hasil yang ditunjukan pada Gambar 5.4.

[6]:
```
import numpy as np
from sklearn import datasets
from sklearn.pipeline import Pipeline
from sklearn.preprocessing import StandardScaler
from sklearn.svm import LinearSVC
import matplotlib.pyplot as plt

iris = datasets.load_iris()
X = iris["data"][:, (2, 3)]  # petal length, petal width
y = (iris["target"] == 2).astype(np.float64)  # Iris virginica

svm_clf = Pipeline([
        ("scaler", StandardScaler()),
        ("linear_svc", LinearSVC(C=1, loss="hinge", random_state=42)),
    ])

svm_clf.fit(X, y)
```

[6]:
```
Pipeline(memory=None,
         steps=[('scaler',
                 StandardScaler(copy=True, with_mean=True, with_std=True)),
                ('linear_svc',
                 LinearSVC(C=1, class_weight=None, dual=True,
                           fit_intercept=True, intercept_scaling=1,
                           loss='hinge', max_iter=1000, multi_class='ovr',
                           penalty='l2', random_state=42, tol=0.0001,
                           verbose=0))],
         verbose=False)
```

Kemudian seperti biasa, kita bisa gunakan model untuk melakukan prediksi.

[7]:
```
svm_clf.predict([[5.5, 1.7]])
```

[7]:
```
array([1.])
```

Catatan. Tidak seperti *classifier* regresi logisitik, *classfier* SVM tidak mengeluarkan probabilitas untuk setiap *class*.

Sedangkan, program berikut akan menghasilkan Gambar 5.4.

[8]:
```
scaler = StandardScaler()
svm_clf1 = LinearSVC(C=1, loss="hinge", random_state=42)
svm_clf2 = LinearSVC(C=100, loss="hinge", random_state=42)

scaled_svm_clf1 = Pipeline([
        ("scaler", scaler),
        ("linear_svc", svm_clf1),
    ])
scaled_svm_clf2 = Pipeline([
        ("scaler", scaler),
        ("linear_svc", svm_clf2),
    ])

scaled_svm_clf1.fit(X, y)
scaled_svm_clf2.fit(X, y)
```

```
/Users/fikyy.suratman/opt/anaconda3/lib/python3.7/site-
packages/sklearn/svm/_base.py:947: ConvergenceWarning: Liblinear failed to
converge, increase the number of iterations.
  "the number of iterations.", ConvergenceWarning)
```

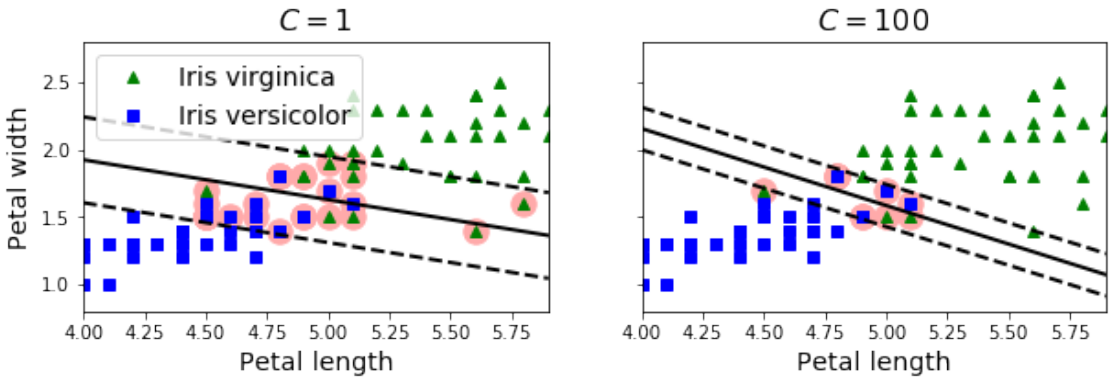
[8]:
```
Pipeline(memory=None,
         steps=[('scaler',
                 StandardScaler(copy=True, with_mean=True, with_std=True)),
                ('linear_svc',
                 LinearSVC(C=100, class_weight=None, dual=True,
                           fit_intercept=True, intercept_scaling=1,
                           loss='hinge', max_iter=1000, multi_class='ovr',
                           penalty='l2', random_state=42, tol=0.0001,
                           verbose=0))],
         verbose=False)
```

[9]:
```
# Convert to unscaled parameters
b1 = svm_clf1.decision_function([-scaler.mean_ / scaler.scale_])
b2 = svm_clf2.decision_function([-scaler.mean_ / scaler.scale_])
w1 = svm_clf1.coef_[0] / scaler.scale_
w2 = svm_clf2.coef_[0] / scaler.scale_
svm_clf1.intercept_ = np.array([b1])
svm_clf2.intercept_ = np.array([b2])
```

```
svm_clf1.coef_ = np.array([w1])
svm_clf2.coef_ = np.array([w2])

# Find support vectors (LinearSVC does not do this automatically)
t = y * 2 - 1
support_vectors_idx1 = (t * (X.dot(w1) + b1) < 1).ravel()
support_vectors_idx2 = (t * (X.dot(w2) + b2) < 1).ravel()
svm_clf1.support_vectors_ = X[support_vectors_idx1]
svm_clf2.support_vectors_ = X[support_vectors_idx2]
```

[10]:
```
fig, axes = plt.subplots(ncols=2, figsize=(10,2.7), sharey=True)

plt.sca(axes[0])
plt.plot(X[:, 0][y==1], X[:, 1][y==1], "g^", label="Iris virginica")
plt.plot(X[:, 0][y==0], X[:, 1][y==0], "bs", label="Iris versicolor")
plot_svc_decision_boundary(svm_clf1, 4, 5.9)
plt.xlabel("Petal length", fontsize=14)
plt.ylabel("Petal width", fontsize=14)
plt.legend(loc="upper left", fontsize=14)
plt.title("$C = {}$".format(svm_clf1.C), fontsize=16)
plt.axis([4, 5.9, 0.8, 2.8])

plt.sca(axes[1])
plt.plot(X[:, 0][y==1], X[:, 1][y==1], "g^")
plt.plot(X[:, 0][y==0], X[:, 1][y==0], "bs")
plot_svc_decision_boundary(svm_clf2, 4, 5.99)
plt.xlabel("Petal length", fontsize=14)
plt.title("$C = {}$".format(svm_clf2.C), fontsize=16)
plt.axis([4, 5.9, 0.8, 2.8])
```

[10]:
```
[4, 5.9, 0.8, 2.8]
```



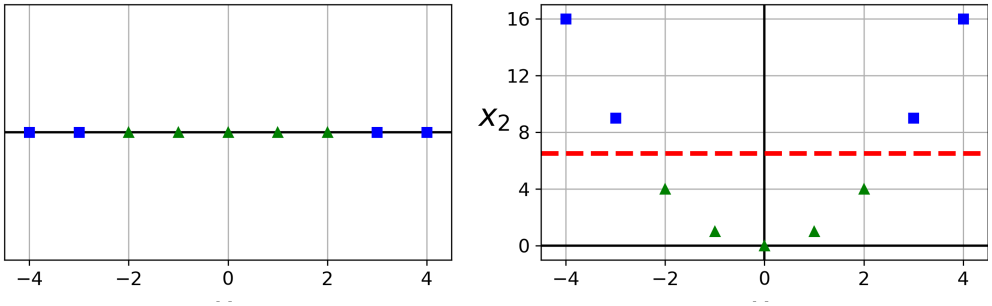
Sebagai alternatif terhadap penggunaan *class* `LinearSVC`, kita juga dapat menggunakan *class* `SVC`

dengan sebuah kernel linier, dengan menulis pada kode `SVC(kernel="linear",C=1)`. Bisa juga kita gunakan *class* `SGDClassifier` dengan `SGDClassifier(loss="hinge", alpha=1/(m*C))`, dimana akan digunakan algoritma *stochastic gradient descent* (SGD) untuk melakukan training *classifier* SVM linier. `SGDClassifier` mempunyai kecepatan konvergensi secepat `LinierSVC`, tetapi sangat berguna pada kondisi klasifikasi online atau pada dataset yang sangat besar dengan kebutuhan memori yang sangat besar (*out of core training*). SGD sudah dijelaskan pada *Chapter 2*.

Catatan. *Class* `LinierSVC` melakukan regularisasi bias, sehingga akan dilakukan pemusatan dataset training dengan cara mengurangkannya harga *mean* dari dataset training tersebut. Hal ini dilakukan secara otomatis ketika kita menggunakan penskalaan dengan `StandardScaler`, dan pastikan juga kita mensetting hyperparameter *loss* dengan `hinge` (yang bukan default). Selain itu, untuk menghasilkan kinerja lebih baik, setting hyperparameter `dual` menjadi `False`, kecuali terdapat jumlah *feature* yang lebih banyak dibandingkan dengan training *instances*.

2 Klasifikasi SVM Nonlinier

Meskipun *classifier* SVM linier cukup efisien dan bekerja dengan biak pada banyak kasus, tetapi banyak dataset yang tidak *linearly separable*. Salah satu pendekatan untuk mengatasi dataset yang nonlinier adalah dengan menambahkan *feature* yang lebih banyak, misalkan menambahkan *feature* polinomial (seperti pada Chapter 2), dimana pada beberapa kasus akan menghasilkan dataset yang *linearly separable*. Contoh ini ditunjukan pada Gambar 5.5, dimana gambar sebelah kiri merepresentasikan dataset sederhana dengan hanya satu *feature* x_1 yang tidak *linearly separable*. Tetapi jika ditambahkan *feature* kedua $x_2 = (x_1)^2$, hasil dataset dua dimensi 2D menjadi *linearly separable*.

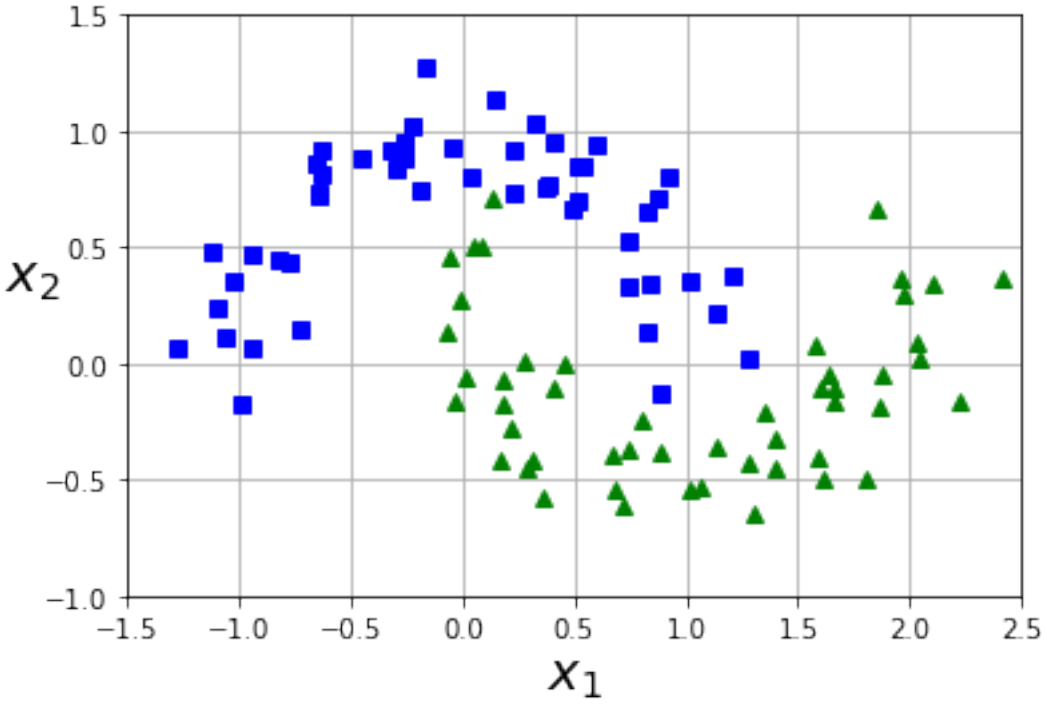


Gambar 5.5. Ilustrasi penambahan feature yang akan mebuat dataset menjadi linearly separable

Untuk mengimplementasikan ide ini menggunakan `Scikit Learn`, kita dapat membuat `Pipeline` menggunakan transformer `PolynomialFeatures`, diikuti dengan `StandardScaler` dan `LinearSVC`. Kita akan coba implementasikan metode ini pada dataset *moons* yang merupakan dataset buatan untuk klasifikasi biner dimana poin-poin data dibentuk menyerupai setengah lingkaran yang saling bertumpukan seperti diperlihatkan pada gambar yang dihasilkan dari kode berikut.

[11]:
```
from sklearn.datasets import make_moons
X, y = make_moons(n_samples=100, noise=0.15, random_state=42)

def plot_dataset(X, y, axes):
    plt.plot(X[:, 0][y==0], X[:, 1][y==0], "bs")
```

```
    plt.plot(X[:, 0][y==1], X[:, 1][y==1], "g^")
    plt.axis(axes)
    plt.grid(True, which='both')
    plt.xlabel(r"$x_1$", fontsize=20)
    plt.ylabel(r"$x_2$", fontsize=20, rotation=0)

plot_dataset(X, y, [-1.5, 2.5, -1, 1.5])
plt.show()
```



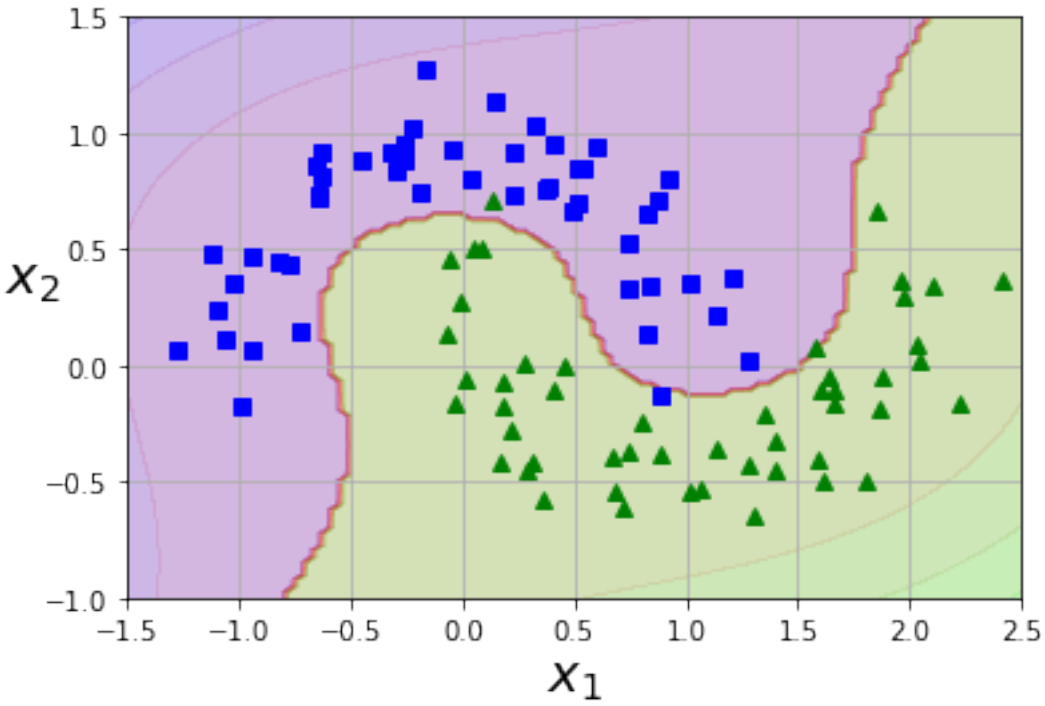
Berikut adalah implementasi klasifikasi nonlinier menggunakan *preprocessing* `PolynomialFeatures`, yang diikuti dengan `StandardScaler` kemudian `LinierSVC`.

[14]:
```
from sklearn.datasets import make_moons
from sklearn.pipeline import Pipeline
from sklearn.preprocessing import PolynomialFeatures

polynomial_svm_clf = Pipeline([
        ("poly_features", PolynomialFeatures(degree=3)),
        ("scaler", StandardScaler()),
        ("svm_clf", LinearSVC(C=10, loss="hinge", random_state=42))
    ])

polynomial_svm_clf.fit(X, y)
```

[14]:
```
Pipeline(memory=None,
         steps=[('poly_features',
                 PolynomialFeatures(degree=3, include_bias=True,
                                    interaction_only=False, order='C')),
                ('scaler',
                 StandardScaler(copy=True, with_mean=True, with_std=True)),
                ('svm_clf',
                 LinearSVC(C=10, class_weight=None, dual=True,
                           fit_intercept=True, intercept_scaling=1,
                           loss='hinge', max_iter=1000, multi_class='ovr',
                           penalty='l2', random_state=42, tol=0.0001,
                           verbose=0))],
         verbose=False)
```

Kemudian kita dapat melakukan ploting batas keputusan (*decision boundary*) hasil training di atas.

[15]:
```
def plot_predictions(clf, axes):
    x0s = np.linspace(axes[0], axes[1], 100)
    x1s = np.linspace(axes[2], axes[3], 100)
    x0, x1 = np.meshgrid(x0s, x1s)
    X = np.c_[x0.ravel(), x1.ravel()]
    y_pred = clf.predict(X).reshape(x0.shape)
    y_decision = clf.decision_function(X).reshape(x0.shape)
    plt.contourf(x0, x1, y_pred, cmap=plt.cm.brg, alpha=0.2)
    plt.contourf(x0, x1, y_decision, cmap=plt.cm.brg, alpha=0.1)

plot_predictions(polynomial_svm_clf, [-1.5, 2.5, -1, 1.5])
plot_dataset(X, y, [-1.5, 2.5, -1, 1.5])

plt.show()
```



Terlihat pada gambar di atas, batas keputusan yang dihasilkan oleh proses pelatihan dapat memisahkan *class* dengan poin-poin data kotak berwarna biru dengan *class* dengan poin-poin segitiga berwarna hijau dari dataset *moons*.

2.1 Kernel Polinomial

Menambahkan *feature* polinomial merupakan cara yang sederhana dan dapat bekerja sangat baik tidak hanya terbatas pada metode SVM saja, tetapi untuk metode-metode lain pada *machine learning*. Derajat atau orde polinomial yang rendah tidak mampu menghadapi dataset yang kompleks, tetapi dengan orde polinomial yang tinggi akan menghasilkan jumlah *feature* yang sangat besar dan membuat model menjadi terlalu lambat. Untungnya, penggunaan orde polinomial yang tinggi pada metode SVM dapat digantikan dengan penggunaan *kernel trick* (akan dijelaskan selanjutnya) dengan hasil yang serupa. Bahkan, dengan menggunakan *kernel trick* kita akan memperoleh hasil kinerja seolah-olah telah menambahkan banyak *feature* polinomial dengan orde polinomial yang sangat tinggi, sehingga disebut dengan ***kernel polinomial***. Dengan metode ini, kita dapat menghindari ledakan kombinatorial dari jumlah *feature* karena kita tidak betul-betul menambahkan banyak *feature*. *Trick* ini diimplementasikan dengan *class* `SVC`. Kode berikut adalah implementasi kernel pada SVM menggunakan dataset *moons*.

[18]:
```
from sklearn.svm import SVC

poly_kernel_svm_clf = Pipeline([
        ("scaler", StandardScaler()),
        ("svm_clf", SVC(kernel="poly", degree=3, coef0=1, C=5))
```

```
    ])
poly_kernel_svm_clf.fit(X, y)
```

[18]:
```
Pipeline(memory=None,
         steps=[('scaler',
                 StandardScaler(copy=True, with_mean=True, with_std=True)),
                ('svm_clf',
                 SVC(C=5, break_ties=False, cache_size=200, class_weight=None,
                     coef0=1, decision_function_shape='ovr', degree=3,
                     gamma='scale', kernel='poly', max_iter=-1,
                     probability=False, random_state=None, shrinking=True,
                     tol=0.001, verbose=False))],
         verbose=False)
```

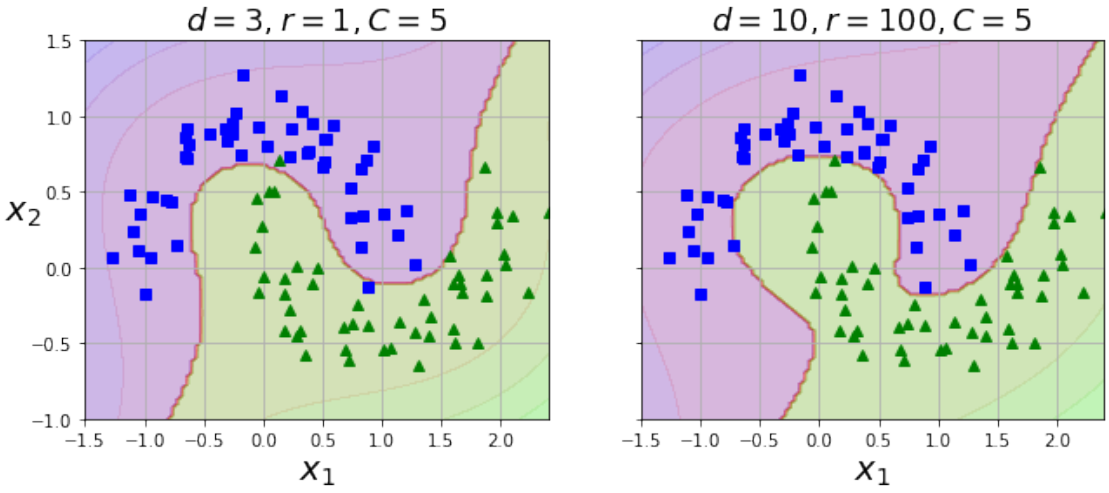
[19]:
```
poly100_kernel_svm_clf = Pipeline([
        ("scaler", StandardScaler()),
        ("svm_clf", SVC(kernel="poly", degree=10, coef0=100, C=5))
    ])
poly100_kernel_svm_clf.fit(X, y)
```

[19]:
```
Pipeline(memory=None,
         steps=[('scaler',
                 StandardScaler(copy=True, with_mean=True, with_std=True)),
                ('svm_clf',
                 SVC(C=5, break_ties=False, cache_size=200, class_weight=None,
                     coef0=100, decision_function_shape='ovr', degree=10,
                     gamma='scale', kernel='poly', max_iter=-1,
                     probability=False, random_state=None, shrinking=True,
                     tol=0.001, verbose=False))],
         verbose=False)
```

Berikut adalah plotting dari *decision boundary* hasil training menggunakan model SVC dengan kernel polinomial.

[20]:
```
fig, axes = plt.subplots(ncols=2, figsize=(10.5, 4), sharey=True)

plt.sca(axes[0])
plot_predictions(poly_kernel_svm_clf, [-1.5, 2.45, -1, 1.5])
plot_dataset(X, y, [-1.5, 2.4, -1, 1.5])
plt.title(r"$d=3, r=1, C=5$", fontsize=18)

plt.sca(axes[1])
plot_predictions(poly100_kernel_svm_clf, [-1.5, 2.45, -1, 1.5])
plot_dataset(X, y, [-1.5, 2.4, -1, 1.5])
plt.title(r"$d=10, r=100, C=5$", fontsize=18)
plt.ylabel("")
```

```
plt.show()
```



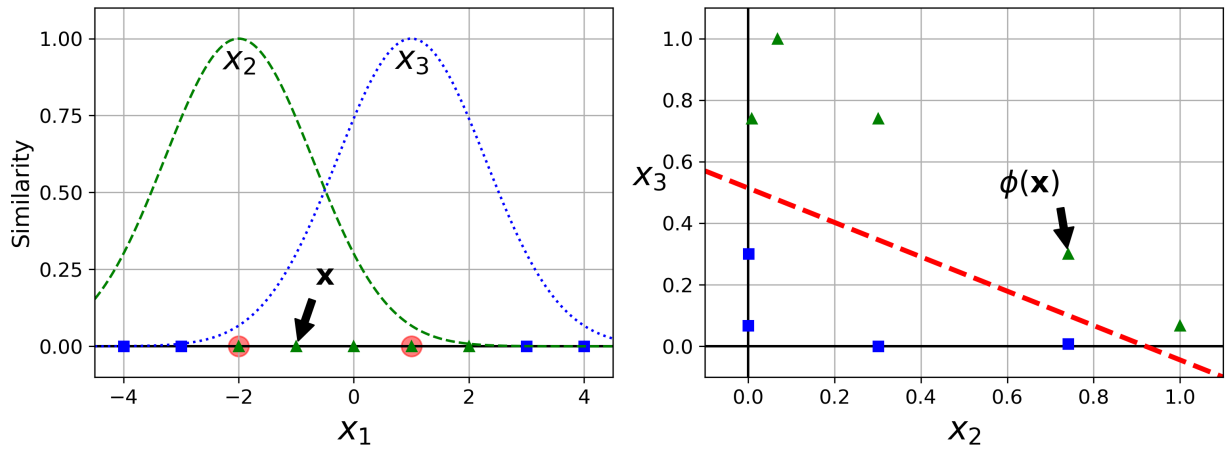
Gambar sebelah kiri menunjukan hasil eksekusi keode di atas yang melakukan training *classifier* SVM menggunakan orde 3 kernel polinomial. Sedangkan gambar sebelah kanan digunakan model yang sama tetapi menggunakan orde 10 kernel polinomial. Maka jika model *overfitting*, kita dapat mengurangi orde polinomial, sedangkan jika *underfitting* kita dapat menambah orde polinomial (lihat *Chapter 3* untuk masalah *overfitting* dan *underfitting*).

2.2 *Similarity features*

Teknik lain untuk mengatasi masalah nonlinier adalah dengan menambahkan *feature* yang dihitung dengan fungsi *similarity*. Fungsi *similarity* mengukur seberapa besar setiap *instance* menyerupai penunjuk tertentu (*landmark*). Sebagai contoh, kita akan menggunakan dataset 1 dimensi (1D) yang telah ditunjukan pada Gambar 5.5 sebelah kiri, dimana kita menambahkan dua *landmark* pada $x_1 = -2$ dan $x_1 = 1$ seperti yang ditunjukan pada Gambar 5.6 sebelah kiri. Selanjutnya, kita definisikan fungsi *similarity* berupa *Gaussian Radial Basis Function* (RBF) dengan $\gamma = 0.3$ (lihat Persamaan (5.1)).

> ***Persamaan (5.1).*** *Gaussian RBF*
>
> $\phi_\gamma(x, l) = \exp\left(-\gamma \|x - l\|^2\right)$
>
> dimana x adalah vektor *feature* dan l adalah *landmark*.

Persamaan (5.1) merupakan fungsi berbentuk lonceng (*bell-shaped*) yang merupakan fungsi Gaussian. Selanjutnya kita dapat menghitung *feature-feature* baru berdasarkan fungsi tersebut. Contoh, jika terdapat instance pada $x_1 = -1$, berarti berjarak 1 dari *landmark* pertama $\|x - l\|^2 = \|(-1) - (-2)\|^2 = 1$, sehingga *feature* baru $x_2 = \exp(-0.3 \times 1) \approx 0.74$. Sedangkan *instance* pada $x_1 = -1$ terhadap *landmark* kedua berjarak 2, $\|x - l\|^2 = \|(-1) - (1)\|^2 = 4$, yang menghasilkan *feature* baru $x_3 = \exp(-0.3 \times 4) \approx 0.3$. Gambar 5.6 sebelah kanan menunjukan hasil transformasi dataset dengan Persamaan (5.1), dengan *feature* original x_1 didrop. Dapat kita lihat transformasi *feature* telah membuat data menjadi *linearly separable*.



Gambar 5.6. Similarity feature menggunakan Gaussian Radial Basis Functions (RBF)

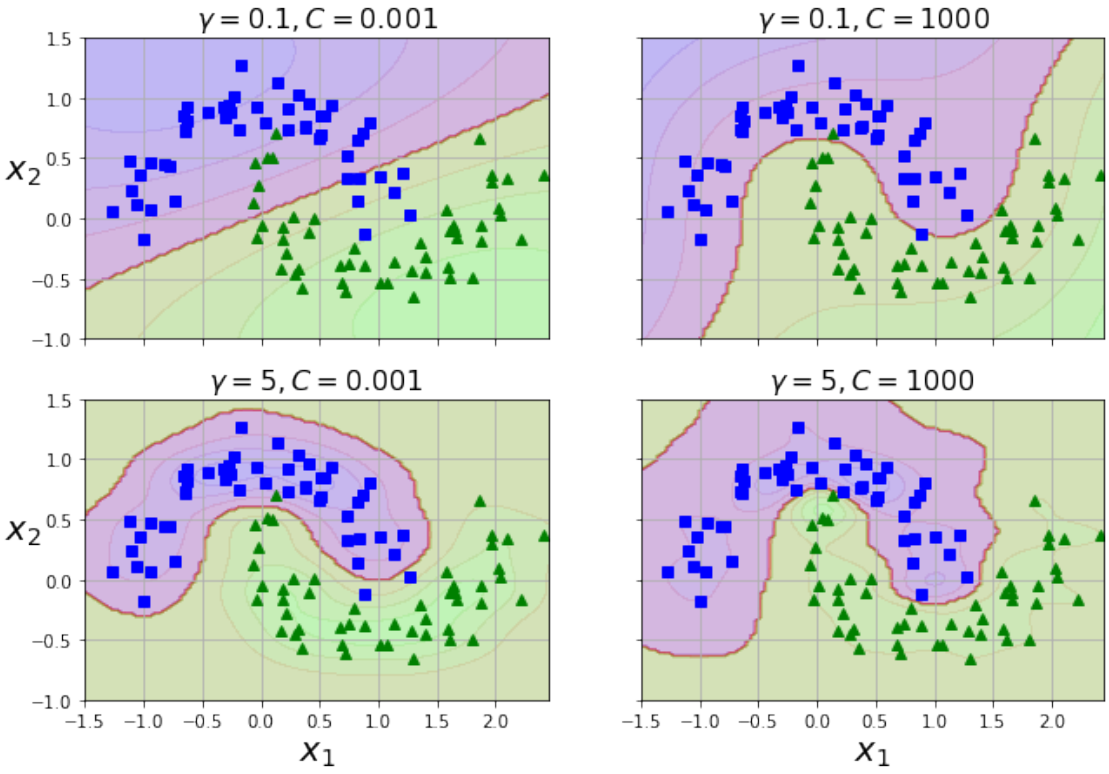
Pertanyaan selanjutnya adalah bagaimanakah memilih *landmark*? Pendekatan paling sederhana adalah membuat *landmark* pada lokasi masing-masing setiap *instance* dari dataset. Dengan melakukan itu akan menghasilkan banyak dimensi sehingga meningkatkan kemungkinan dataset training yang ditransformasi menjadi *linearly separable*. Kelemahannya adalah training set dengan m *instances* dan n *features* akan menghasilkan training set dengan m *instances* dan m *features* (dengan asumsi *feature* original didrop), sehingga jika training set sangat besar, maka akan dihasilkan jumlah *feature* yang sangat besar pula.

2.3 Kernel Gaussian RBF

Seperti metode polinomial *feature* (kernel polinomial), *similarity feature* sangat berguna untuk algoritma-algoritma *machine learning*, tetapi bisa jadi harga komputasi makin tinggi untuk menghitung *feature-feature* tambahan, terutama untuk training set yang besar. Sekali lagi *kernel trick* akan membuat keajaiban SVM, yaitu memungkinkan untuk memperoleh hasil yang serupa dengan menambahkan *similarity features*. Berikut kode yang menggunakan Kernel Gaussian RBF.

[27]:
```
from sklearn.svm import SVC

gamma1, gamma2 = 0.1, 5
C1, C2 = 0.001, 1000
hyperparams = (gamma1, C1), (gamma1, C2), (gamma2, C1), (gamma2, C2)

svm_clfs = []
for gamma, C in hyperparams:
    rbf_kernel_svm_clf = Pipeline([
            ("scaler", StandardScaler()),
            ("svm_clf", SVC(kernel="rbf", gamma=gamma, C=C)) # Kernel Gaussian␣
 ↪RBF
        ])
    rbf_kernel_svm_clf.fit(X, y)
    svm_clfs.append(rbf_kernel_svm_clf)
```

```
fig, axes = plt.subplots(nrows=2, ncols=2, figsize=(10.5, 7), sharex=True,␣
 ↪sharey=True)

for i, svm_clf in enumerate(svm_clfs):
    plt.sca(axes[i // 2, i % 2])
    plot_predictions(svm_clf, [-1.5, 2.45, -1, 1.5])
    plot_dataset(X, y, [-1.5, 2.45, -1, 1.5])
    gamma, C = hyperparams[i]
    plt.title(r"$\gamma = {}, C = {}$".format(gamma, C), fontsize=16)
    if i in (0, 1):
        plt.xlabel("")
    if i in (1, 3):
        plt.ylabel("")
plt.show()
```



Gambar di atas menunjukan model-model SVM yang menggunakan kernel Gaussian RBF, untuk nilai hyperparameter γ dan C yang berbeda-beda. Dengan bertambahnya γ maka akan menyebabkan pengaruh range dari sebuah training *instance* akan mengecil, dikarenakan kurva berbentuk lonceng akan mempunyai variansi (standar deviasi) yang mengecil. Hal ini menyebabkan kurva batas keputusan akan menjadi lebih tidak beraturan, bergerigi di sekitar masing-masing *instances*. Sebaliknya, nilai γ yang semakin kecil membuat variansi membesar, sehingga setiap *instance* akan menghasilkan range dengan pengaruh yang lebih melebar, sehingga kurva batas

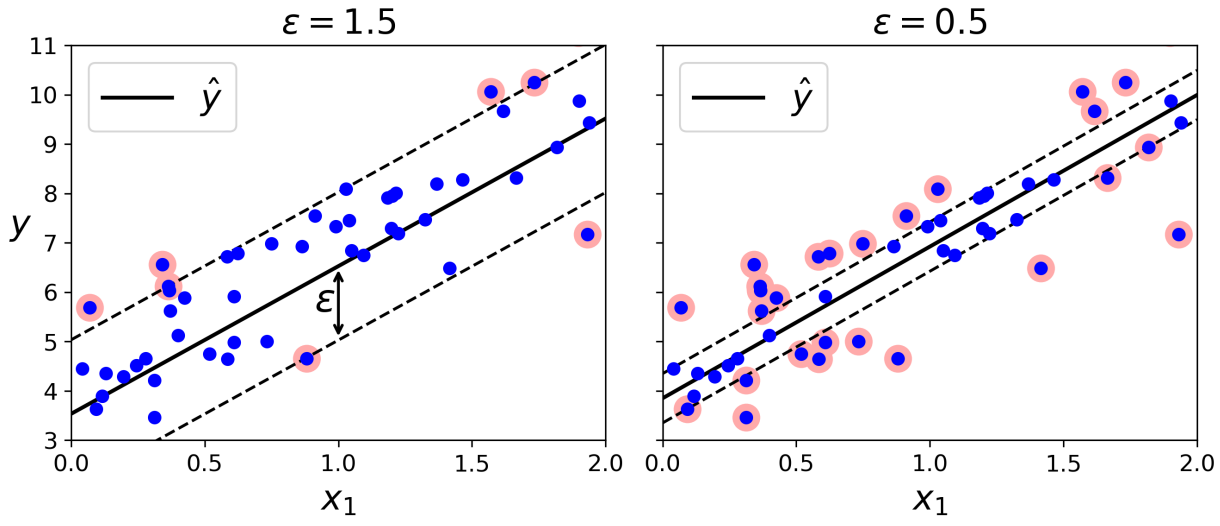
keputusan akan semakin halus. Sehingga dapat disimpulkan bahwa γ berlaku seperti hyperparameter regularisasi. Jika model mengalami *overfitting*, kita harus menguranginya, dan jika *underfitting* maka bisa ditambahkan.

Kernel-kernel jenis lain ada tetapi jarang digunakan. Beberapa kernel digunakan khusus untuk struktur data tertentu. *String kernels* kadang digunakan ketika mengklasifikasikan dokumen text atau *DNA sequences* (mis., menggunakan *string subsequnece* atau kernel-kernel yang berdasarkan *Levenshtein distance*).

Catatan. Dengan banyaknya kernel yang dapat dipilih, bagaimana kita memutuskan yang akan dipakai? Sebagai *rule of thumb*, selalu coba pertama kali dengan kernel linier (ingat bahwa `LinearSVC` lebih cepat dibandingkan dengan `SVC(kernel="linear")`) terutama jika training set sangat besar atau banyak sekali *feature* yang dipergunakan. Jika training set tidak terlalu besar, kita juga harus mencoba kernel Gaussian RBF yang bekerja baik pada hampir kebanyakan kasus. Kemudian, jika masih terdapat kelonggaran waktu dan *computing power* kita dapat bereksperimen dengan beberapa kernel lain, yang dikonfirmasi dengan *cross-validation* dan *grid search*. Biasanya kita menginginkan bereksperimen seperti itu jika terdapat kernel-kernel yang khusus untuk struktur dataset training yang kita miliki.

3 Regresi SVM

Algoritma SVM sangat beragam, tidak hanya mendukung klasifikasi linier dan nonlinier, tetapi juga mendukung untuk regresi linier dan nonlinier. Untuk menggunakan SVM sebagai regresi (bukan klasifikasi), caranya adalam membalik objektif dari *classifier* SVM. SVM untuk regeresi mempunyai objektif untuk *fitting* sebanyak mungkin *instances* pada jalan pemisah (*on the street*) sementara membatasi *margin violations* (*instance-instance* yang berada diluar jalan pemisah). Lebar jalan pemisah dikontrol dengan menggunakan hyperparameter ϵ. Gambar 5.7 menunjukan dua model SVM linier untuk regresi yang dilatih pada data linier acak, sebelah kiri dengan margin yang besar ($\epsilon = 1.5$), dan sebelah kanan dengan margin yang kecil ($\epsilon = 0.5$).



Gambar 5.7. Regresi SVM

Menambahkan data training dalam margin tidak akan mempengaruhi hasil prediksi model, se-

hingga model disebut dengan ϵ – *insensitive*. Kita dapat menggunakan *class* `LinearSVR` untuk membuat regresi SVM. Kode berikut menghasilkan model yang ditunjukan pada Gambar 5.7 sebelah kiri (training data harus diskala dan dipusatkan terlebih dahulu).

[28]:
```
np.random.seed(42)
m = 50
X = 2 * np.random.rand(m, 1)
y = (4 + 3 * X + np.random.randn(m, 1)).ravel()
```

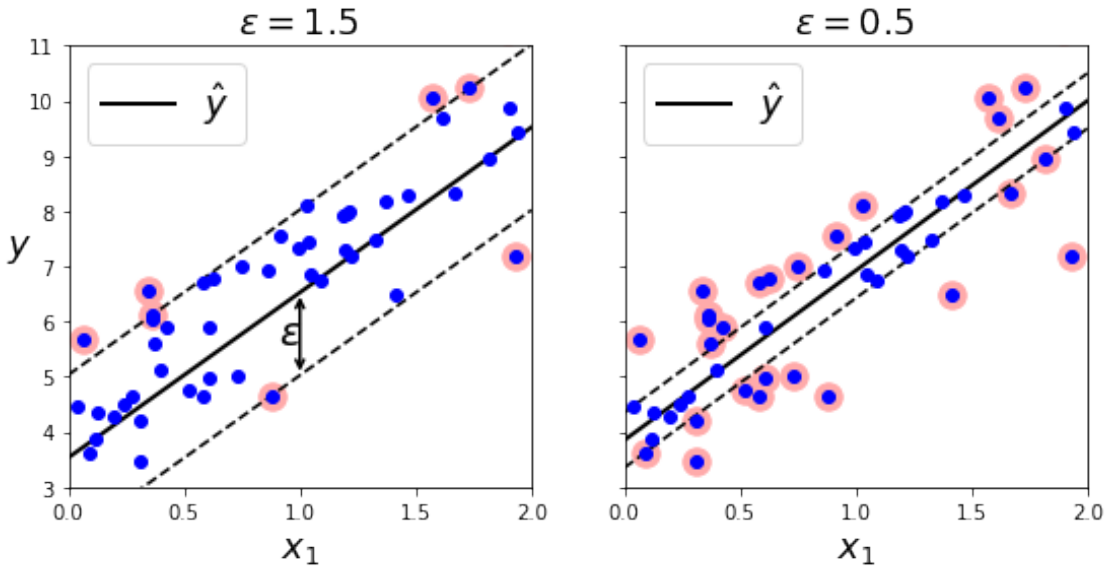
[30]:
```
from sklearn.svm import LinearSVR

svm_reg = LinearSVR(epsilon=1.5, random_state=42)
svm_reg.fit(X, y)
```

[30]:
```
LinearSVR(C=1.0, dual=True, epsilon=1.5, fit_intercept=True,
          intercept_scaling=1.0, loss='epsilon_insensitive', max_iter=1000,
          random_state=42, tol=0.0001, verbose=0)
```

[31]:
```
svm_reg1 = LinearSVR(epsilon=1.5, random_state=42)
svm_reg2 = LinearSVR(epsilon=0.5, random_state=42)
svm_reg1.fit(X, y)
svm_reg2.fit(X, y)

def find_support_vectors(svm_reg, X, y):
    y_pred = svm_reg.predict(X)
    off_margin = (np.abs(y - y_pred) >= svm_reg.epsilon)
    return np.argwhere(off_margin)

svm_reg1.support_ = find_support_vectors(svm_reg1, X, y)
svm_reg2.support_ = find_support_vectors(svm_reg2, X, y)

eps_x1 = 1
eps_y_pred = svm_reg1.predict([[eps_x1]])
```

[33]:
```
def plot_svm_regression(svm_reg, X, y, axes):
    x1s = np.linspace(axes[0], axes[1], 100).reshape(100, 1)
    y_pred = svm_reg.predict(x1s)
    plt.plot(x1s, y_pred, "k-", linewidth=2, label=r"$\hat{y}$")
    plt.plot(x1s, y_pred + svm_reg.epsilon, "k--")
    plt.plot(x1s, y_pred - svm_reg.epsilon, "k--")
    plt.scatter(X[svm_reg.support_], y[svm_reg.support_], s=180,␣
 ↪facecolors='#FFAAAA')
    plt.plot(X, y, "bo")
    plt.xlabel(r"$x_1$", fontsize=18)
    plt.legend(loc="upper left", fontsize=18)
    plt.axis(axes)
```

```
fig, axes = plt.subplots(ncols=2, figsize=(9, 4), sharey=True)
plt.sca(axes[0])
plot_svm_regression(svm_reg1, X, y, [0, 2, 3, 11])
plt.title(r"$\epsilon = {}$".format(svm_reg1.epsilon), fontsize=18)
plt.ylabel(r"$y$", fontsize=18, rotation=0)
#plt.plot([eps_x1, eps_x1], [eps_y_pred, eps_y_pred - svm_reg1.epsilon], "k-",␣
 ↪linewidth=2)
plt.annotate(
        '', xy=(eps_x1, eps_y_pred), xycoords='data',
        xytext=(eps_x1, eps_y_pred - svm_reg1.epsilon),
        textcoords='data', arrowprops={'arrowstyle': '<->', 'linewidth': 1.5}
    )
plt.text(0.91, 5.6, r"$\epsilon$", fontsize=20)
plt.sca(axes[1])
plot_svm_regression(svm_reg2, X, y, [0, 2, 3, 11])
plt.title(r"$\epsilon = {}$".format(svm_reg2.epsilon), fontsize=18)

plt.show()
```



Untuk menyelesaikan masalah regresi nonlinier, kita dapat menggunakan model *kernelized* SVM. Kode berikut akan menunjukan regresi SVM pada training set acak kuadratik, menggunakan kernel dengan polinomial orde dua.

[39]:
```
# Membangkitkan data sintetik
np.random.seed(42)
m = 100
X = 2 * np.random.rand(m, 1) - 1
y = (0.2 + 0.1 * X + 0.5 * X**2 + np.random.randn(m, 1)/10).ravel()
```

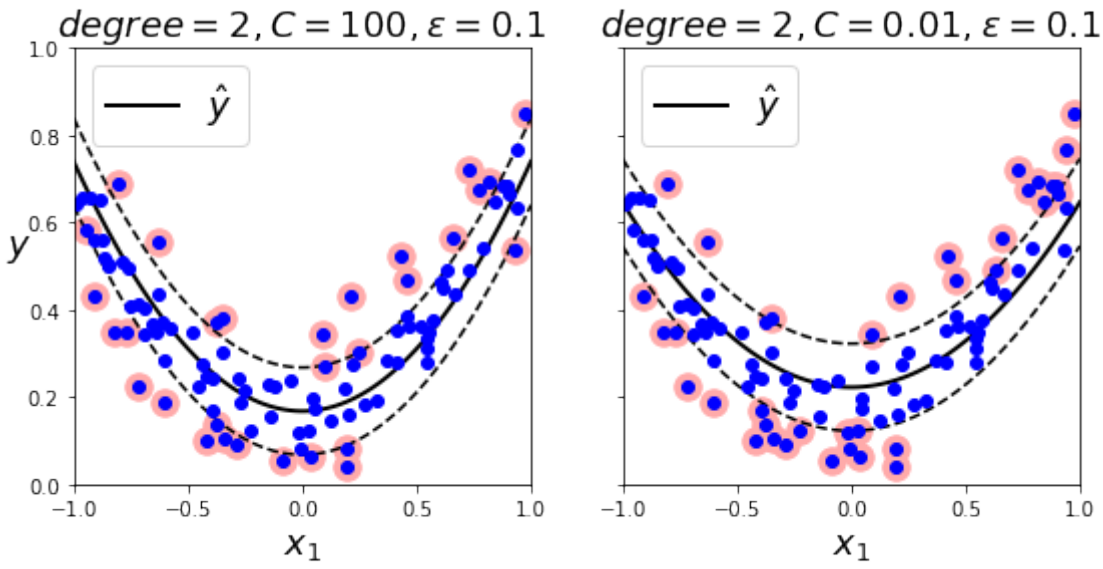
[35]:
```
from sklearn.svm import SVR

svm_poly_reg = SVR(kernel="poly", degree=2, C=100, epsilon=0.1, gamma="scale")
svm_poly_reg.fit(X, y)
```

[35]:
```
SVR(C=100, cache_size=200, coef0=0.0, degree=2, epsilon=0.1, gamma='scale',
    kernel='poly', max_iter=-1, shrinking=True, tol=0.001, verbose=False)
```

[40]:
```
# Bagian kode yang menggunakan Scikit-Learn class SVR (yang mendukung kernel␣
 ↪trick) dengan gambar di bawah
from sklearn.svm import SVR

svm_poly_reg1 = SVR(kernel="poly", degree=2, C=100, epsilon=0.1, gamma="scale")
svm_poly_reg2 = SVR(kernel="poly", degree=2, C=0.01, epsilon=0.1, gamma="scale")
svm_poly_reg1.fit(X, y)
svm_poly_reg2.fit(X, y)
```

[40]:
```
SVR(C=0.01, cache_size=200, coef0=0.0, degree=2, epsilon=0.1, gamma='scale',
    kernel='poly', max_iter=-1, shrinking=True, tol=0.001, verbose=False)
```

[41]:
```
fig, axes = plt.subplots(ncols=2, figsize=(9, 4), sharey=True)
plt.sca(axes[0])
plot_svm_regression(svm_poly_reg1, X, y, [-1, 1, 0, 1])
plt.title(r"$degree={}, C={}, \epsilon = {}$".format(svm_poly_reg1.degree,␣
 ↪svm_poly_reg1.C, svm_poly_reg1.epsilon), fontsize=18)
plt.ylabel(r"$y$", fontsize=18, rotation=0)
plt.sca(axes[1])
plot_svm_regression(svm_poly_reg2, X, y, [-1, 1, 0, 1])
plt.title(r"$degree={}, C={}, \epsilon = {}$".format(svm_poly_reg2.degree,␣
 ↪svm_poly_reg2.C, svm_poly_reg2.epsilon), fontsize=18)

plt.show()
```



Terdapat sedikit regularisasi pada gambar sebelah kiri (dengan C yang besar), dan lebih banyak lagi regularisasi pada gambar sebelah kanan (dengan C kecil).

Class `SVR` untuk regularisasi merupakan *class* ekivalen dari `SVC` untuk klasifikasi, dan *class* `LinearSVR` untuk regresi ekivalen dengan `LinearSVC` untuk klasifikasi. *Class* `LinearSVR` melakukan penskalaan secara linier sesuai dengan ukuran training (seperti pada `LinearSVC`), sementara *class* `SVR` akan melambat jika training set bertambah sangat banyak (seperti *class* `SVC`).

4 Teori SVM secara singkat

Pada bagian ini akan dijelaskan sedikit teori bagaimana SVM dibangun untuk melakukan prediksi. Pada *Chapter 2*, vector θ mendefinisikan semua parameter model, termasuk bias θ_0 dan semua bobot *feature* input dari θ_1 sampai dengan θ_n. Seperti sebelumnya $x_0 = 1$ untuk semua *instansces*. Pada bagian ini kita akan menggunakan konvensi yang lebih *common* ketika berhubungan dengan SVMs, yaitu menggunakan simbol b untuk bias, sedangkan untuk bobot *feature* kita akan gunakan vektor pembobot w. Dengan demikian vektor w tidak mengandung parameter bias w_0 karena sudah diganti dengan b.

4.1 Fungsi Keputusan (*Decision Function*) dan Prediksi (*Prediction*)

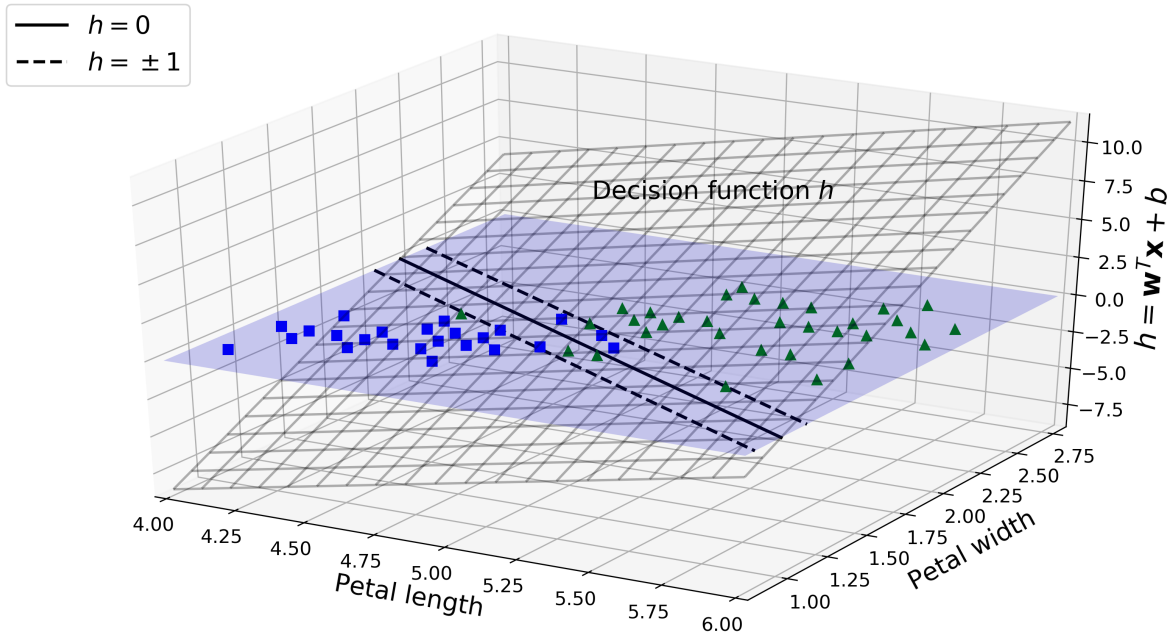
Model *classfier* SVM linier memprediksi *class* dari data instance x baru dengan cara menghitung fungsi keputusan $w^T + b = w_1x_1 + w_2x_2 + \cdots + b$. Jika hasil positif, *class* terprediksi $\hat{y}$ adalah *positive class* (1), sebaliknya akan terprediksi *negative class* (0), seperti ditunjukan oleh Persamaan (5.1).

Persamaan (5.2). Prediksi dengan *classifier* SVM Linier

$$\hat{y} = \begin{cases} 0 \text{ jika } w^T x + b < 0 \\ 1 \text{ jika } w^T x + b \geq 0 \end{cases}$$

Gambar 5.8 menunjukan fungsi keputusan pada model yang dipakai pada Gambar 5.4 sebelah kiri. Gambar 5.8menunjukan bidang 2D karena dataset ini mempunyai dua *feature* (panjang dan lebar petal). Batas keputusan (*decision boundary*) adalah satu set titik-titik dimana fungsi keputusan bernilai 0, yang merupakan perpotongan antara dua bidang. Hasil perpotongan tersebut berupa garis lurus, yang direpresentasikan dengan garis lurus hitam tebal pada Gambar 5.8. Secara umum, jika terdapat sejumlah n *feature*, fungsi keputusan adalah *hyperplane* berdimensi n, dan batas keputusan berupa *hyperplane* berdimensi (n-1).

Garis putus-putus pada Gambar 5.8 merupakan poin-poin data yang menghasilkan fungsi keputusan sama dengan 1 dan -1. Kedua garis tersebut paralel dan berjarak sama ke batas keputusan (garis lurus hitam tebal), dan keduanya membentuk margin disekitarnya. Melakukan training *classifier* SMV linear mempunyai arti menemukan vektor w dan b yang membuat margin selebar mungkin sementara juga menghindari margin *violations* (*hard margin*) atau dengan membatasinya (*soft margin*).

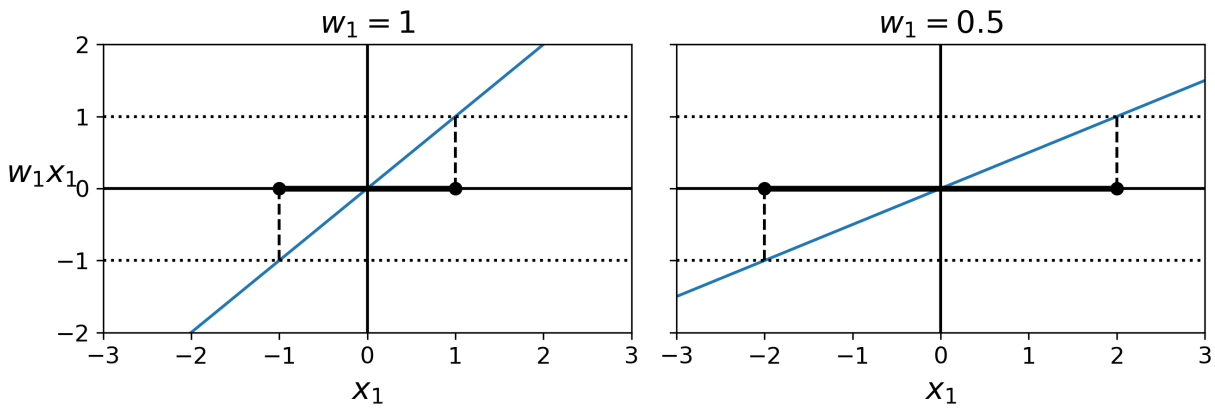


Gambar 5.8. Fungsi keputusan untuk dataset Iris

4.2 Objektif Training

Jika dimisalkan terdapat *slope* dari sebuah fungsi keputusan, yang sama dengan norm dari vektor pembobot w atau ditulis $\|w\|$. Jika kita membagi *slope* ini dengan 2, titik-titik dimana fungsi keputusan sama dengan ± 1 akan menjauh dua kali lipat dari batas keputusan. Dengan kata lain, membagi 2 *slope* akan mengalikan margin 2 kali pula. Hal ini lebih mudah dipahami dalam ilustrasi 2D yang ditunjukan pada Gambar 5.9. Semakin kecil vektor pembobot w, maka margin semakin membesar.

Sehingga kita menginginkan $\|w\|$ untuk mendapatkan margin yang lebih besar. Jika kita juga ingin menghindari adanya *margin violations* (*hard margin*), maka kita membutuhkan fungsi keputusan lebih besar dari 1 untuk semua training *instances* yang bernilai positif dan lebih kecil dari



Gambar 5.9. Vektor pembobot yang lebih kecil akan menghasilkan margin yang lebih besar

-1 untuk semua training *instances* yang bernilai negatif. Jika kita definisikan $t^{(i)} = -1$ untuk *instances* yang bernilai negatif (jika $y(i) = 0$) dan $t^{(i)} = 1$ untuk *instances* yang bernilai positif (jika $y(i) = 1$), maka kita dapat mengekspresikan kondisi ini dengan $t^{(i)}(w^T x^{(i)} + b) \geq 1$ untuk semua *instances*.

Dengan alasan di atas kita dapat megekspresikan objektif dari *classifier* SVM linier dengan *hard margin* sebagai permasalahan *constrained optimization* yang dapat dituliskan pada Persamaan (5.3)

> ***Persamaan (5.3).*** Objektif dari persamaan optimasi untuk *classifier* SVM linier dengan *hard margin*
>
> $$\begin{aligned} \underset{w,b}{\text{minimize}} \quad & \frac{1}{2} w^T w \\ \text{subject to} \quad & t^{(i)}(w^T x^{(i)} + b) \geq 1, \text{ for } i = 1, \ldots, m. \end{aligned}$$

Catatan. Kita meminimalkan $\frac{1}{2} w^T w$ yang sama dengan $\frac{1}{2}\|w\|^2$, bukan meminimalkan $\|w\|$. Hal ini karena $\frac{1}{2}\|w\|^2$ mempunyai turunan yang sama dengan w. Sedangkan $\|w\|$ tidak bisa diturunkan pada $w = 0$. Algoritma-algoritma optimasi bekerja jauh lebih baik pada fungsi-fungsi yang dapat diturunkan.

Untuk menghasilkan fungsi objektif optimasi menggunakan *soft margin* maka variabel *slack* $\zeta^{(i)} \geq 0$ untuk setiap *instance*. $\zeta^{(i)}$ mengukur seberapa besar instance ke-i diperbolehkan untuk melanggar (*violate*) margin. Sekarang kita mempunyai 2 objektif yang bertentangan, membuat variable *slack* sekecil mungkin untuk mengurangi *margin violations*, dan membuat $\frac{1}{2} w^T w$ sekecil mungkn untuk menambah margin. Pada permasalahan inilah hyperparameter `C` akan mengambil peran untuk memudahkan kita mendefinisikan *trade-off* antara dua objektif yang bertentangan tadi. Fungsi objektif permasalahan optimasi untuk *classifier* SVM dengan *soft margin* dapat dilihat pada Persamaan (5.4).

> ***Persamaan (5.4).*** Objektif dari persamaan optimasi untuk *classifier* SVM linier dengan *soft margin*
>
> $$\begin{aligned} \underset{w,b}{\text{minimize}} \quad & \frac{1}{2} w^T w + C \sum_{i=1}^{m} \zeta^{(i)} \\ \text{subject to} \quad & t^{(i)}(w^T x^{(i)} + b) \geq 1 - \zeta^{(i)} \text{ and } \zeta^{(i)} \geq 0, \text{ for } i = 1, \ldots, m. \end{aligned}$$

Permasalahan optimasi untuk *hard margin* Persamaan (5.3) dan *soft margin* Persamaan (5.4) keduanya merupakan permasalahan optimasi *convex* dengan *constrained* linier. Permasalahan ini disebut juga dengan ***Quadratic Programming*** (QP). Banyak algoritma-algoritma yang tersedia untuk memecahkan permasalahan QP dengan menggunakan bermacam teknik. Hal ini di luar pembahasan pada buku ini. Untuk mempelajari permasalahan optimasi *convex* dapat dilihat pada buku ***Convex Optimization*** dari Stephen Boyd dan Lieven Vanderberghe.