演習： SVM (NumPy)

In [1]:

```
import numpy as np
import matplotlib.pyplot as plt
%matplotlib inline
plt.style.use('ggplot')
```

In [2]:

```
def gen_data():
    """ 学習用データを生成する"""
    x0 = np.random.normal(size=50).reshape(-1, 2) - 2.
    x1 = np.random.normal(size=50).reshape(-1, 2) + 2.

    X_train = np.concatenate([x0, x1])
    y_train = np.concatenate([np.zeros(25), np.ones(25)]).astype(np.int)

    return X_train, y_train
```

In [3]:

```
X_train, y_train = gen_data()
```

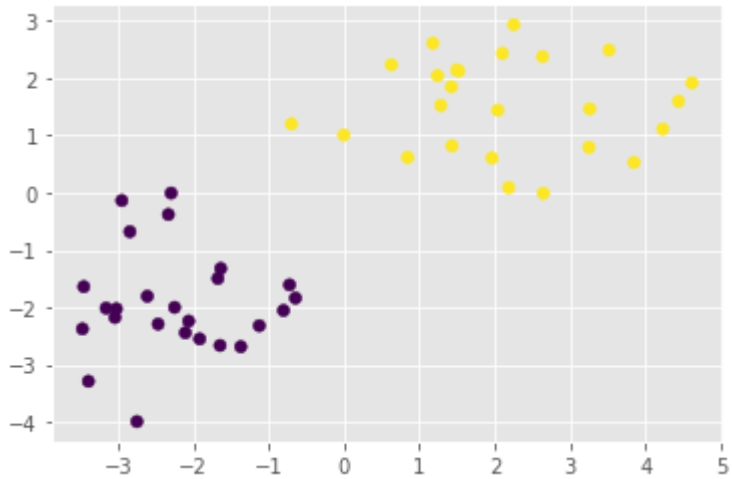
In [4]:

```
print(X_train.shape)
print(y_train.shape)
```

```
(50, 2)
(50,)
```

In [5]:

```
plt.scatter(X_train[:, 0], X_train[:, 1], c=y_train)
plt.show()
```



学習

SVMでのマージン最大化は、以下の目的関数を最大化することと同じである。
$\tilde{L}(a) = a^T 1 - \frac{1}{2} a^T H a$
ここで、a を **ラグランジュ乗数** と呼ぶ。この a を以下のように再急降下法で更新していく。
$a \leftarrow a + \eta_1 (1 - Ha) \cdots$ (1)
$a \leftarrow a - \eta_2 (a^T t) t \cdots$ (2)
また、$\frac{d\tilde{L}}{da} = 1 - Ha$ である。

In [6]:

```
eta1 = 0.01
eta2 = 0.001
```

線形カーネル： $k = XX^T$

In [7]:

```
# 線形カーネル
kernel = np.dot(X_train, X_train.T)
```

In [8]:

```
print(kernel.shape)
```

```
(50, 50)
```

In [9]:

```
t = np.where(y_train == 1.0, 1.0, -1.0)
print(t)
```

```
[-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.]
```

In [10]:

```
H = np.outer(t, t) * kernel  # np.outerは外積を求める
```

In [11]:

```
print(H.shape)
```

```
(50, 50)
```

In [12]:

```
n_samples = len(X_train)
a = np.ones(n_samples)  # aの初期値を設定する
print(a.shape)
```

```
(50,)
```

In [13]:

```
for _ in range(500):
    grad = 1 - np.dot(H, a)  # 目的関数の微分
    a += eta1 * grad  # 上式の(1)
    a -= eta2 * np.dot(np.dot(a.T, t), t)  # 上式の(2)
    a = np.where(a > 0, a, 0)
```

In [14]:

```
print(a.shape)
print(a)
```

```
(50,)
[0.         0.         0.         0.         0.         0.
 0.         0.         0.         0.46410139 0.         0.
 0.         0.         0.         0.         0.         0.
 0.         0.         0.         0.         0.         0.
 0.         0.         0.         0.         0.         0.
 0.         0.8710422  0.         0.         0.         0.
 0.         0.         0.         0.         0.         0.
 0.         0.        ]
```

予測

新しいデータ点 x については $y(x)$ の正負で分類する。
ここで、先ほど求めた a で $a_i = 0$ は予測に影響しないので $a_i > 0$ に対応するデータ点（ **サポートベクトル** ）のみ保持する。
b はサポートベクトルのインデックス集合 S とすると、以下のように求められる。
$b = \frac{1}{S} \sum_{s \in S} (t_s - \sum_{i=1}^{n} a_i t_i k(x_i, x_s))$

In [15]:

```
index = a > 1e-6  # 条件を満たすのがTrueになる
support_vectors = X_train[index]  # X_train と True に対応する配列を作成する
```

In [16]:

```
print(index.shape)
print(support_vectors)
```

```
(50,)
[[-2.29368129 -0.0046888 ]
 [-0.69853701  1.20284513]]
```

In [17]:

```
support_vector_t = t[index]
print(support_vector_t)
```

```
[-1.  1.]
```

In [18]:

```
support_vector_a = a[index]
print(support_vector_a)
```

[0.46410139 0.8710422]

In [19]:

```
kernel[index][:, index]
```

Out[19]:

```
array([[5.26099586, 1.59658137],
       [1.59658137, 1.93479035]])
```

In [20]:

```
# bを求める式の第2項
term2 = kernel[index][:, index].dot(support_vector_a * support_vector_t)
```

In [21]:

```
b = (support_vector_t - term2).mean()
```

In [22]:

```
print(b)
```

0.05331865338257663

In [23]:

```
xx0, xx1 = np.meshgrid(np.linspace(-5, 5, 100), np.linspace(-5, 5, 100))
X_test = np.array([xx0, xx1]).reshape(2, -1).T
```

In [24]:

```
print(xx0.shape)
print(xx1.shape)
print(X_test.shape)
```

(100, 100)
(100, 100)
(10000, 2)

In [25]:

```
y_project = np.ones(len(X_test)) * b
```

In [26]:

```
print(y_project.shape)
print(y_project)
```

(10000,)
[0.05331865 0.05331865 0.05331865 ... 0.05331865 0.05331865 0.05331865]

In [27]:

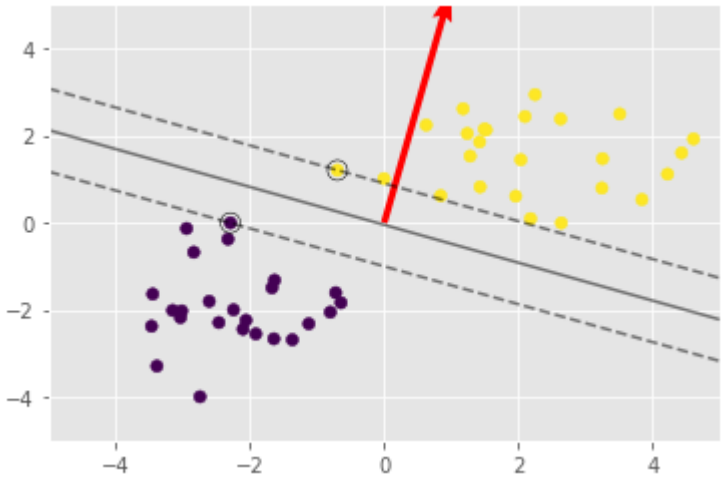
```
for i in range(len(X_test)):
    for a, sv_t, sv in zip(support_vector_a, support_vector_t, support_vectors):
        y_project[i] += a * sv_t * sv.dot(X_test[i])
```

In [28]:

```
y_pred = np.sign(y_project)
```

In [40]:

```
# 訓練データを可視化
plt.scatter(X_train[:, 0], X_train[:, 1], c=y_train)
# サポートベクトルを可視化
plt.scatter(support_vectors[:, 0], support_vectors[:, 1],
            s=100, facecolors='none', edgecolors='k')
# マージンと決定境界を可視化
plt.contour(xx0, xx1, y_project.reshape(100, 100), colors='k',
            levels=[-1, 0, 1], alpha=0.5, linestyles=['--', '-', '--'])

# ベクトルを可視化 ([X, Y], U, V, [C], **kw)
plt.quiver(0, 0, 0.1, 0.35, width=0.01, scale=1, color='red')
plt.show()
```



考察：
サポートベクトルの決定は、うまくいっているように見える。
マージンを最大にするようなサポートベクトルがきちんと選択されているため。

訓練データ生成②　（線形分離不可能）

In [83]:

```
factor = .2
n_samples = 50
linspace = np.linspace(0, 2 * np.pi, n_samples // 2 + 1)[:-1]
outer_circ_x = np.cos(linspace)
outer_circ_y = np.sin(linspace)
inner_circ_x = outer_circ_x * factor
inner_circ_y = outer_circ_y * factor
```

In [84]:

```
X_train = np.vstack((np.append(outer_circ_x, inner_circ_x),
                     np.append(outer_circ_y, inner_circ_y))).T  # 縦に結合して転置する
X_train += np.random.normal(scale=0.15, size=X.shape)  # データ点にランダム値を与える
print(X_train[20:30])
```

```
[[ 0.29098055 -1.09206916]
 [ 0.7271199  -0.91576883]
 [ 0.98527548 -0.6457048 ]
 [ 1.00814739 -0.26935862]
 [ 0.96607428 -0.09083515]
 [ 0.0866548  -0.2108567 ]
 [ 0.05690082  0.12030869]
 [ 0.15025617 -0.00855903]
 [ 0.09742251  0.0089035 ]
 [ 0.39528398  0.37241041]]
```

In [85]:

```
y_train = np.hstack([np.zeros(n_samples // 2, dtype=np.intp),
                     np.ones(n_samples // 2, dtype=np.intp)])
print(y_train)
```

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

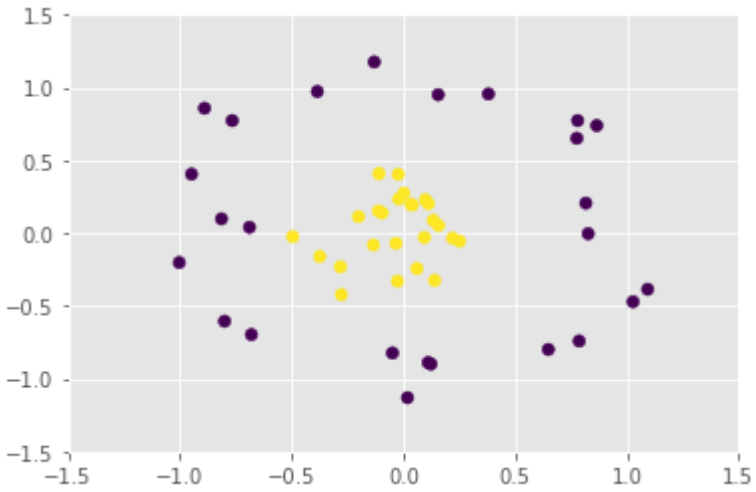
In [86]:

```
print(X_train.shape)
print(y_train.shape)
```

```
(50, 2)
(50,)
```

In [101]:

```
plt.scatter(x_train[:,0], x_train[:,1], c=y_train)
plt.xlim(-1.5, 1.5)
plt.ylim(-1.5, 1.5)
plt.show()
```



上記グラフのデータ点は、線形分離が不可能である。つまり直線で分離ができない。

学習

元のデータ空間では線形分離は出来ないが、特徴空間上で線形分離することを考える。
今回はRBFカーネル（ガウシアンカーネル）を利用する。

$k(x_i, x_j) = exp(-\frac{||x_i - x_j||^2}{2\sigma^2})$

In [88]:

```
def rbf(u, v):
        sigma = 0.8
        kernel = np.exp(-0.5 * ((u - v)**2).sum() / sigma**2)
        return kernel
```

In [89]:

```
t = np.where(y_train == 1.0, 1.0, -1.0)
print(t)
```

```
[-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.]
```

In [90]:

```
n_samples = len(X_train)
print(n_samples)
```

```
50
```

In [91]:

```
# RBFカーネル
kernel = np.zeros((n_samples, n_samples))  # 初期化

for i in range(n_samples):
    for j in range(n_samples):
        kernel[i, j] = rbf(X_train[i], X_train[j])

print(kernel.shape)
print(kernel)
```

```
(50, 50)
[[1.         0.94284778 0.78773079 ... 0.75674245 0.52908795 0.56173764]
 [0.94284778 1.         0.91447702 ... 0.55881457 0.33966749 0.36733754]
 [0.78773079 0.91447702 1.         ... 0.41680098 0.2020624  0.24163683]
 ...
 [0.75674245 0.55881457 0.41680098 ... 1.         0.86274321 0.93469245]
 [0.52908795 0.33966749 0.2020624  ... 0.86274321 1.         0.96508421]
 [0.56173764 0.36733754 0.24163683 ... 0.93469245 0.96508421 1.        ]]
```

In [92]:

```
eta1 = 0.01
eta2 = 0.001
n_iter = 5000
```

In [93]:

```
H = np.outer(t, t) * kernel
```

In [94]:

```
# 線形分離可能問題と同様に、再急降下法でaを更新していく
a = np.ones(n_samples)

for _ in range(n_iter):
    grad = 1 - H.dot(a)
    a += eta1 * grad
    a -= eta2 * a.dot(t) * t
    a = np.where(a > 0, a, 0)
```

予測

新しいデータ点 x に対しては $y(x) = w\phi(x) + b = \sum_{i=1}^{n} a_i t_i k(x, x_i) + b$ の正負によって分類する。
ここで、最適化の結果得られた $a_i (i = 1, 2, \ldots, n)$ の中で $a_i = 0$ に対応するデータ点は予測に影響を与えないので，$a_i > 0$ に対応するデータ点（サポートベクトル）のみ保持しておく。b はサポートベクトルのインデックスの集合を S とすると，$b = \frac{1}{S} \sum_{s \in S} (t_s - \sum_{i=1}^{n} a_i t_i k(x_i, x_s))$ によって求める。

In [95]:

```
index = a > 1e-6
support_vectors = X_train[index]
support_vector_t = t[index]
support_vector_a = a[index]
```

In [96]:

```
term2 = kernel[index][:, index].dot(support_vector_a * support_vector_t)  # bの第2項を求める
b = (support_vector_t - term2).mean()
```

In [97]:

```
xx0, xx1 = np.meshgrid(np.linspace(-1.5, 1.5, 100), np.linspace(-1.5, 1.5, 100))
X_test = np.array([xx0, xx1]).reshape(2, -1).T
```

In [98]:

```
y_project = np.ones(len(X_test)) * b

for i in range(len(X_test)):
    for a, sv_t, sv in zip(support_vector_a, support_vector_t, support_vectors):
        y_project[i] += a * sv_t * rbf(X_test[i], sv)
```

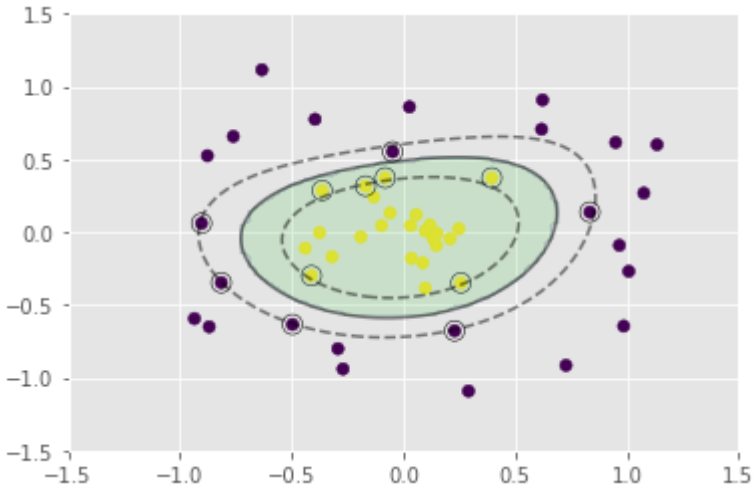
In [99]:

```
y_pred = np.sign(y_project)
print(y_pred.shape)
```

```
(10000,)
```

In [100]:

```
# 訓練データを可視化
plt.scatter(X_train[:, 0], X_train[:, 1], c=y_train)
# サポートベクトルを可視化
plt.scatter(support_vectors[:, 0], support_vectors[:, 1], s=100, facecolors='none', edgecolors
='k')
# 領域を可視化
plt.contourf(xx0, xx1, y_pred.reshape(100, 100), alpha=0.2, levels=np.linspace(0, 1, 3))
# マージンと決定境界を可視化
plt.contour(xx0, xx1, y_project.reshape(100, 100), colors='k', levels=[-1, 0, 1], alpha=0.5, lin
estyles=['--', '-', '--'])
plt.show()
```



上記のように、線形分離不可能なデータセットも、カーネルトリックを利用することで
分類がうまくできていることがわかる。

ソフトマージンSVM

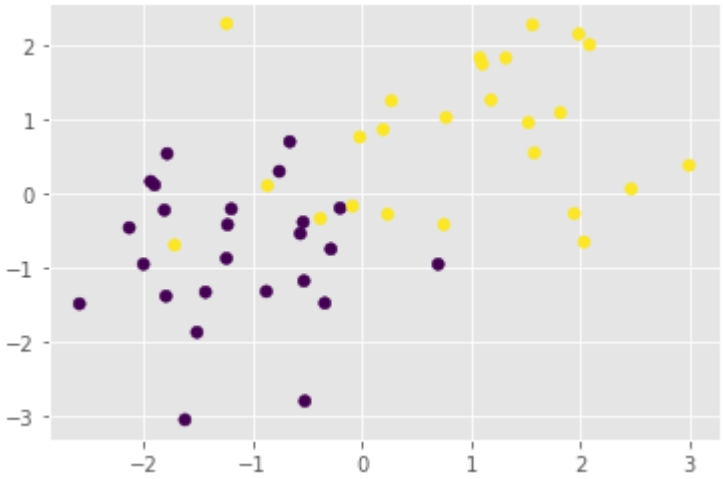
訓練データ生成③（重なりあり）

In [103]:

```
x0 = np.random.normal(size=50).reshape(-1, 2) - 1.
x1 = np.random.normal(size=50).reshape(-1, 2) + 1.
X_train = np.concatenate([x0, x1])
y_train = np.concatenate([np.zeros(25), np.ones(25)]).astype(np.int)
```

In [104]:

```
plt.scatter(X_train[:, 0], X_train[:, 1], c=y_train)
plt.show()
```



学習

分離不可能な場合は学習できないが、データ点がマージン内部に入ることや誤分類を許容することでその問題を回避する。スラック変数 $\xi_i \geq 0$ を導入し、マージン内部に入ったり誤分類された点に対しては, $\xi_i = |1 - t_i y(x_i)|$ とし、これらを許容する代わりに対しペナルティを与えるように、最適化問題を以下のように修正する。

$$\begin{aligned} &\min_{w,b} && \frac{1}{2}||w||^2 + C\sum_{i=1}^{n}\xi_i \\ &\text{subject to} && t_i(w\phi(x_i)+b) \geq 1-\xi_i \quad (i=1,2,\cdots,n) \end{aligned}$$

ただし, C はマージンの大きさと誤差の許容度のトレードオフを決めるパラメータである。この最適化問題をラグランジュ乗数法などを用いると、結局最大化する目的関数はハードマージンSVMと同じとなる。

$$\tilde{L}(a) = \sum_{i=1}^{n} a_i - \frac{1}{2}\sum_{i=1}^{n}\sum_{j=1}^{n} a_i a_j t_i t_j \phi(x_i)^{\mathrm{T}}\phi(x_j)$$

ただし、制約条件が $a_i \geq 0$ の代わりに $0 \leq a_i \leq C (i = 1, 2, \ldots, n)$ となる。（ハードマージンSVMと同じ $\sum_{i=1}^{n} a_i t_i = 0$ も制約条件）

In [105]:

```
t = np.where(y_train == 1.0, 1.0, -1.0)
n_samples = len(X_train)
```

In [106]:

```
# 線形カーネル
kernel = np.dot(X_train, X_train.T)
```

In [107]:

```
C = 1
eta1 = 0.01
eta2 = 0.001
n_iter = 1000
H = np.outer(t, t) * kernel
```

In [110]:

```
# 再急降下法でaを更新していく
a = np.ones(n_samples)

for _ in range(n_iter):
    grad = 1 - H.dot(a)
    a += eta1 * grad
    a -= eta2 * a.dot(t) * t
    a = np.clip(a, 0, C)

print(a)
```

```
[1.         0.         0.         0.         1.         0.
 1.         1.         1.         0.         0.70050465 0.95150066
 1.         1.         0.         0.012274   1.         0.98734181
 1.         1.         0.         0.21643642 0.         0.1366398
 1.         0.         0.         0.         0.         1.
 1.         0.         0.         1.         0.17985432 1.
 1.         0.         0.95143499 0.         1.         1.
 0.         0.         0.         1.         0.         0.60562175
 1.         0.        ]
```

予測

In [112]:

```
index = a > 1e-8
support_vectors = X_train[index]
support_vector_t = t[index]
support_vector_a = a[index]
```

In [114]:

```
term2 = kernel[index][:, index].dot(support_vector_a * support_vector_t)
b = (support_vector_t - term2).mean()
```

In [115]:

```
xx0, xx1 = np.meshgrid(np.linspace(-4, 4, 100), np.linspace(-4, 4, 100))
X_test = np.array([xx0, xx1]).reshape(2, -1).T
```

In [116]:

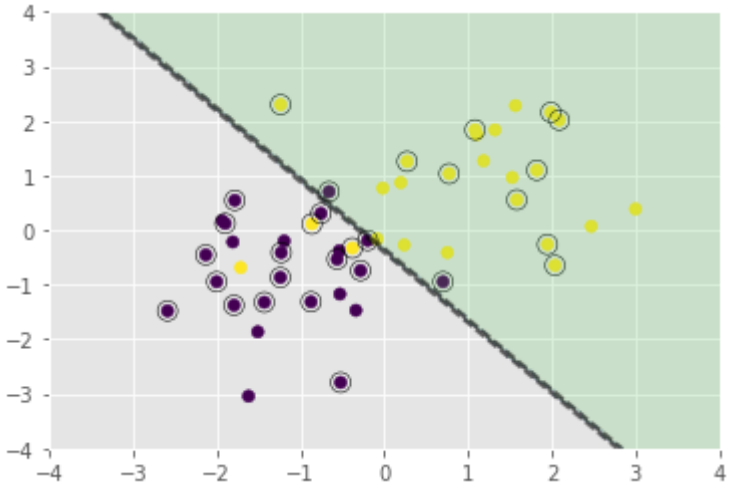
```
y_project = np.ones(len(X_test)) * b

for i in range(len(X_test)):
    for a, sv_t, sv in zip(support_vector_a, support_vector_t, support_vectors):
        y_project[i] += a * sv_t * sv.dot(X_test[i])
```

In [117]:

```
y_pred = np.sign(y_project)
```

In [118]:

```
# 訓練データを可視化
plt.scatter(X_train[:, 0], X_train[:, 1], c=y_train)
# サポートベクトルを可視化
plt.scatter(support_vectors[:, 0], support_vectors[:, 1], s=100, facecolors='none', edgecolors
='k')
# 領域を可視化
plt.contourf(xx0, xx1, y_pred.reshape(100, 100), alpha=0.2, levels=np.linspace(0, 1, 3))
# マージンと決定境界を可視化
plt.contour(xx0, xx1, y_project.reshape(100, 100), colors='k',
                    levels=[-1, 0, 1], alpha=0.5, linestyles=['--', '-', '--'])
plt.show()
```



In [ ]:

```

```