Study AI レポート

機械学習

・線形回帰モデル

回帰問題はある入力（離散あるいは連続値）」から出力（連続値）を予測する問題である。回帰で扱うデータは入力の各要素を説明変数または特徴量と呼ぶ。出力は目的変数と呼ばれ、スカラー値を取る。線形回帰モデルは回帰問題を解くための機械学習モデルの一つであり、教師あり学習である。入力と m 次元パラメータの線形結合を出力するモデルであり、m=1 の時は単回帰モデルと呼ばれる。入力とパラメータの内積に切片も加えられる。未知のパラメータを連立方程式、行列計算で表現される。線形回帰モデルのパラメータは最小二乗法で推定される。ボストンの住宅データセットを線形回帰モデルで分析する例がよくテストセットとして使用されている。

```
def train(xs, ys):
    cov = np.cov(xs, ys, ddof=0)
    a = cov[0, 1] / cov[0, 0]
    b = np.mean(ys) - a * np.mean(xs)
    return cov, a, b
```

```
from sklearn.linear_model import LinearRegression
model = LinearRegression()
reg = model.fit(xs.reshape(-1, 1), ys.reshape(-1, 1))

print("coef_: {}".format(reg.coef_))
print("intercept_: {}".format(reg.intercept_))
```

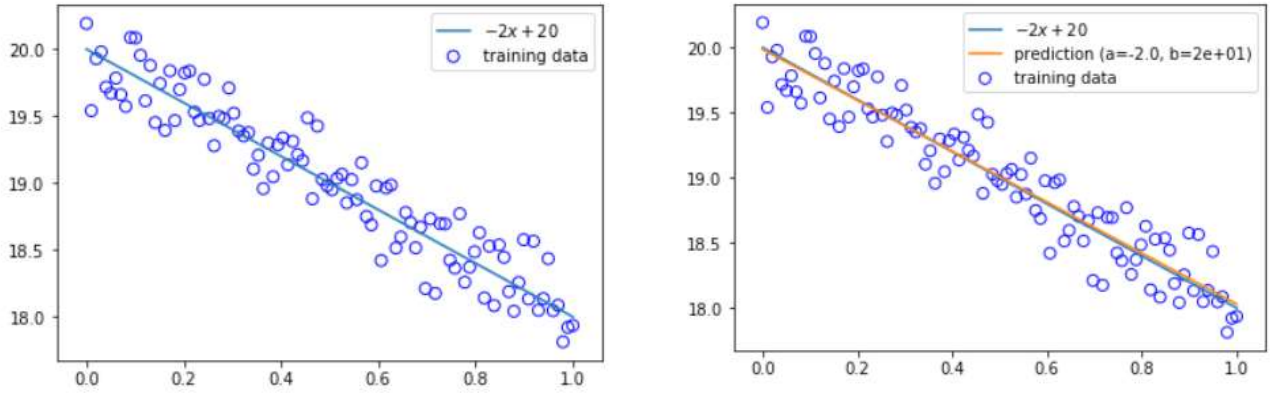


図 1）最小二乗法で予測した線形回帰モデルとの比較

・非線形回帰モデル

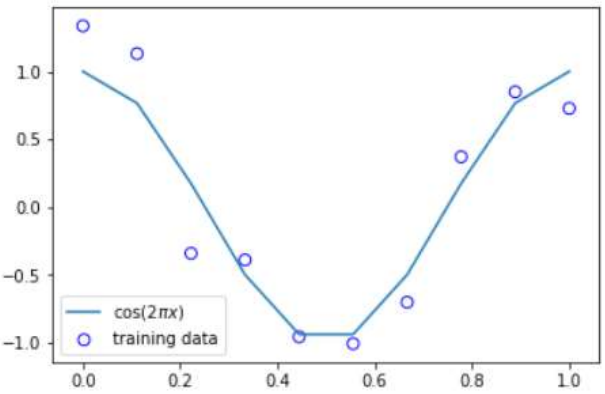
現実問題としてデータの構造を線形で捉えられる場合は限られている。複雑な非線形構造を内在する現象に対して非線形回帰モデリングを実施するのが非線形回帰モデルである。非線形回帰モデルには基底展開法というという手法が用いられる。基底関数と呼ばれる既知の非線形関数とパラメータベクトルの線形結合によってモデルを表す方法である。基底関数としては多項式関数、ガウス型基底関数、スプライン関数などがある。未知パラメータは最小二乗法や最尤法により推定される。非線形回帰モデルは優れた方法であるが、表現力の高すぎるモデルが使われた場合過学習が起きやすいという問題があり、また表現力が低いモデルだと未学習が発生する。適切な表現力のモデルを使用する事が重要で、その実現のために正則化法などモデルが複雑になりすぎないようにする工夫がある。

機械学習で重要な事は、学習したデータに対する結果よりも未学習のデータに対する推論の正しさ即ち汎化性能なので、クロスバリデーションなどの全データを分割し、学習用と検証データを入れ替えながら性能を評価する手法がある。

```python
def polynomial_features(xs, degree=3):
    """多項式特徴ベクトルに変換
        X = [[1, x1, x1^2, x1^3],
             [1, x2, x2^2, x2^3],
              ...
             [1, xn, xn^2, xn^3]]"""
    X = np.ones((len(xs), degree+1))
    X_t = X.T #(100, 4)
    for i in range(1, degree+1):
        X_t[i] = X_t[i-1] * xs
    return X_t.T
```

```python
Phi = polynomial_features(xs)
Phi_inv = np.dot(np.linalg.inv(np.dot(Phi.T, Phi)), Phi.T)
w = np.dot(Phi_inv, ys)
```

```python
Phi_test = polynomial_features(xs)
ys_pred = np.dot(Phi_test, w)
```



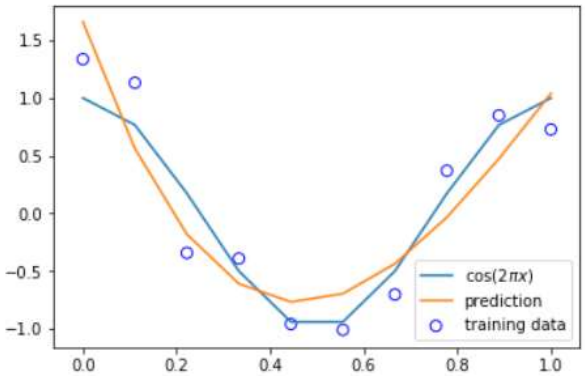
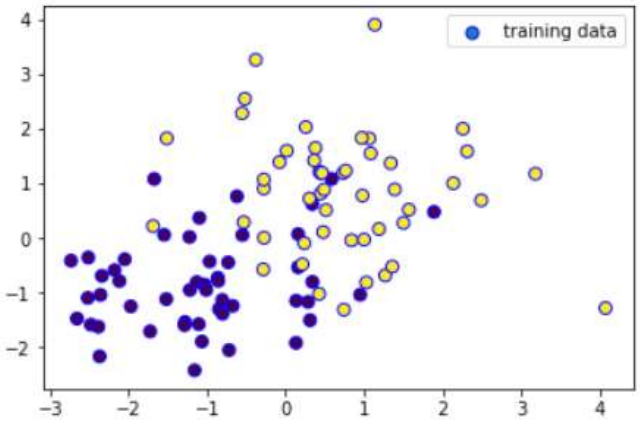


図２）３次多項式回帰予測　cos 式と予測

・ロジスティック回帰モデル

ロジスティクス回帰モデルは、名前は回帰であるが２クラス分類を解くために使用される手法である。ロジスティクス回帰は線形分離可能なクラスに対して高い性能を発揮し、産業界において最も広く使用されている分類アルゴリズムの一つである。ロジスティクス回帰の大きな特徴は、活性化関数としてシグモイド関数を使用する事である。ステップ関数では微分ができないが、シグモイド関数は微分可能であり、あるクラスに分類される確率として出力できる事が利点である。例えばアヤメの A 種類である可能性が 80%などと出せるので、後は判別する閾値（例 50%）よりも高いか低いかで分類を実施する事ができる。

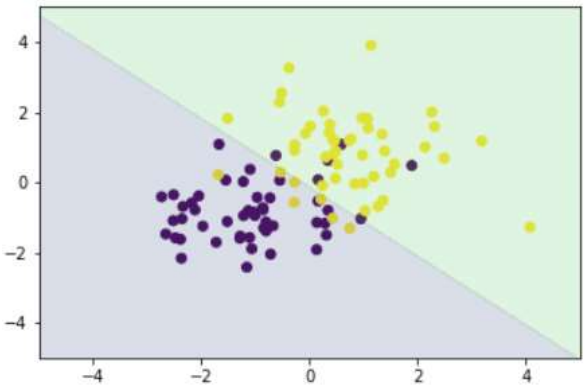



```python
def sigmoid(x):
    return 1 / (1 + np.exp(-x))

def sgd(X_train, max_iter, eta):
    w = np.zeros(X_train.shape[1])
    for _ in range(max_iter):
        w_prev = np.copy(w)
        sigma = sigmoid(np.dot(X_train, w))
        grad = np.dot(X_train.T, (sigma - y_train))
        w -= eta * grad
        if np.allclose(w, w_prev):
            return w
    return w

X_train = add_one(x_train)
max_iter=100
eta = 0.01
w = sgd(X_train, max_iter, eta)
```

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

```python
X_test = add_one(xx)
proba = sigmoid(np.dot(X_test, w))
y_pred = (proba > 0.5).astype(np.int)
```

0.5 より大きければクラス１，小さければ０

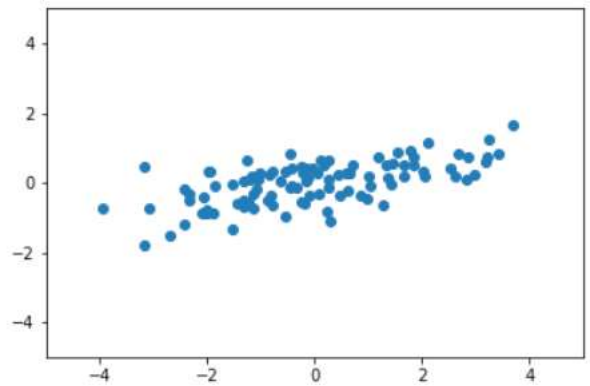


sklearn.linear_model の model=LogisticRegression(fit_intercept=True)を用いても同じ結果になる

・主成分分析

主成分分析とは、多変量データを持つ構造をより小数個の指標に圧縮する手法である。変量の個数を減らすことに伴う情報の損失をなるべく小さくする事が重要で、二次元か三次元にまで圧縮する事で可視化が可能になる。情報の量を分散の大きさと捉えて、線形変換後の変数の分散が最大となる射影軸を探索する事が重要である。主成分分析は PCA（Prinicipal Component Analysis）と略され、広く使用されている教師なし線形変換法である。PCA は以下の手順で進められる。1.m 次元のデータセットを標準化する 2.データセットの共分散行列を作成する 3.共分散行列を固有ベクトルと固有値に分解する 4.固有値を降順でソートする事で、対応する固有ベクトルをランク付けする 5.最も大きい k 個の固有値に対応する k 個の固有ベクトルを選択する 6.上位 k 個の固有ベクトルから射影行列Wを生成する 7. 射影行列Wを使って d 次元の入力データセット X を変換し、新しい k 次元の特徴部分空間を取得する。

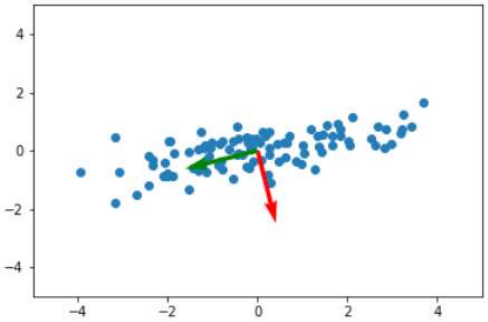
```python
n_sample = 100

def gen_data(n_sample):
    mean = [0, 0]
    cov = [[3, 0.8], [0.8, 0.5]]
    return np.random.multivariate_normal(mean, cov, n_sample)
```



```python
n_components=2

def get_moments(X):
    mean = X.mean(axis=0)
    stan_cov = np.dot((X - mean).T, X - mean) / (len(X) - 1)
    return mean, stan_cov

def get_components(eigenvectors, n_components):
#     W = eigenvectors[:, -n_components:]
#     return W.T[::-1]
    W = eigenvectors[:, ::-1][:, :n_components]
    return W.T

def plt_result(X, first, second):
    plt.scatter(X[:, 0], X[:, 1])
    plt.xlim(-5, 5)
    plt.ylim(-5, 5)
    # 第１主成分
    plt.quiver(0, 0, first[0], first[1], width=0.01, scale=6, color='red')
    # 第２主成分
    plt.quiver(0, 0, second[0], second[1], width=0.01, scale=6, color='green')
```

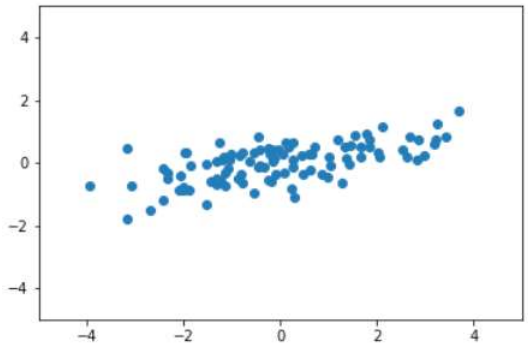
```python
#分散共分散行列を標準化
meean, stan_cov = get_moments(X)
#固有値と固有ベクトルを計算
eigenvalues, eigenvectors = np.linalg.eigh(stan_cov)
components = get_components(eigenvectors, n_components)

plt_result(X, eigenvectors[0, :], eigenvectors[1, :])
```

分散を持ったデータを作成し、第一主成分と第二主成分に色をつけたベクトル表示をする



```
mean = X.mean(axis=0)
X_ = np.dot(Z, components.T) + mean
```

```
plt.scatter(X_[:, 0], X_[:, 1])
plt.xlim(-5, 5)
plt.ylim(-5, 5)
```

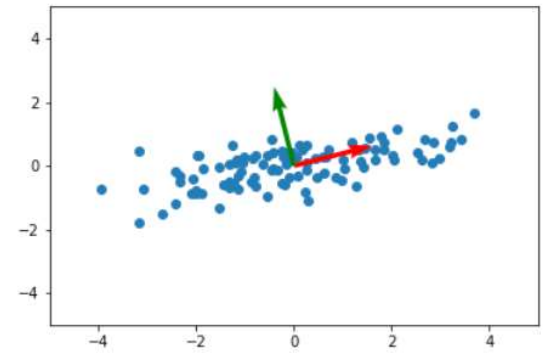


逆変換して出力を出した。サイキットラーンの PCA と比較する。

```
from sklearn.decomposition import PCA
pca = PCA(n_components=2)
pca.fit(X)
```

```
print('components: []'.format(pca.components_))
print('mean: []'.format(pca.mean_))
print('covariance: []'.format(pca.get_covariance()))
```

```
components: [[ 0.96982542  0.24380045]
 [-0.24380045  0.96982542]]
mean: [-0.02302941  0.00727137]
covariance: [[2.89463237 0.67313425]
 [0.67313425 0.38615611]]
```

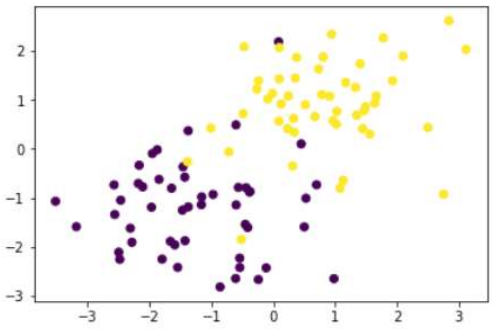


アルゴリズム実装した結果と同じ結果が得られている事が確認できる。

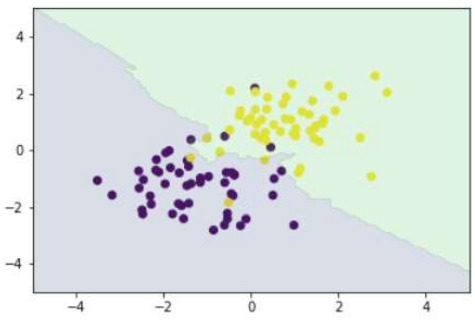
・アルゴリズム

k 近傍法（kNN:k-Nearest Neighbor）とは分類問題のための機械学習手法である。最近傍のデータを k 個取ってきて、それらが最も多く所属するクラスに識別する。kNN は怠惰学習の代表的な例であり、トレーニングデータから判別関数を学習しない事による。予測を行うにあたってモデルのトレニンーグを必要としないが予測のコストはかかるので、k 値を正しく選択するために過学習と学習不足のバランスをうまく取る事が肝心である。

```
def gen_data():
    x0 = np.random.normal(size=100).reshape(-1, 2) - 1
    x1 = np.random.normal(size=100).reshape(-1, 2) + 1.
    x_train = np.concatenate([x0, x1])
    y_train = np.concatenate([np.zeros(50), np.ones(50)]).astype(np.int)
    return x_train, y_train
```

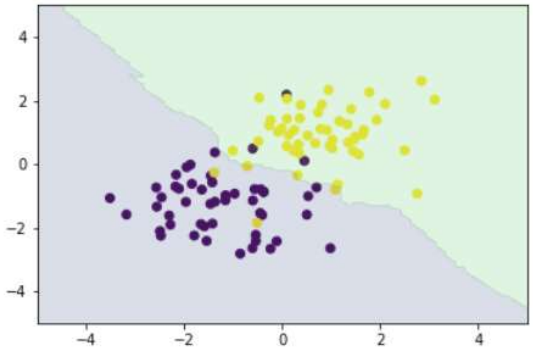


```
n_neighbors = 3

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

y_pred = knc_predict(n_neighbors, X_train, ys_train, X_test)
plt_resut(X_train, ys_train, y_pred)
```



n=3 の時は、境界面がなめらかではない。

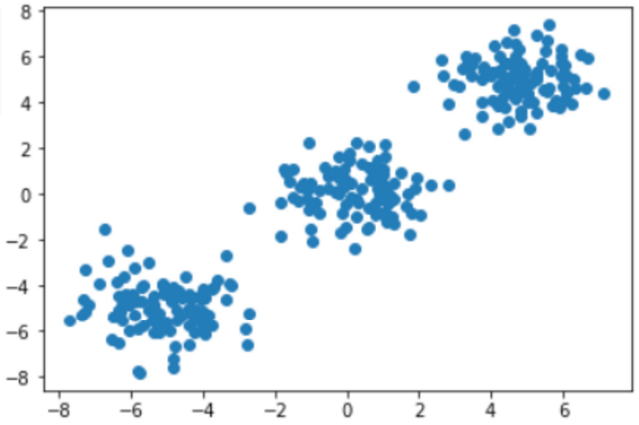
```
n_neighbors = 7
```



n= 7 にすると、境界面がなめらかになる。

k-means は教師なし学習のクラスタリング手法であり、与えられたデータを k 個のクラスタに分類する。アルゴリズムは 1.各クラスタ中心の初期値を設定する 2.各データ点に対して、各クラスタ中心との距離を計算し、最も距離が近いクラスタを割り当てる 3.各クラスタの平均ベクトル（中心）を計算する 4.収束するまで 2,3 の処理を繰り返す。注意すべき点としては、中心の初期値を変えるとクラスタリング結果も変わりうる事、k 値を変えるとクラスタリング結果も変わる事である。

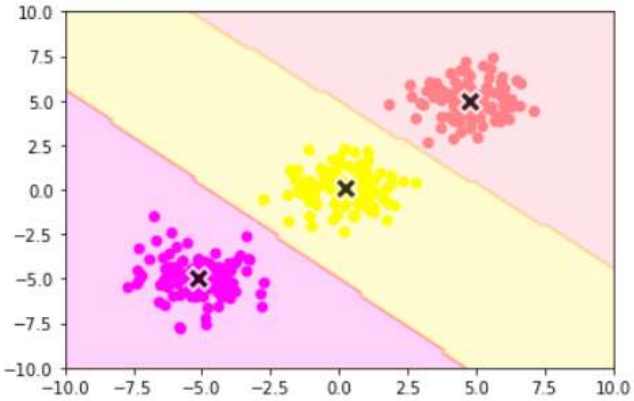
```
def gen_data():
    x1 = np.random.normal(size=(100, 2)) + np.array([-5, -5])
    x2 = np.random.normal(size=(100, 2)) + np.array([5, 5])
    x3 = np.random.normal(size=(100, 2)) + np.array([0, 0])
    return np.vstack((x1, x2, x3))
```



```
def plt_result(X_train, centers, xx):
    #  データを可視化
    plt.scatter(X_train[:, 0], X_train[:, 1], c=y_pred, cmap='spring')
    # 中心を可視化
    plt.scatter(centers[:, 0], centers[:, 1], s=200, marker='X', lw=2, c='black', edgecolor="white")
    # 領域の可視化
    pred = np.empty(len(xx), dtype=int)
    for i, x in enumerate(xx):
        d = distance(x, centers)
        pred[i] = np.argmin(d)
    plt.contourf(xx0, xx1, pred.reshape(100, 100), alpha=0.2, cmap='spring')
```

```
y_pred = np.empty(len(X_train), dtype=int)
for i, x in enumerate(X_train):
    d = distance(x, centers)
    y_pred[i] = np.argmin(d)
```

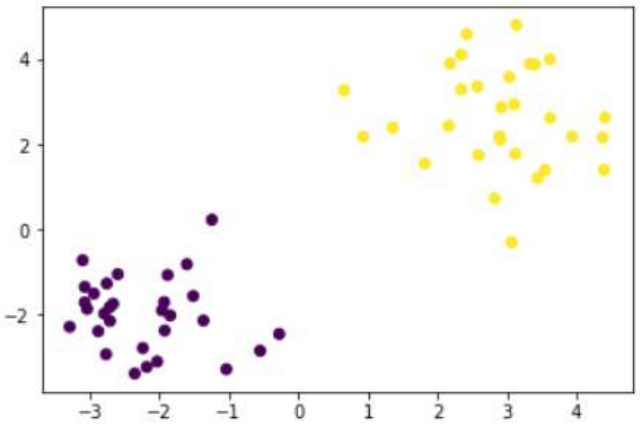
```
xx0, xx1 = np.meshgrid(np.linspace(-10, 10, 100), np.linspace(-10, 10, 100))
xx = np.array([xx0, xx1]).reshape(2, -1).T

plt_result(X_train, centers, xx)
```



・サポートベクターマシン

サポートベクターマシン（ＳＶＭ）は２クラス分類問題に使用されるアルゴリズムである。２クラス分類は与えられた入力データが２つのカテゴリーのどちらに属するかを識別する問題である。ＳＶＭの特徴は、データと識別面（分類境界）との距離、つまりマージンが最大となるようにパラメータを求める。識別面を決定するのは一部のデータでよく、このデータの事をサポートベクトルという。マージン最大化によって解が定まるのは、データが線形分離可能の場合のみで、このようなＳＶ分類の事をハードマージンと呼ぶ。しかながら、現実の問題ではその仮定は強すぎるため、誤識別に対するペナルティを導入する。このペナルティを表現するのがスラック変数であり、マージン最大化とペナルティの最小化を行うＳＶＭをソフトマージンＳＶＭと呼ぶ。

線形分離可能な場合

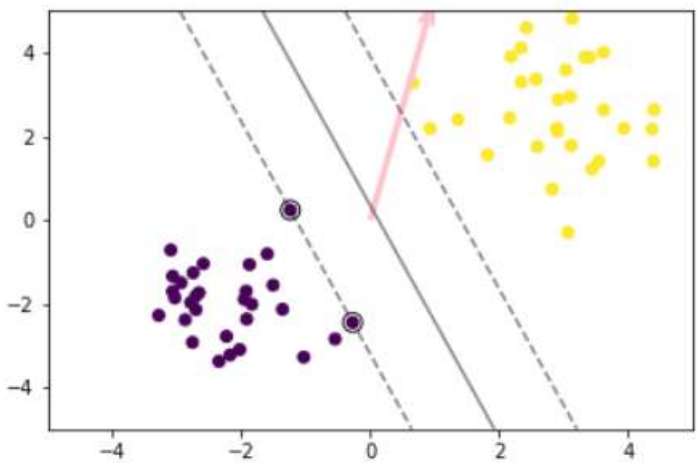


データ生成（seed を離して生成する）

```
def gen_data():
    x0 = np.random.normal(size=60).reshape(-1, 2) - 2.
    x1 = np.random.normal(size=60).reshape(-1, 2) + 3.
    X_train = np.concatenate([x0, x1])
    ys_train = np.concatenate([np.zeros(30), np.ones(30)]).astype(np.int)
    return X_train, ys_train
```

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

X_test = xx
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])
y_pred = np.sign(y_project)
```

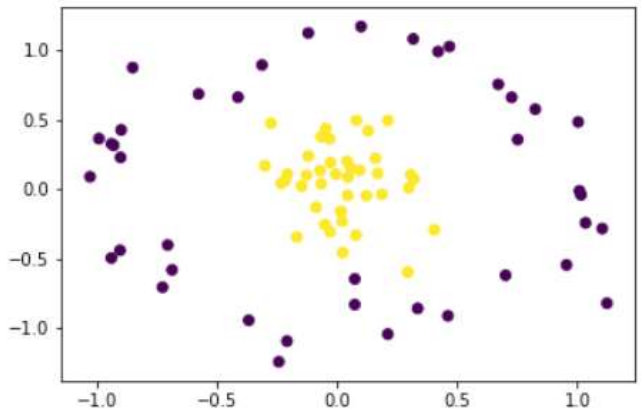
```
# 訓練データを可視化
plt.scatter(X_train[:, 0], X_train[:, 1], c=ys_train)
# サポートベクトルを可視化
plt.scatter(support_vectors[:, 0], support_vectors[:, 1],
                    s=100, facecolors='none', edgecolors='k')
# 領域を可視化
#plt.contourf(xx0, xx1, y_pred.reshape(100, 100), alpha=0.2, levels=np.linspace(0, 1, 3))
# 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=['--', '-', '--'])


# マージンと決定境界を可視化
plt.quiver(0, 0, 0.1, 0.35, width=0.01, scale=1, color='pink')
```



線形分離不可能な場合

データ群を円形に取り囲むように配置する

```
factor = .2
n_samples = 80
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

X = np.vstack((np.append(outer_circ_x, inner_circ_x),
               np.append(outer_circ_y, inner_circ_y))).T
y = np.hstack([np.zeros(n_samples // 2, dtype=np.intp),
               np.ones(n_samples // 2, dtype=np.intp)])
X += np.random.normal(scale=0.15, size=X.shape)
x_train = X
y_train = y
```



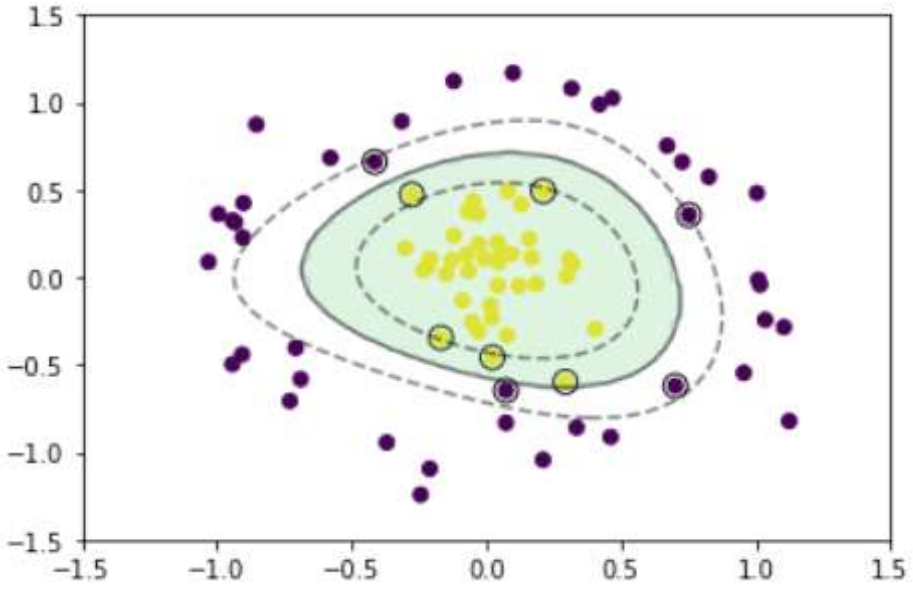
RBF カーネルを使って特徴量空間での線形分離を実施する

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

X_train = x_train
t = np.where(y_train == 1.0, 1.0, -1.0)

n_samples = len(X_train)
# RBFカーネル
K = np.zeros((n_samples, n_samples))
for i in range(n_samples):
    for j in range(n_samples):
        K[i, j] = rbf(X_train[i], X_train[j])

eta1 = 0.01
eta2 = 0.001
n_iter = 5000

H = np.outer(t, t) * K

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)
```

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

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

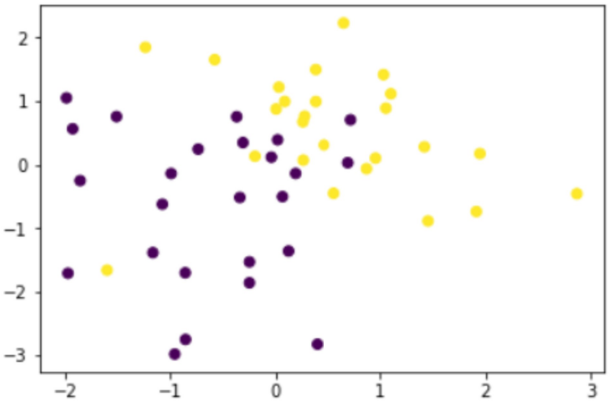
```
xx0, xx1 = np.meshgrid(np.linspace(-1.5, 1.5, 100), np.linspace(-1.5, 1.5, 100))
xx = np.array([xx0, xx1]).reshape(2, -1).T

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

```
# 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=['--', '-', '--'])
```



重なりがある場合。Seed を近づけてデータ生成

```
x0 = np.random.normal(size=50).reshape(-1, 2) - 0.5
x1 = np.random.normal(size=50).reshape(-1, 2) + 0.5
x_train = np.concatenate([x0, x1])
y_train = np.concatenate([np.zeros(25), np.ones(25)]).astype(np.int)
```



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

n_samples = len(X_train)
# 線形カーネル
K = X_train.dot(X_train.T)

C = 1
eta1 = 0.01
eta2 = 0.001
n_iter = 1000

H = np.outer(t, t) * K

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)
```

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

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

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

X_test = xx
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])
y_pred = np.sign(y_project)
```

```
# 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=['--', '-', '--'])
```