【E 資格学習レポート】機械学習

機械学習については以下 6 つの科目でレポートする

１．線形回帰モデル

・線形回帰モデル

・回帰問題を解くための機械学習モデルの一つ

・教師あり学習(教師データから学習)

・線形結合(入力とパラメータの内積)理解する

・機械学習の基本的な手法を理解し実装する。

・最小二乗法により推定理解する

・モデル数式の幾何学的意味理解

・連立方程式が行列表現

・機械学習の定義

・データの分割とモデルの汎化性能測定、学習データの使い方理解

・線形回帰モデルの導入。

・線形回帰モデルのパラメータは最小二乗法で推定

・例としての不動産価格の予測。

・線形結合。モデルのパラメータ。

・単回帰モデル（説明変数が 1 次元）直線。重回帰モデル（説明変数が多次元）曲線。

・データの分割（学習用データと検証用データ）とモデルの汎化性能。

・最尤法による回帰係数の推定

・平均二乗誤差（残差平方和）の微分。（非線形、リッジ回帰（正則化）に関しても計算自体は全く一緒。）

・損失関数の選択（平均二乗誤差は一般に外れ値に弱い)。射影行列。

・pandas、numpy、scikit-learn の使い方

・線形単回帰分析

```
#カラムを指定してデータを表示
df[['RM']].head()
```

	RM
0	6.575
1	6.421
2	7.185
3	6.998
4	7.147

重回帰分析(2 変数)

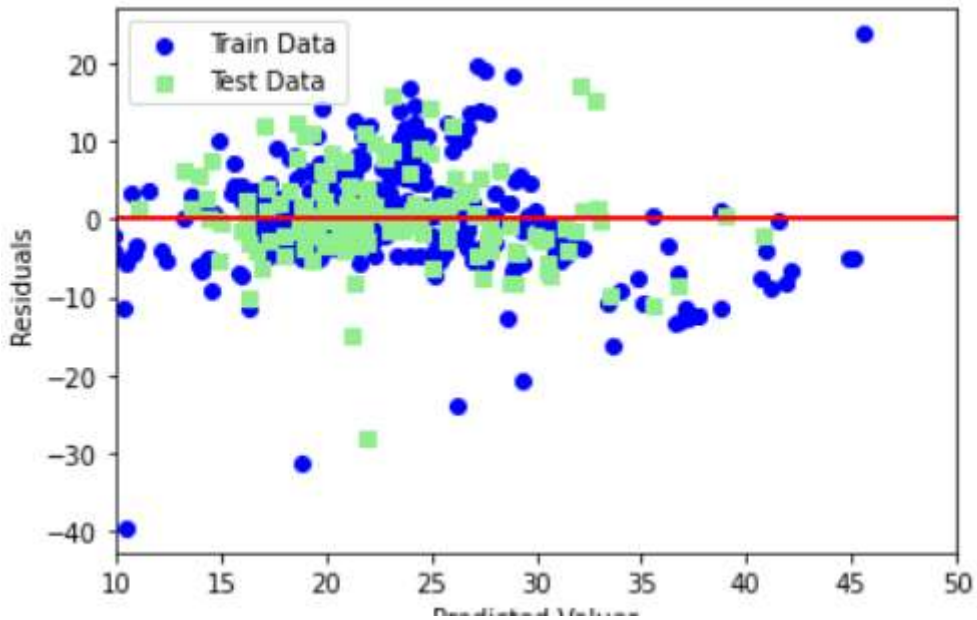
```
#カラムを指定してデータを表示
df[['CRIM', 'RM']].head()
```

	CRIM	RM
0	0.00632	6.575
1	0.02731	6.421
2	0.02729	7.185
3	0.03237	6.998
4	0.06905	7.147

参照コード

```
# matplotlib をインポート
import matplotlib.pyplot as plt
# Jupyter を利用していたら、以下のおまじないを書くと notebook 上に図が表示
%matplotlib inline
```

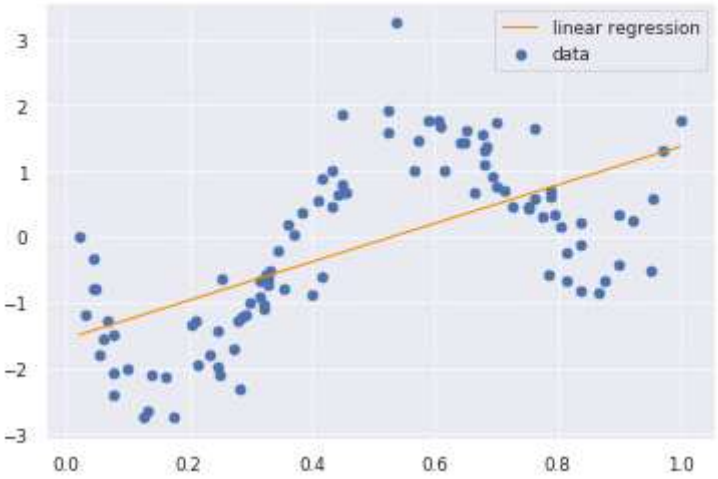
```
# 学習用、検証用それぞれで残差をプロット
plt.scatter(y_train_pred, y_train_pred - y_train, c = 'blue', marke
r = 'o', label = 'Train Data')
plt.scatter(y_test_pred, y_test_pred - y_test, c = 'lightgreen', ma
rker = 's', label = 'Test Data')
plt.xlabel('Predicted Values')
plt.ylabel('Residuals')
# 凡例を左上に表示
plt.legend(loc = 'upper left')
# y = 0 に直線を引く
plt.hlines(y = 0, xmin = -10, xmax = 50, lw = 2, color = 'red')
plt.xlim([10, 50])
plt.show()
```

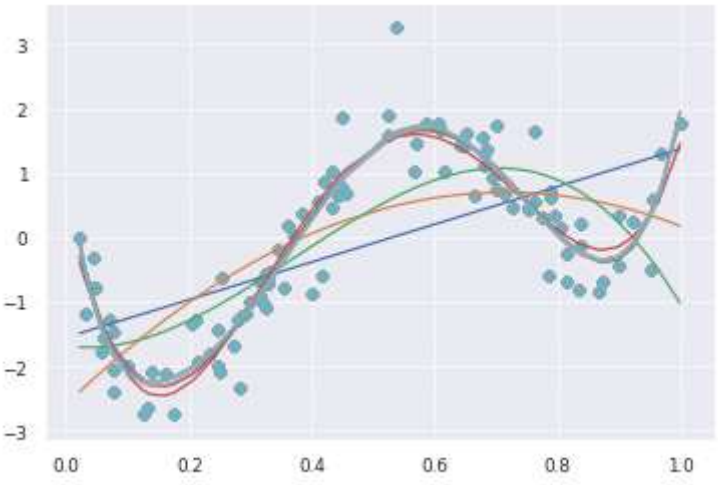


２．非線形回帰モデル

・非線形回帰モデル。基底展開法。回帰関数として、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との線形結合を使用。
・よく使われる基底関数
・基底関数には、多項式（１～９次）やガウス基底がある。基底展開法も線形回帰と同じ枠組みで推定可能。
・未学習（underfitting）と過学習（overfitting）。
・過学習への対策。学習データの数を増やす。不要な基底関数を削除。正則化法。
・罰則が無かった場合は、最小二乗推定量。L2 ノルムを利用、Ridge 推定量、縮小推定。L1 ノルムを利用、Lasso 推定量、スパース推定。
・モデルの表現力を押さえる（どの程度⇒ガンマ）基底関数の個数、位置、バンド巾そして正則化パラメータ。
・データの分割とモデルの汎化性能測定
・モデル選択。ホールドアウト法。クロスバリデーション（交差検証法）。あるモデルホールドアウトで検証したところ 70%の精度、CV で 65%だった場合でも、汎化性能の推定としては CV を利用する。

```
0.3921547168787944
```





参照コード：

```
#PolynomialFeatures(degree=1)

deg = [1,2,3,4,5,6,7,8,9,10]
for d in deg:
    regr = Pipeline([
        ('poly', PolynomialFeatures(degree=d)),
        ('linear', LinearRegression())
    ])
    regr.fit(data, target)
    # make predictions
    p_poly = regr.predict(data)
    # plot regression result
    plt.scatter(data, target, label='data')
    plt.plot(data, p_poly, label='polynomial of degree %d' % (d))
```

3．ロジスティクス回帰モデル

・「ロジスティック回帰モデル」

・分類問題（クラス分類）例：実数全体を{0,1}へ移す。

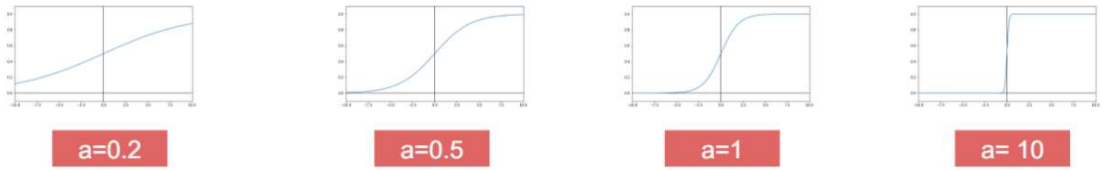
例) titanic データ、シグモイド関数判断後、(0 なら生還、1 なら死亡)

・線形回帰で使用して線形結合をそのまま「シグモイド関数」の入力にしてしまう。(そのまま回帰モデルをあてはめると上手くいかない。)

・a を増加させる事により、曲線の勾配が増加し、単位ステップ関数に近づく。

・パラメータが変わるとシグモイド関数の形が変わる。

$$\sigma(x) = \frac{1}{1+\exp(-ax)}$$



・「良い性質」を持っている：シグモイド関数を微分した時に自分自身（シグモイド関数）で表現する事ができる。

・最適化の際に有利（MSE や尤度関数の最大、最小にする点を求める：微分が必要）

シグモイド関数の出力を Y＝ 1 になる確率に対応させる。

数式

$$P(Y=1|\boldsymbol{x}) = \sigma(w_0 + w_1 x_1)$$

データが与えられた時にY=1になる確率

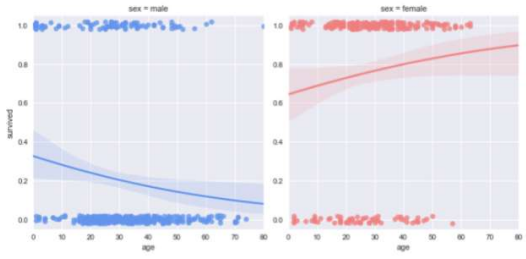
切片　回帰係数

説明変数

既知：入力データ

未知：学習で決める

幾何学的な意味



データYは確率が0.5以上ならば1・未満なら0と予測

・「最尤推定」

・ロジスティック回帰モデルではベルヌーイ分布を利用する。

・データからベルヌーイ分布のパラメータを推定。

・尤度関数を最大化するようなパラメータを選ぶ推定方法を最尤推定という。

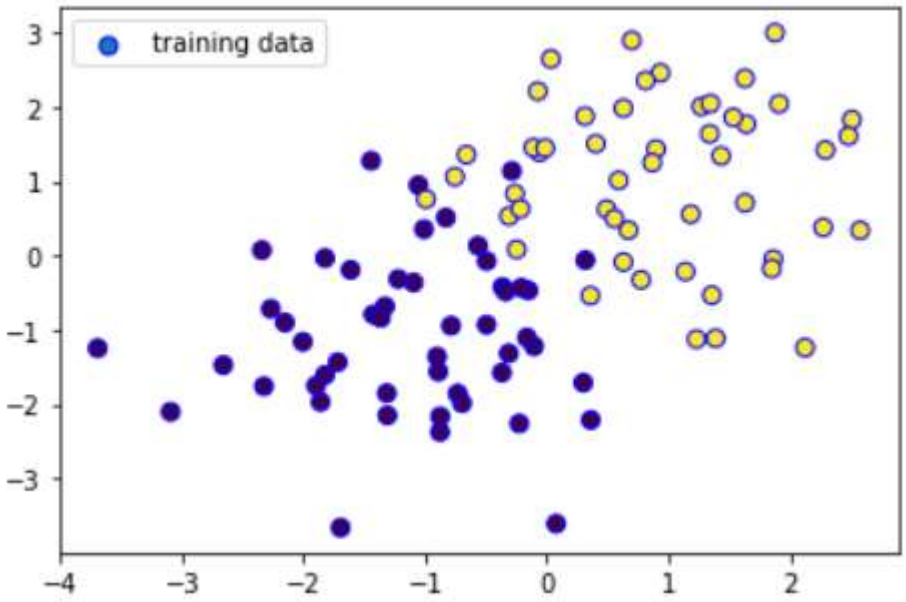
ロジスティック回帰モデルの最尤推定。

・確率 p はシグモイド関数となるため、推定するパラメータは重みパラメータ（w）と

なる。
・尤度関数はパラメータ（w）のみに依存する関数。
・尤度関数を最大とするパラメータ（w）を探す（推定）
・「マイナスをかけたもの（負の尤度関数）を最小化」し「最小二乗法の最小化」と合わせる。
・（対数をとると微分の計算が簡単）
・尤度とは確率 p の掛け算。 p は[0,1]の値を取るため、桁落ちの可能性大。
・勾配降下法（Gradient Descent）：解析的に解を求めるのは難しい。
・確率的勾配降下法（ＳＧＤ）：1 回更新するたびに全部のデータをメモリにのせられない状況がある。ミニバッチ＝１。
・「モデルの評価」
・学習済みの「ロジスティック回帰モデル」の性能を測る。
・指標について。混同行列（Confusion Matrix）。
・分類の評価方法。
・「正解率」が良く使われる。
・再現率（Recall）。「癌である人に対して、あなたは癌ではありません」というのは病気の検診の場合だと絶対に避けなければいけません。
・適合率（Precision）。「スパムと予測したものが確実にスパムである」。
Ｆ値。再現率と適合率の調和平均。

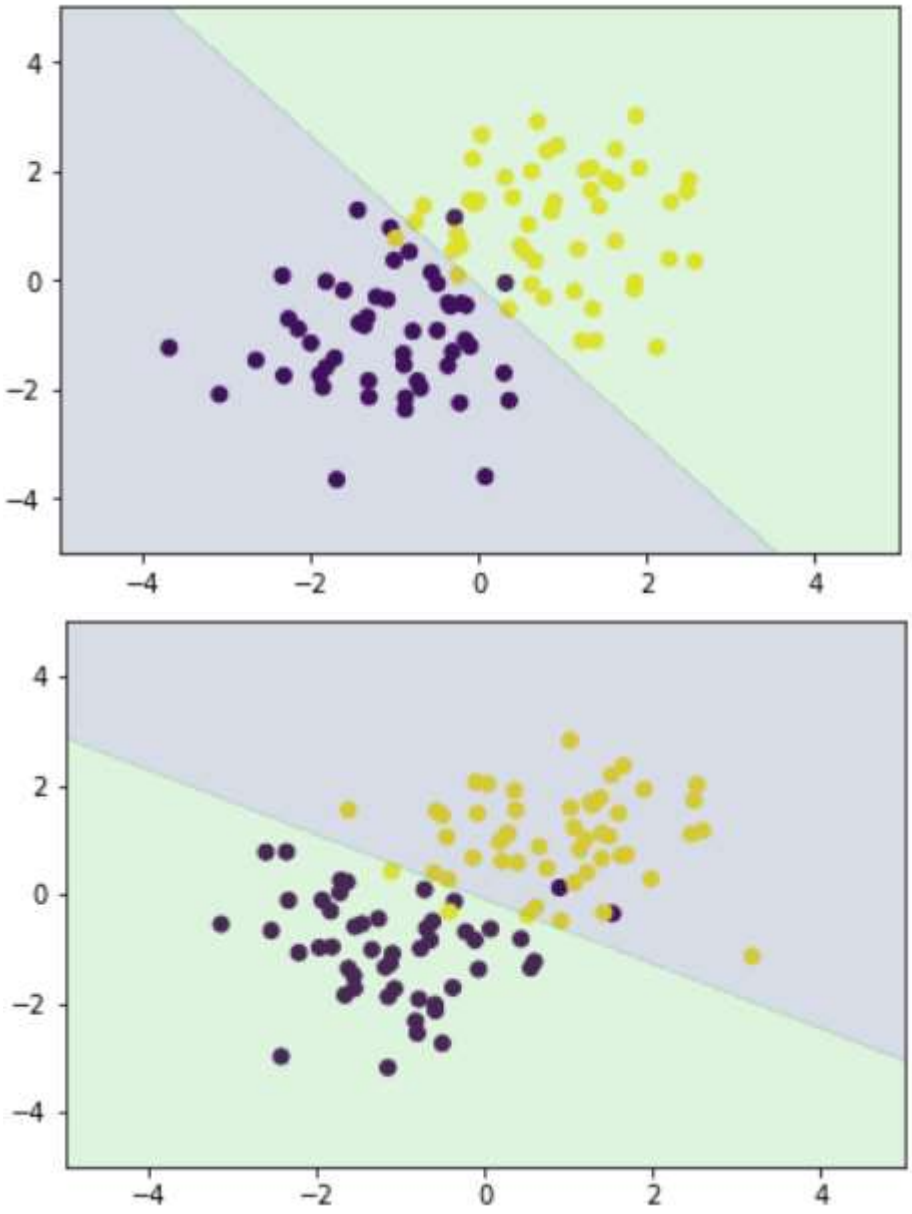
実施結果

・訓練データ生成



・学習

・予測



参照コード：

```
%matplotlib inline
import numpy as np
import matplotlib.pyplot as plt
n_sample = 100
```

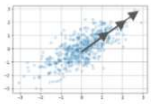
```
harf_n_sample = 50
var = .2

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

def plt_data(x_train, y_train):
    plt.scatter(x_train[:, 0], x_train[:, 1], c=y_train, facecolor=
"none", edgecolor="b", s=50, label="training data")
    plt.legend()
#データ作成
x_train, y_train = gen_data(n_sample, harf_n_sample)
#データ表示
plt_data(x_train, y_train)
def add_one(x):
    return np.concatenate([np.ones(len(x))[:, None], x], axis=1)
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)
xx0, xx1 = np.meshgrid(np.linspace(-5, 5, 100), np.linspace(-
5, 5, 100))
xx = np.array([xx0, xx1]).reshape(2, -1).T
plt.scatter(x_train[:, 0], x_train[:, 1], c=y_train)
plt.contourf(xx0, xx1, proba.reshape(100, 100), alpha=0.2, levels=n
p.linspace(0, 1, 3))
from sklearn.linear_model import LogisticRegression
model=LogisticRegression(fit_intercept=True)
model.fit(x_train, y_train)
proba = model.predict_proba(xx)
y_pred = (proba > 0.5).astype(np.int)
plt.scatter(x_train[:, 0], x_train[:, 1], c=y_train)
plt.contourf(xx0, xx1, proba[:, 0].reshape(100, 100), alpha=0.2, le
vels=np.linspace(0, 1, 3))
```

４．主成分分析

・次元圧縮（多変量データの持つ構造をより小数個の指標に圧縮)。
変量の個数を減らすことに伴う、情報の損失はなるべく小さくしたい。
少数変数を利用した分析や可視化(2・3 次元の場合)が実現可能相関 0.8 の正規分布から生成したデータ 2 次元を 1 次元に圧縮主成分分析。

- 以下の制約付き最適化問題を解く
  - ノルムが1となる制約を入れる(制約を入れないと無限に解がある)



目的関数 $\arg\max_{\boldsymbol{a}\in\mathbb{R}^m} \boldsymbol{a}_j^T Var(\bar{X})\boldsymbol{a}_j$ 制約条件 $\boldsymbol{a}_j^T\boldsymbol{a}_j = 1$

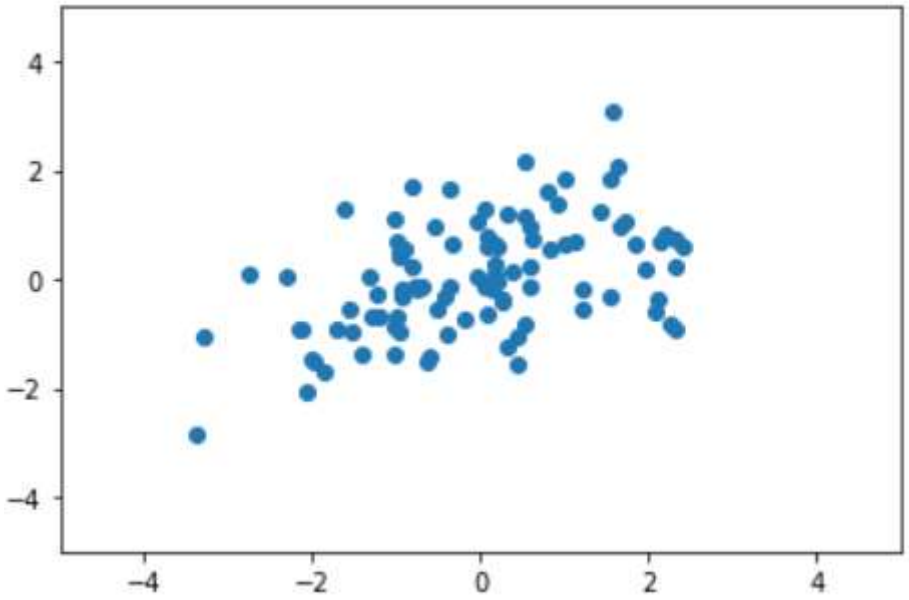
- 制約つき最適化問題の解き方
  - ラグランジュ関数を最大にする係数ベクトルを探索 (微分して0になる点)

ラグランジュ関数 $E(\boldsymbol{a}_j) = \boldsymbol{a}_j^T Var(\bar{X})\boldsymbol{a}_j - \lambda(\boldsymbol{a}_j^T\boldsymbol{a}_j - 1)$

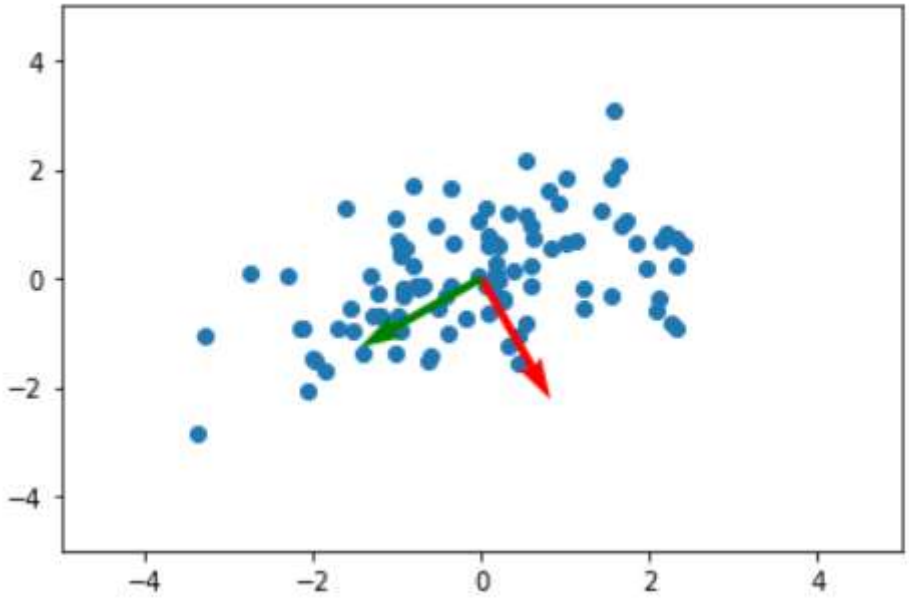
ラグランジュ乗数　目的関数　制約条件

・線形変換後の変数の分散（情報の量）が最大となる射影軸を探索。
・線形変換後の分散は分散共分散行列と a に関する 2 次形式で書ける (a は係数ベクトル)。
・ラグランジュ関数を最大にする係数ベクトルを探索する事によって、制約つき最適化問題が解ける。
・主成分分析（ＰＣＡ）の結果、元のデータの分散共分散行列の固有値と固有ベクトルを求めれば、それが分散を最大にする軸になっている。
・寄与率：第Ｋ主成分が持つ情報量の割合。
・累積寄与率：第 1 - Ｋ主成分まで圧縮した際の情報損失量の割合。

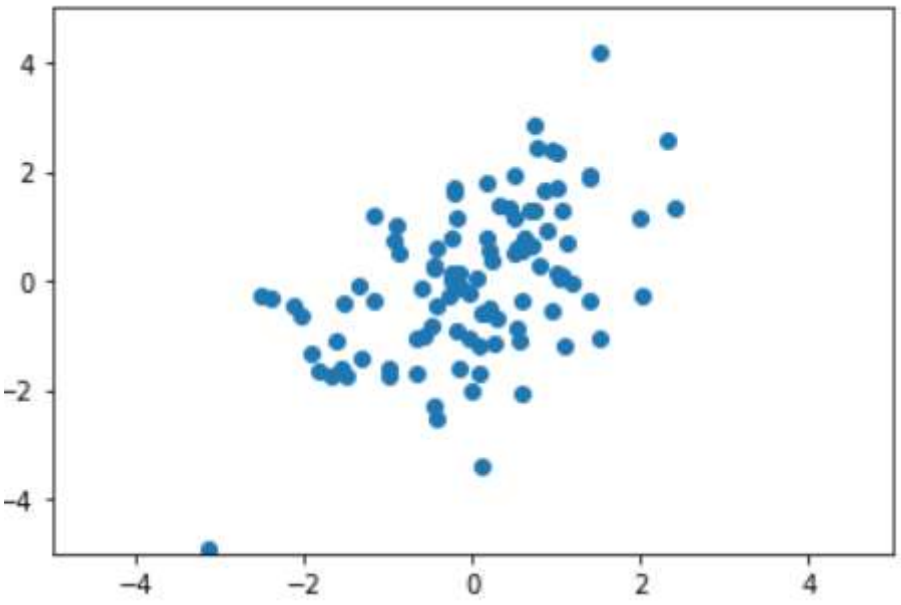
・訓練データ



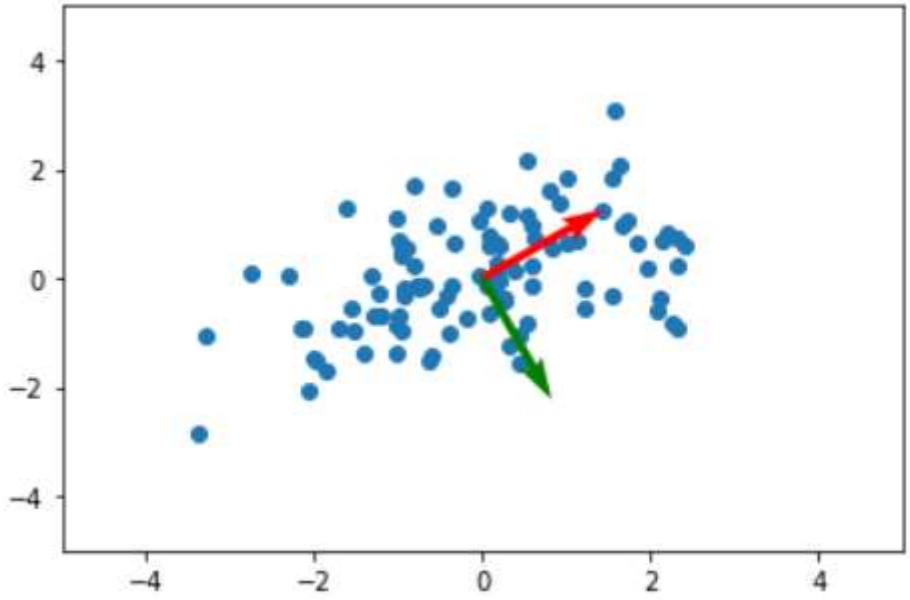
・学習



・射影



・逆変換



参照コード：

```
%matplotlib inline
import numpy as np
import matplotlib.pyplot as plt
n_sample = 100

def gen_data(n_sample):
```

```
    mean = [0, 0]
    cov = [[2, 0.7], [0.7, 1]]
    return np.random.multivariate_normal(mean, cov, n_sample)

def plt_data(X):
    plt.scatter(X[:, 0], X[:, 1])
    plt.xlim(-5, 5)
    plt.ylim(-5, 5)
X = gen_data(n_sample)
plt_data(X)
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, col
or='green')
#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, :])
ef transform_by_pca(X, pca):
    mean = X.mean(axis=0)
    return np.dot(X-mean, components)
Z = transform_by_pca(X, components.T)
plt.scatter(Z[:, 0], Z[:, 1])
plt.xlim(-5, 5)
plt.ylim(-5, 5)
mean = X.mean(axis=0)
X_ = np.dot(Z, components.T) + mean
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()))
from sklearn.decomposition import PCA
pca = PCA(n_components=2)
pca.fit(X)
plt_result(X, pca.components_[0, :], pca.components_[1, :])

print('components: {}'.format(pca.components_))
print('mean: {}'.format(pca.mean_))
print('covariance: {}'.format(pca.get_covariance()))
```

５．アルゴリズム

・「ｋ近傍法（ｋＮＮ）」

・k=1 は(最近傍法)と同じ

・分類問題のための機械学習手法

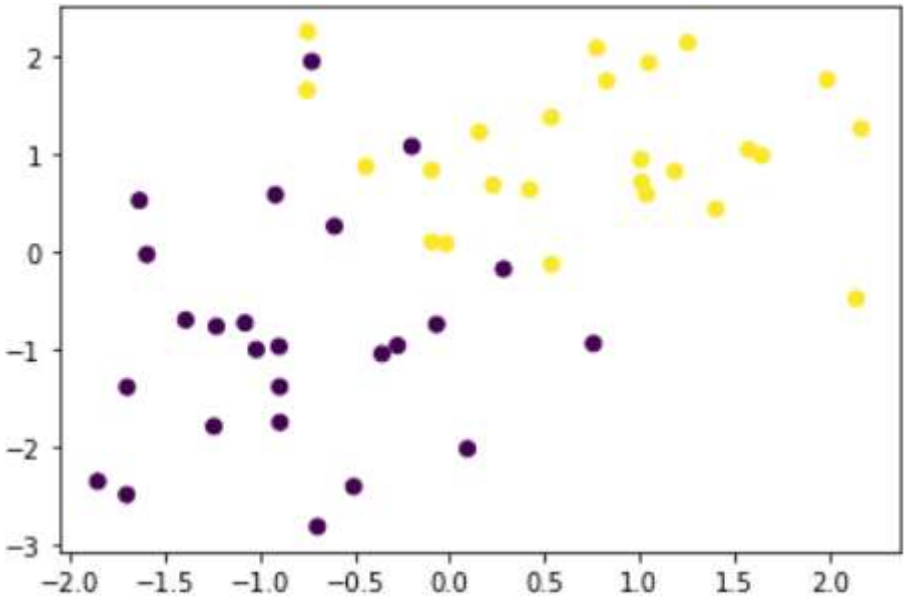
・分類したい点の近傍からＫ個を取ってきて、それらがももっとも多く所属するクラスに識別。

・Ｋを大きくすると決定境界は滑らかになる。

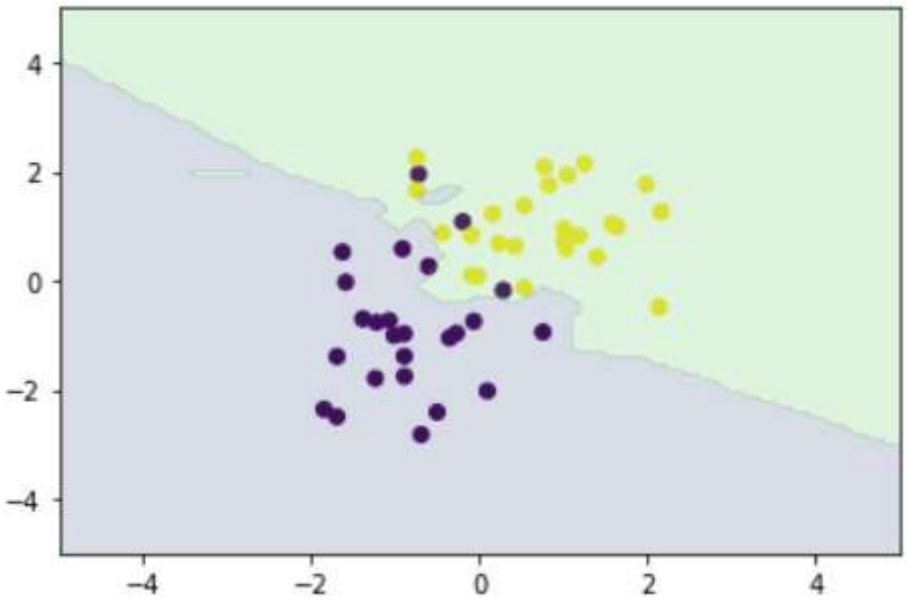
・陽に訓練ステップはない（学習不要、重要）

・ｋ近傍法、ｋ平均法共に距離の計算が必要

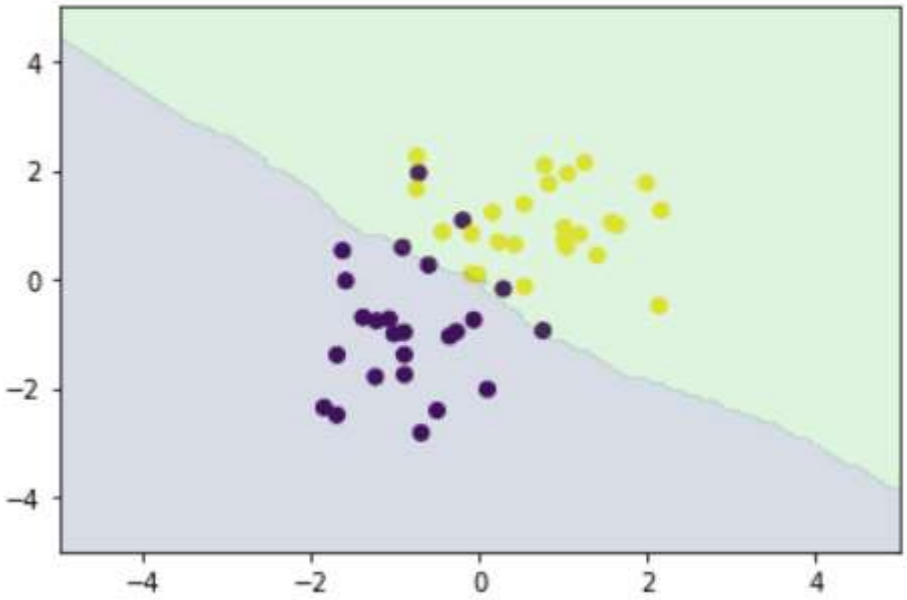
・訓練データ



KNN=3(予測)



KNN=15(予測)　⇒KNN=3 より分界線が直線に近くになる



参照コード：

```
%matplotlib inline
import numpy as np
import matplotlib.pyplot as plt
from scipy import stats
//訓練データ生成
```

```
def gen_data():
    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)
    return x_train, y_train
X_train, ys_train = gen_data()
plt.scatter(X_train[:, 0], X_train[:, 1], c=ys_train)
```

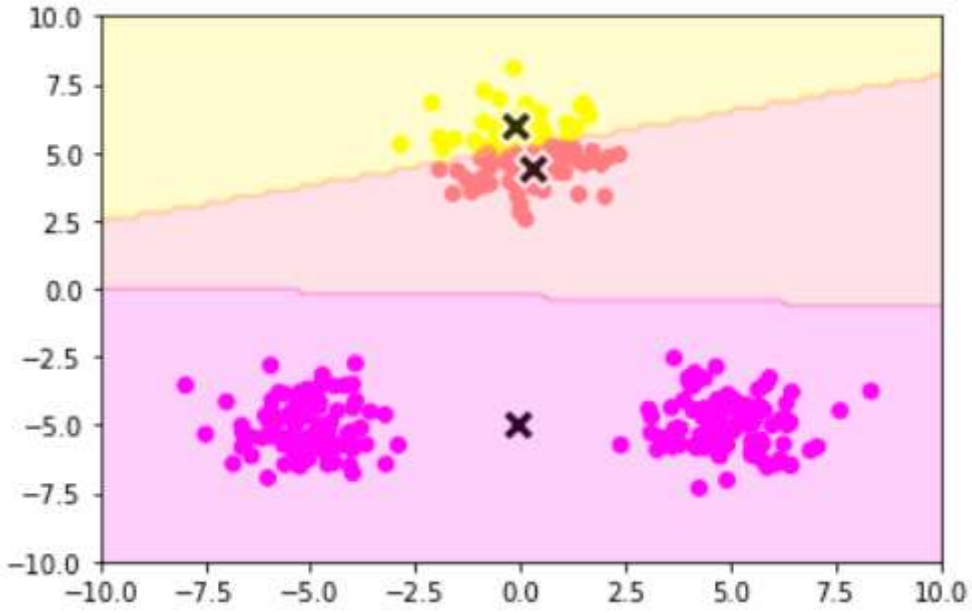
//予测

```
def distance(x1, x2):
    return np.sum((x1 - x2)**2, axis=1)

def knc_predict(n_neighbors, x_train, y_train, X_test):
    y_pred = np.empty(len(X_test), dtype=y_train.dtype)
    for i, x in enumerate(X_test):
        distances = distance(x, X_train)
        nearest_index = distances.argsort()[:n_neighbors]
        mode, _ = stats.mode(y_train[nearest_index])
        y_pred[i] = mode
    return y_pred

def plt_resut(x_train, y_train, y_pred):
    xx0, xx1 = np.meshgrid(np.linspace(-5, 5, 100), np.linspace(-
5, 5, 100))
    xx = np.array([xx0, xx1]).reshape(2, -1).T
    plt.scatter(x_train[:, 0], x_train[:, 1], c=y_train)
    plt.contourf(xx0, xx1, y_pred.reshape(100, 100).astype(dtype=np
.float), alpha=0.2, levels=np.linspace(0, 1, 3))

n_neighbors = 15

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

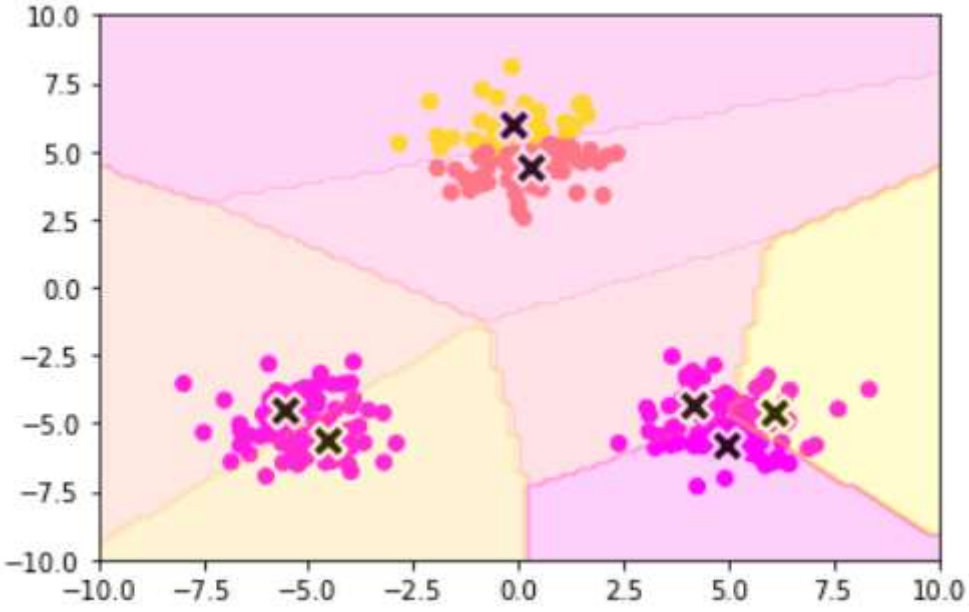
```
y_pred = knc_predict(n_neighbors, X_train, ys_train, X_test)
plt_resut(X_train, ys_train, y_pred)
```

・「ｋ平均法（ｋ－ｍｅａｎｓ）」
・教師なし学習
・k 平均法(k-means)のアルゴリズム
  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
・クラスタリング手法
・与えられたデータをｋ個のクラスタに分類する。
・ｋ平均法のアルゴリズム
・ｋはチューニングパラメータ。ｋの値を変えるとクラスタリングの結果も変わる。
・中心の初期値を変えるとクラスタリング結果も変わる。
・ｋ－ｍｅａｎｓ＋＋は１個目の初期値を置くと、その初期値からなるべく遠くなるように（分散や確率の概念を利用して）次の初期値を置く。

・クラスタ＝３の結果（実施）



・クラスタ＝7 の結果（実施）



参照コード：

```
def distance(x1, x2):
    return np.sum((x1 - x2)**2, axis=1)

n_clusters = 7
```

```
iter_max = 100

# 各クラスタ中心をランダムに初期化
centers = X_train[np.random.choice(len(X_train), n_clusters, replac
e=False)]

for _ in range(iter_max):
    prev_centers = np.copy(centers)
    D = np.zeros((len(X_train), n_clusters))
    # 各データ点に対して、各クラスタ中心との距離を計算
    for i, x in enumerate(X_train):
        D[i] = distance(x, centers)
    # 各データ点に、最も距離が近いクラスタを割り当
    cluster_index = np.argmin(D, axis=1)
    # 各クラスタの中心を計算
    for k in range(n_clusters):
        index_k = cluster_index == k
        centers[k] = np.mean(X_train[index_k], axis=0)
    # 収束判定
    if np.allclose(prev_centers, centers):
        break

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='sprin
g')
    # 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)
```

６．サポートベクターマシーン

・サポートベクターマシン（ＳＶＭ）

サポートベクトルマシン(SVM)は、1990 年代に提案され、2000 年代前半に急速に発展した手法です

・２クラス分類のための機械学習手法

・線形モデルの正負で２値分類

・線形サポートベクトル分類(ハードマージン)

・線形サポートベクトル分類(ソフトマージン)

♣ SVM における双対表現

ここまでの議論を整理しましょう。結局、線形 SV 分類を用いて分類境界を決定する問題は、ハードマージンの場合には

$$\min_{w,b} \frac{1}{2}\|\boldsymbol{w}\|^2, \quad y_i\left[\boldsymbol{w}^{\mathrm{T}}\boldsymbol{x}_i + b\right] \geq 1 \quad (i = 1, \cdots, n)$$

ソフトマージンの場合には、

$$\min_{w,b,\xi}\left[\frac{1}{2}\|\boldsymbol{w}\|^2 + C\sum_{i=1}^{n}\xi_i\right], \quad y_i\left[\boldsymbol{w}^{\mathrm{T}}\boldsymbol{x}_i + b\right] \geq 1 - \xi_i, \quad \xi_i \geq 0 \quad (i = 1, \cdots, n)$$

・線形判別関数と最も近いデータ点との距離をマージンと言う。

マージンが最大化となる線形判別関数を求める。

・目的関数の導出。

・ＳＶＭの主問題（主問題の目的関数と制約条件）

・上の最適化問題をラグランジュの未定乗数法で解く。

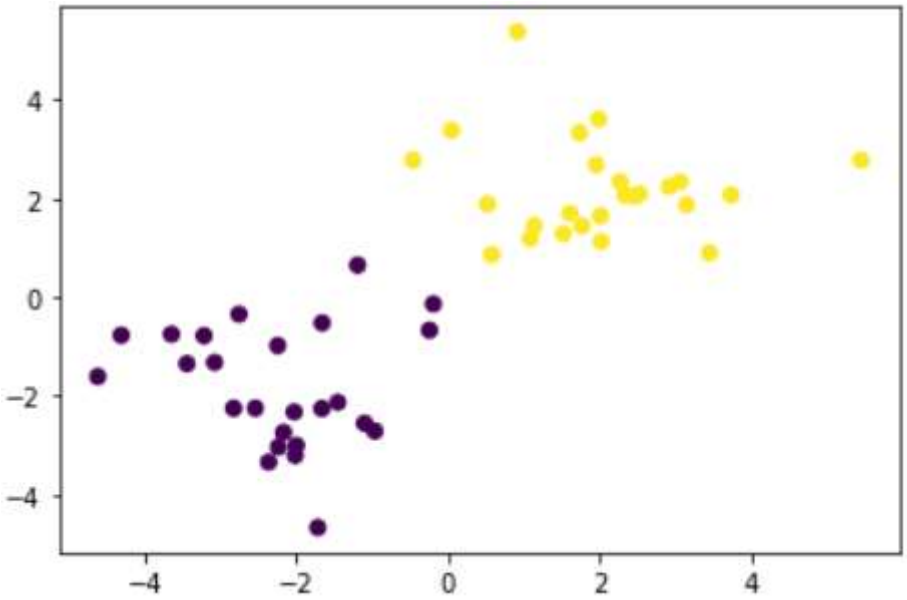
・ＫＫＴ条件（制約付き最適化問題において最適解が満たす条件）

・双対問題

主問題と双対問題（主問題の最適解と双対問題の最適解は１対１対応）

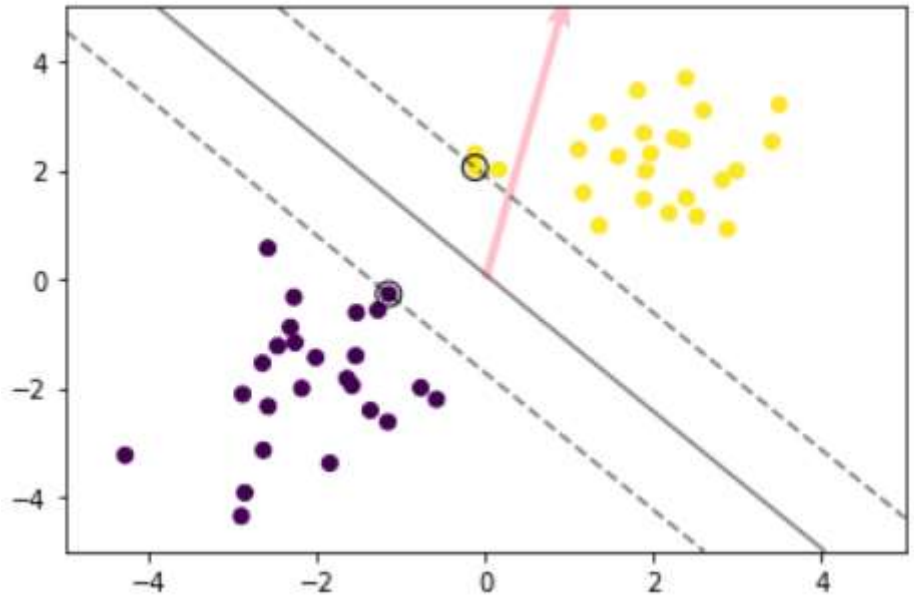
線形分離可能実施

・訓練データ生成



・学習

・予測



参照コード：

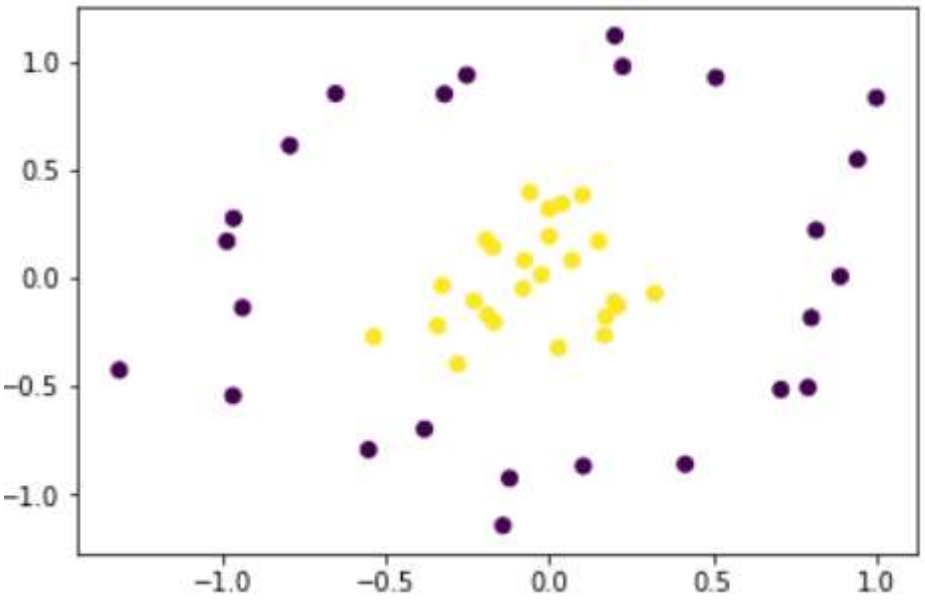
```
%matplotlib inline
import numpy as np
import matplotlib.pyplot as plt
def gen_data():
    x0 = np.random.normal(size=50).reshape(-1, 2) - 2.
```

```
    x1 = np.random.normal(size=50).reshape(-1, 2) + 2.
    X_train = np.concatenate([x0, x1])
    ys_train = np.concatenate([np.zeros(25), np.ones(25)]).astype(n
p.int)
    return X_train, ys_train
X_train, ys_train = gen_data()
plt.scatter(X_train[:, 0], X_train[:, 1], c=ys_train)
t = np.where(ys_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)

eta1 = 0.01
eta2 = 0.001
n_iter = 500

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(-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, supp
ort_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')
```

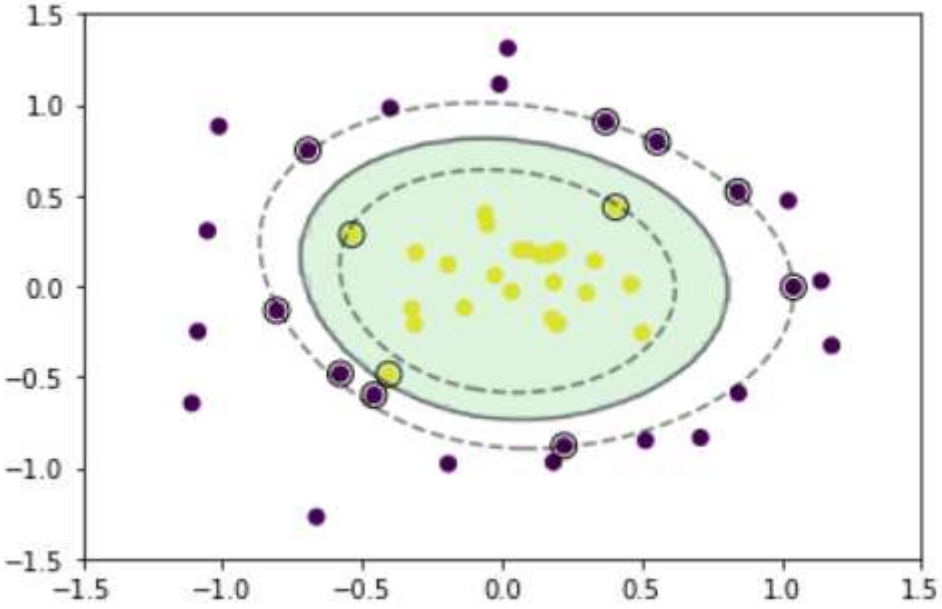
線形分離不可能実施

・訓練データ生成



・学習

・予測



参照コード：

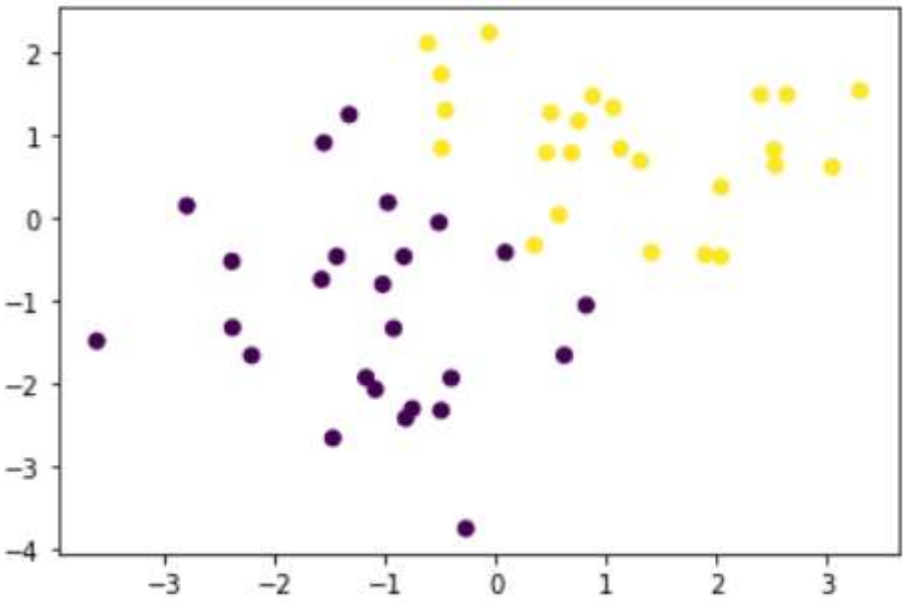
```
factor = .2
n_samples = 50
linspace = np.linspace(0, 2 * np.pi, n_samples // 2 + 1)[:-1]
outer_circ_x = np.cos(linspace)
```

```
outer_circ_y = np.sin(linspace)
inner_circ_x = outer_circ_x * factor
inner_circ_y = outer_circ_y * factor

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
plt.scatter(x_train[:,0], x_train[:,1], c=y_train)
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, supp
ort_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=['--
', '-', '--'])
```

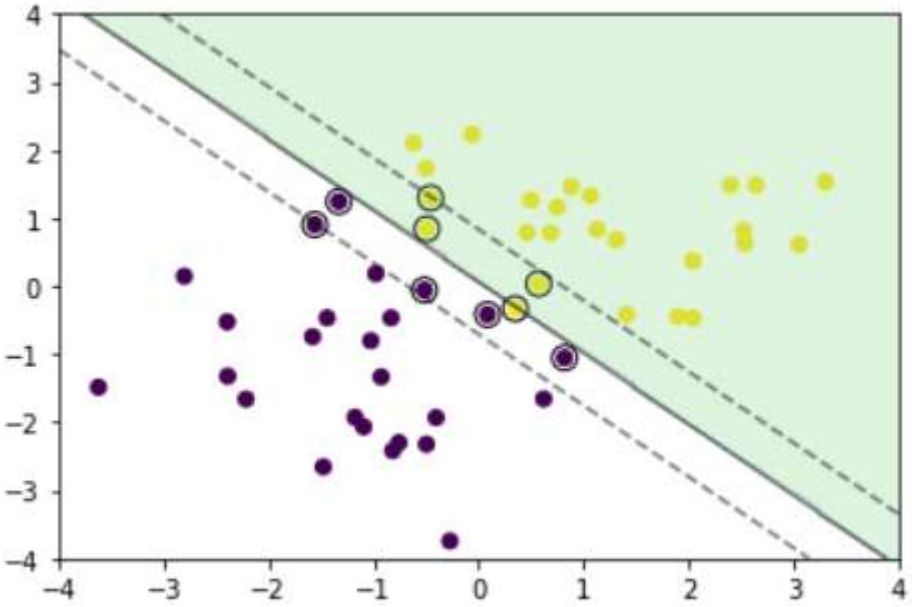
ソフトマージン SVM(重なりあり)

・訓練データ生成



・学習

・予測



参照コード

```
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
)
plt.scatter(x_train[:, 0], x_train[:, 1], c=y_train)
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, supp
ort_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=['--
', '-', '--'])
```

参考資料

YouTube（Able Programming）機械学習をはじめよう

https://www.youtube.com/playlist?list=PLdG31GUo-My_YlVF8BClMDBaJ0IJyEFjS

【機械学習】AI とは？ | 機械学習と AI の関係/機械学習入門

【機械学習】機械学習入門 / k 最近傍法 | 機械学習の手順と基本的なアルゴリズム

【機械学習】線形回帰（前編）| 線形回帰の理論

【機械学習】線形回帰（後編）| 重回帰と正則化

【機械学習】ロジスティック回帰（前編）| ロジスティック回帰の理論と実装

【機械学習】ロジスティック回帰（後編）| 多項ロジスティック回帰

【機械学習】サポートベクトルマシン（前編）| SVM の理論、ハードマージンとソフトマージン

【機械学習】サポートベクトルマシン（中編）| ラグランジュの未定乗数法、双体問題

【機械学習】サポートベクトルマシン（後編）| カーネル法、多クラス分類・回帰問題

【機械学習】決定木（CART）| 決定木の理論と実装

【機械学習】アンサンブル学習（前編）| バギング・スタッキング・バンピング、ランダムフォレスト

【機械学習】アンサンブル学習（後編）| AdaBoost、勾配ブースティング
【機械学習】モデルの評価と選択 | 交差検証、さまざまな評価基準
【機械学習】モデルの改良と前処理 | スケーリング、グリッドサーチ
【機械学習】次元削減 | 教師なし学習、主成分分析
【機械学習】クラスター分析 | 階層的クラスタリング、k-means クラスタリング
これでわかる！！Jupyter Notebook 超入門