教師あり学習（その他）

Part 1

4日目 パーセプトロン

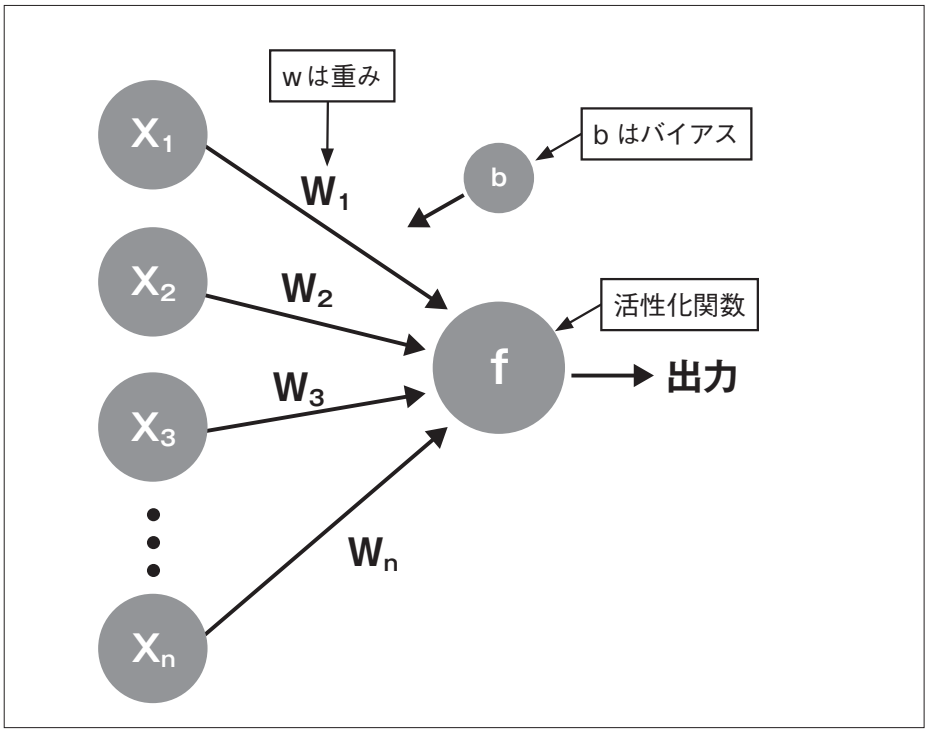
機械学習の入門では、「分離問題」を解決するアルゴリズムとして「ロジスティック回帰」や「サポートベクターマシン」「ニューラルネットワーク」などが紹介されますが、どのアルゴリズムにおいても「パーセプトロン」という概念が基本にあるようです。そこで、まずは「パーセプトロンとは何か」から調べることにしました。

パーセプトロンには、「単純パーセプトロン」と「多層パーセプトロン」の2種類があります。「多層パーセプトロン」の方は、ニューラルネットワークやディープラーニングへつながる技術らしいので明日やるとして、今日は「単純パーセプトロン」を攻略することにしましょう。

▶単純パーセプトロン

単純パーセプトロンは、学習データをクラス1かクラス2（1か0）のどちらかに分類するアルゴリズムです。このような分類方法を「2クラス分類」（2値分類）と言います（**図1**）。

図1●単純パーセプトロン



単純パーセプトロンは、入力に対する出力が教師データと異なる場合、「重み」を変更してから次の入力へ進みます。もし、出力が教師データと一致した場合、重みはそのままで

す。これを、繰り返して最適な重みを見つけるのが単純パーセプトロンの学習です。

▶バイアス

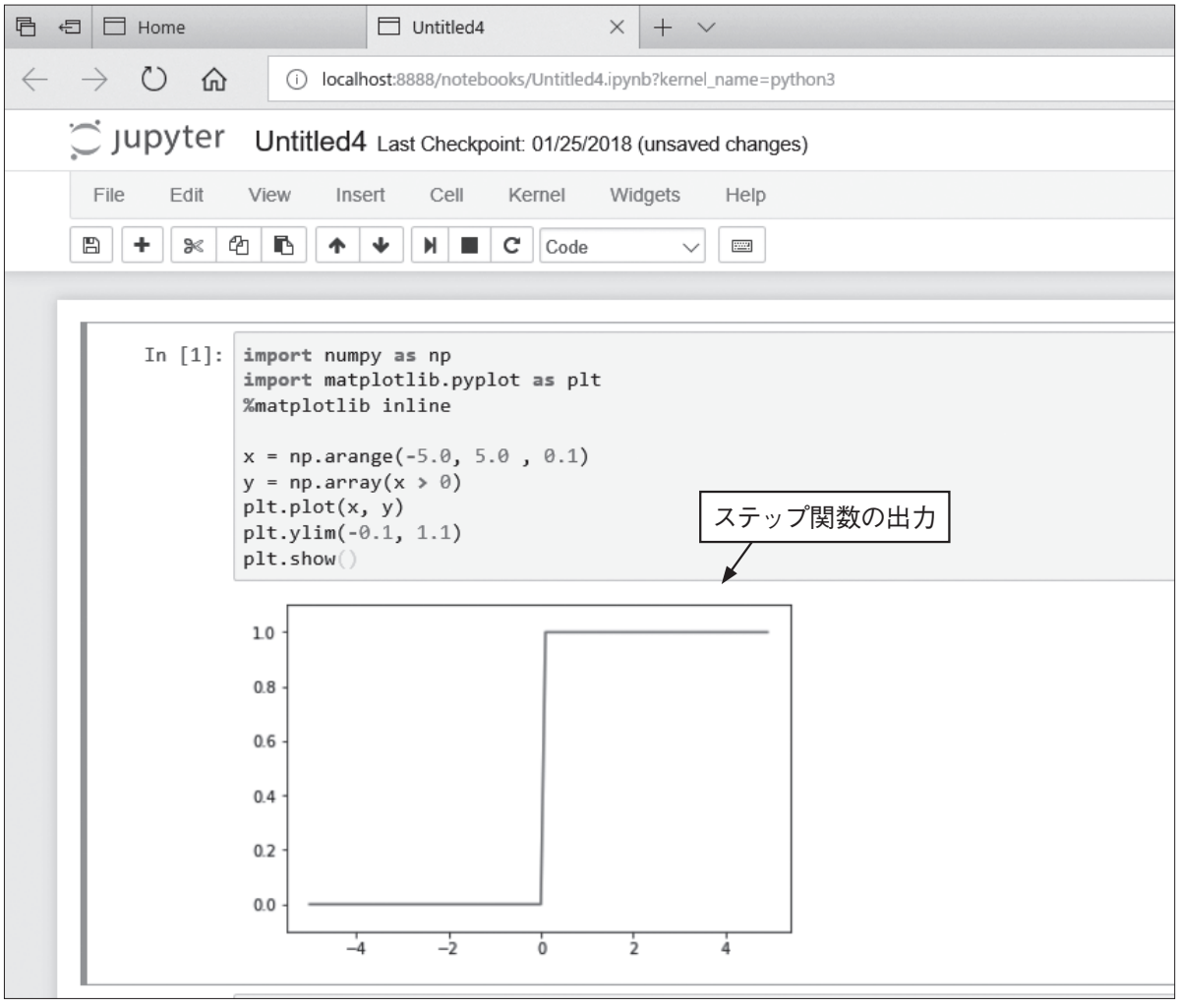
単純パーセプトロンには、入力とひも付く「重み」がありますが、実際の計算では重みに「バイアス」という値を加えて、重みを特定の方向に偏らせることができます。このバイアス値も、学習中に調整されます。

▶活性化関数

活性化関数は、入力信号の総和がいくつになったら出力するのかを決める関数です。一般的に、「単純パーセプトロン」は1か0を出力するので、活性化関数には「ステップ関数」が使われます。

ステップ関数をPythonで実装してmatplotlibでグラフにしてみます（**図2**）。

図2●ステップ関数



```
In [1]: import numpy as np
        import matplotlib.pyplot as plt
        %matplotlib inline

        x = np.arange(-5.0, 5.0 , 0.1)
        y = np.array(x > 0)
        plt.plot(x, y)
        plt.ylim(-0.1, 1.1)
        plt.show()
```

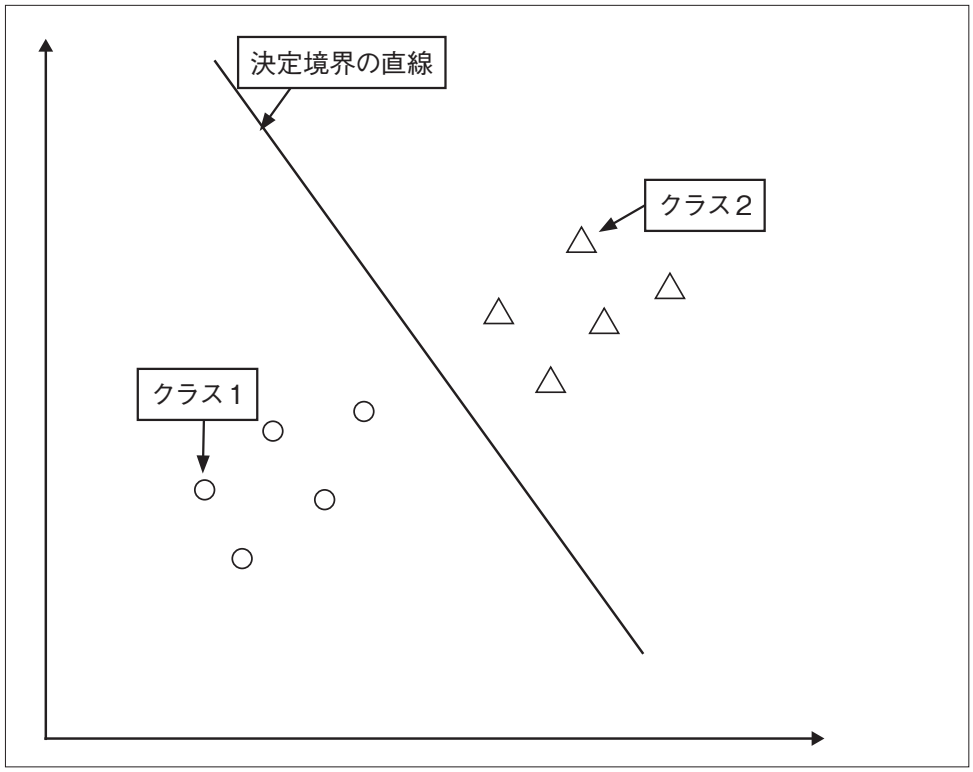
▶単純パーセプトロンの学習方法

もう少し、単純パーセプトロンがどのようにクラスを分離するのか説明します。

図3のような、クラス1とクラス2を2分する直線のあるグラフを考えます。このような直線を「決定境界」、直線でクラスを分割できることを「線形分離可能」と言います。

決定境界の直線は、未知のデータが入力されたとき、そのデータがクラス1に分類されるのか、クラス2に分類されるのかを決めます。この、決定境界の直線を引くには、「重み

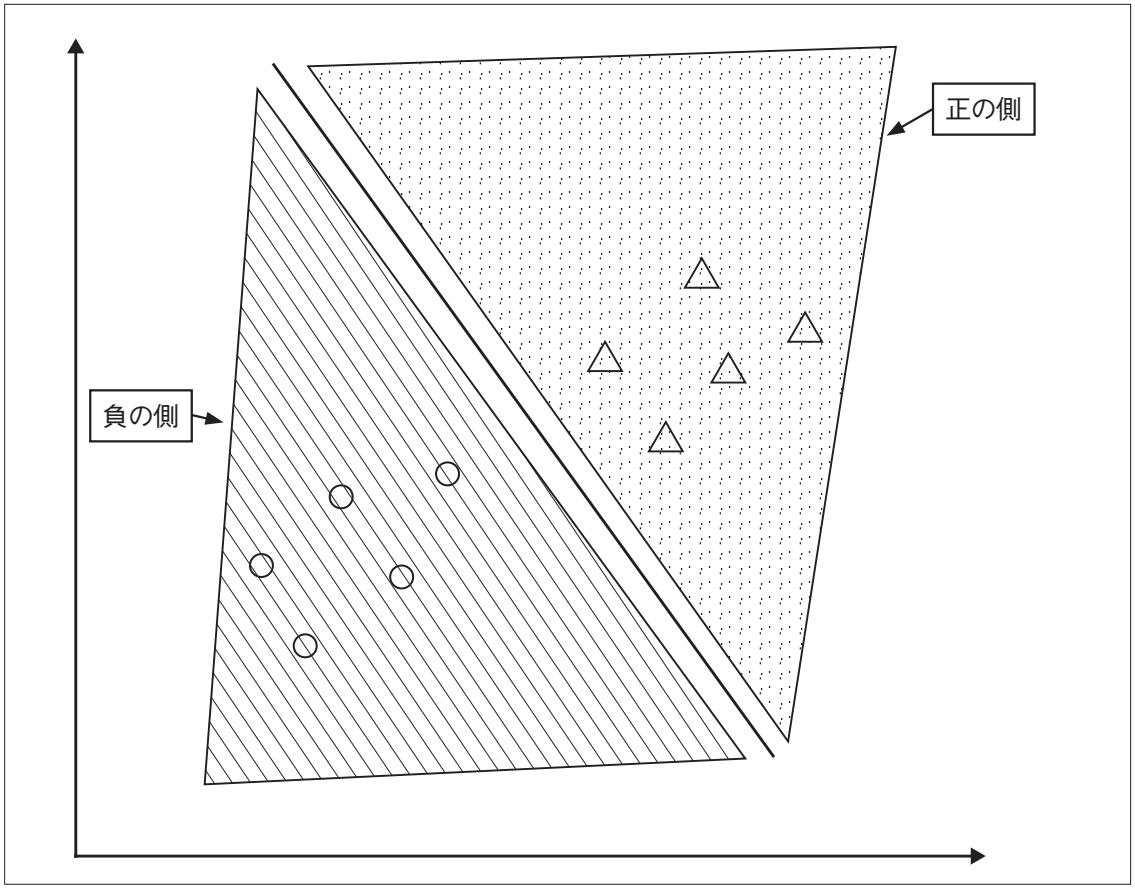
図3●線形分離可能



ベクトル係数」を求める必要があります。

重みベクトル係数の求め方は、まず、最適な決定境界が引かれているとして、直線を境に正の側と負の側に分けます（**図4**）。このように、正の側にあるのか負の側にあるのか

図4●正と負に分ける



を判断する関数を「識別関数」と言います。そのため、決定境界を「識別面」と呼ぶ場合もあります。次に、負の側のデータを反転移動して正の側に移動します（**図5**）。

このように、片方のデータを反転移動しても決定境界自体に影響はありません。つまり、クラス1のデータを負のデータ、クラス2のデータを正のデータとする決定境界は、クラス1のデータを正側に反転させて「重みベクトルを法線ベクトルとする直線」を求めれば良いことになります（**図6**）。

この「重みベクトルを法線ベクトルとする直線」を引くために、重みベクトルの係数wを求めます。この係数は決定境界を導き出すための「パラメータ」とも呼ばれます。

それでは、単純パーセプトロンで決定境界を得るためにどのようにして重みベクトルの係数を得るのか説明します。ここでは、ラベル1のデータをクラス1（決定境界の負の側）に、ラベル2のデータをクラス2（決定境界の正の側）として分離することにします。

図５●負の側のデータを反転移動

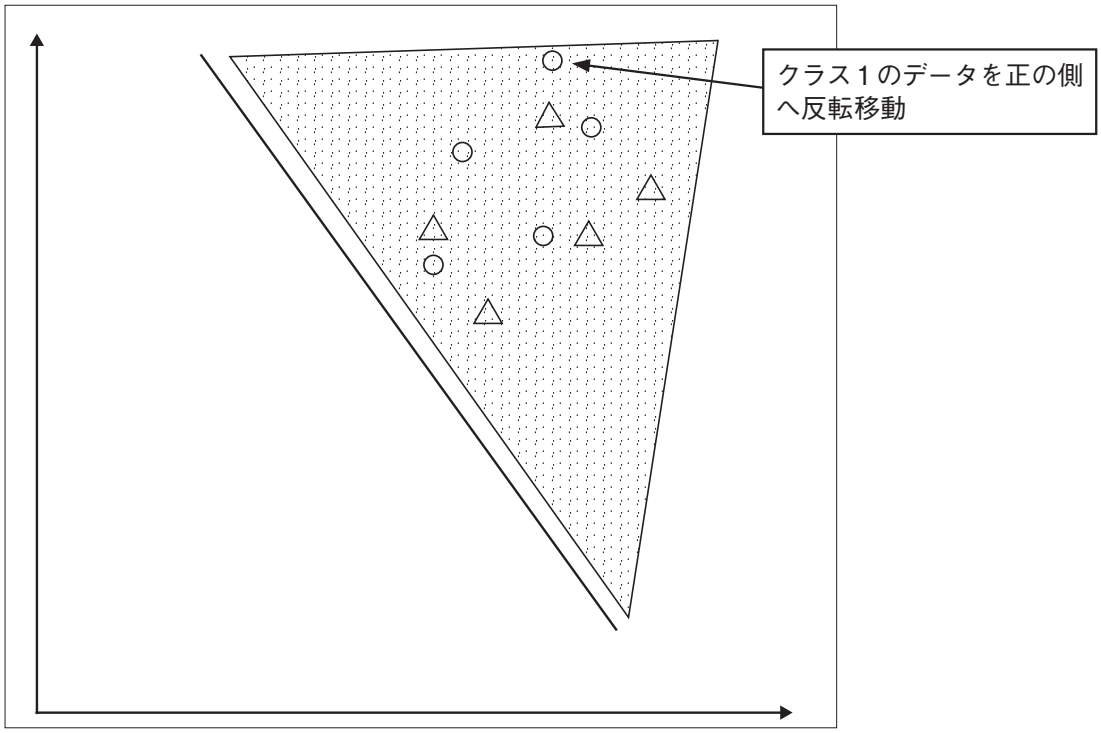
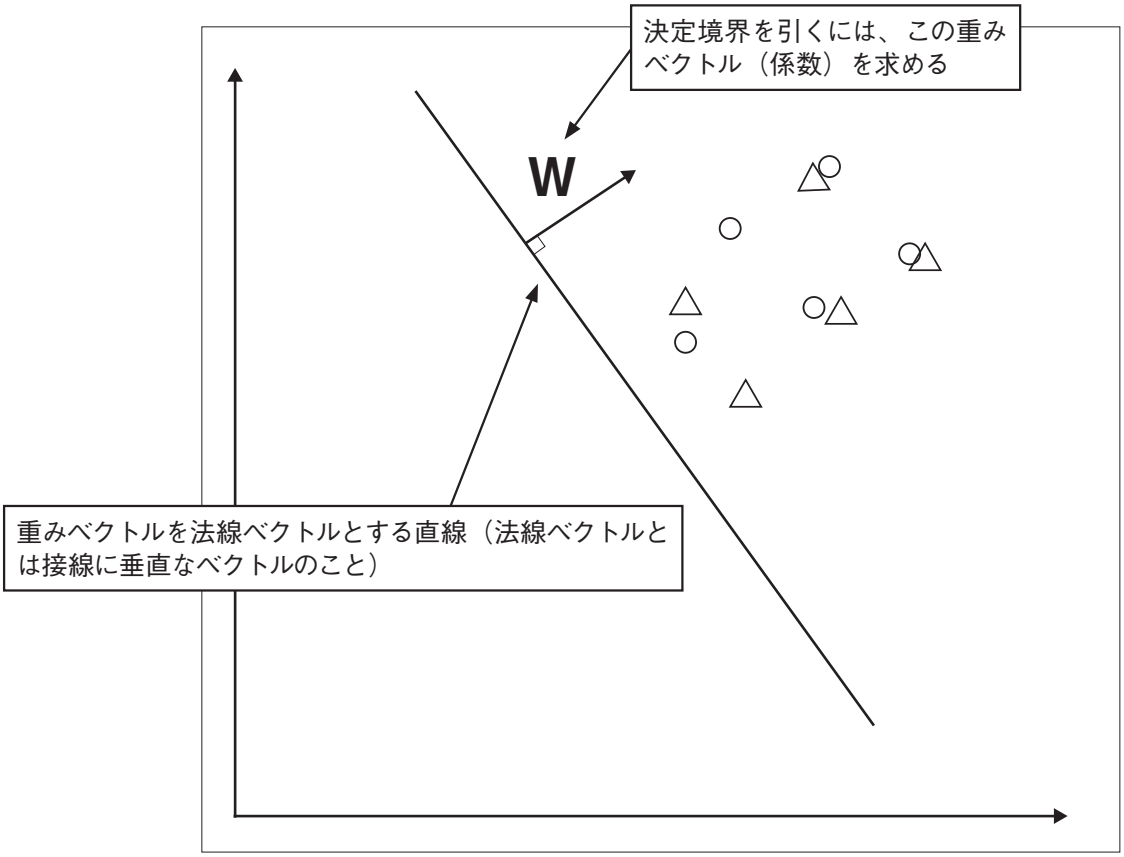
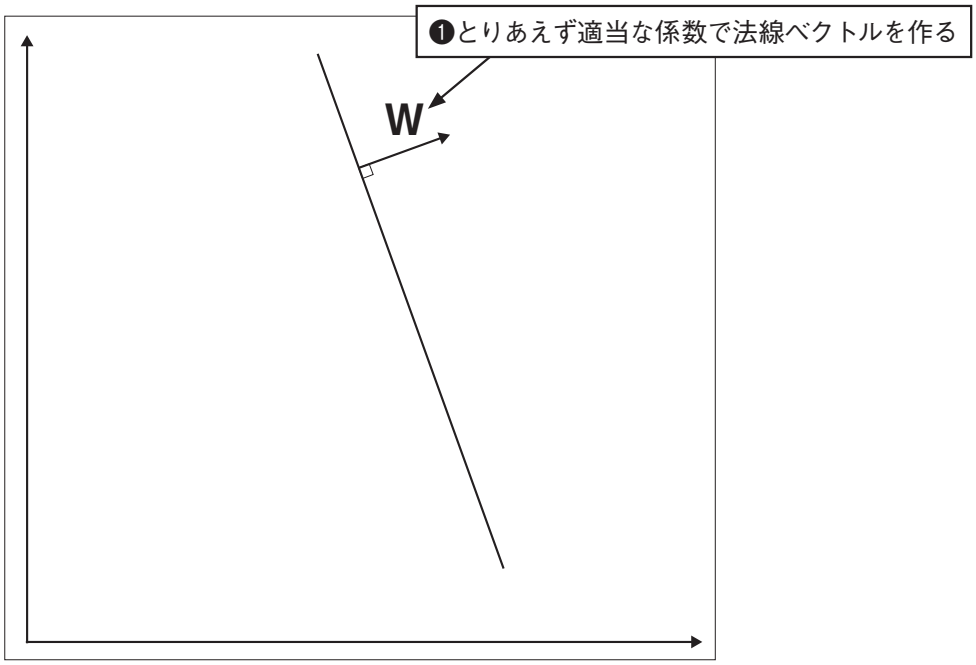


図６●決定境界を導き出す



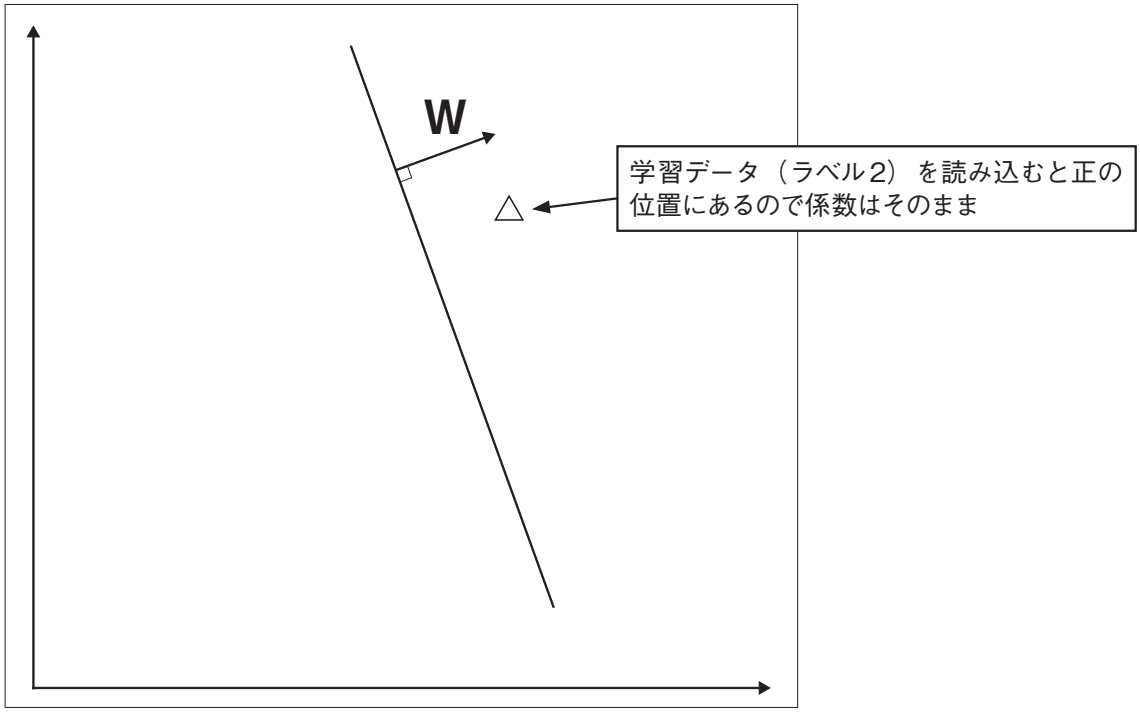
まず、適当な係数を用意して「仮の決定境界」を引いておきます（**図7**）。
次に学習データを読み込みますが、もし、ラベル1なら反転した値にします。

図7●決定境界の考え方（その1）



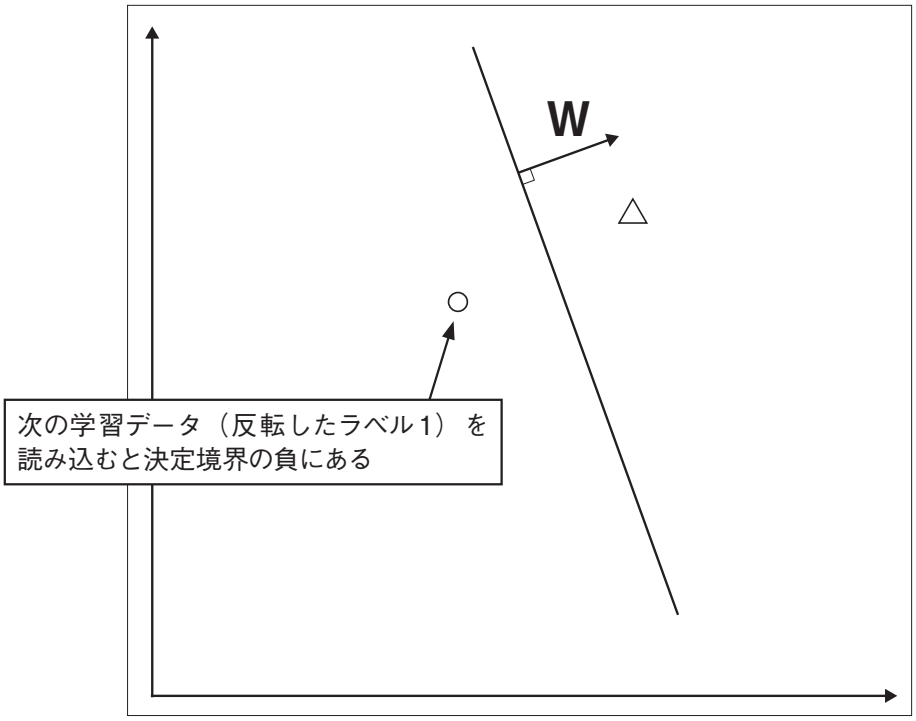
読み込んだ学習データが、「仮の決定境界線」の正の側にあるのか負の側にあるのかを判断します。もし、正の位置にある場合は、重みベクトルの係数はそのままです（**図8**）。

図8●決定境界の考え方（その2）



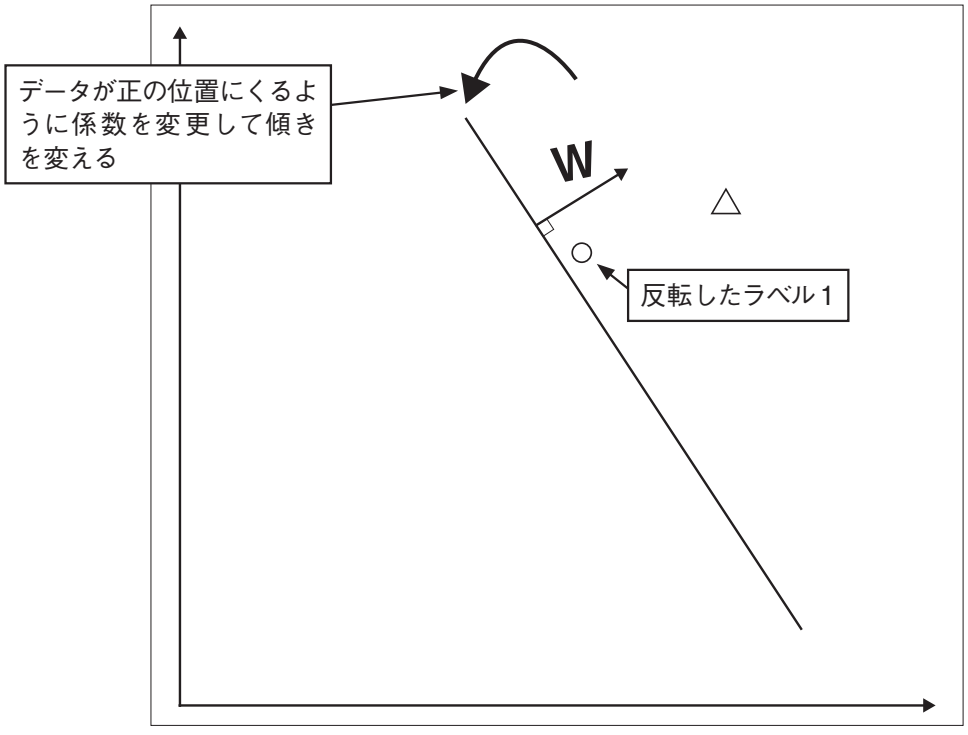
しかし、読み込んだデータが負の位置にある場合は…（**図9**）、

図9●決定境界の考え方（その3）



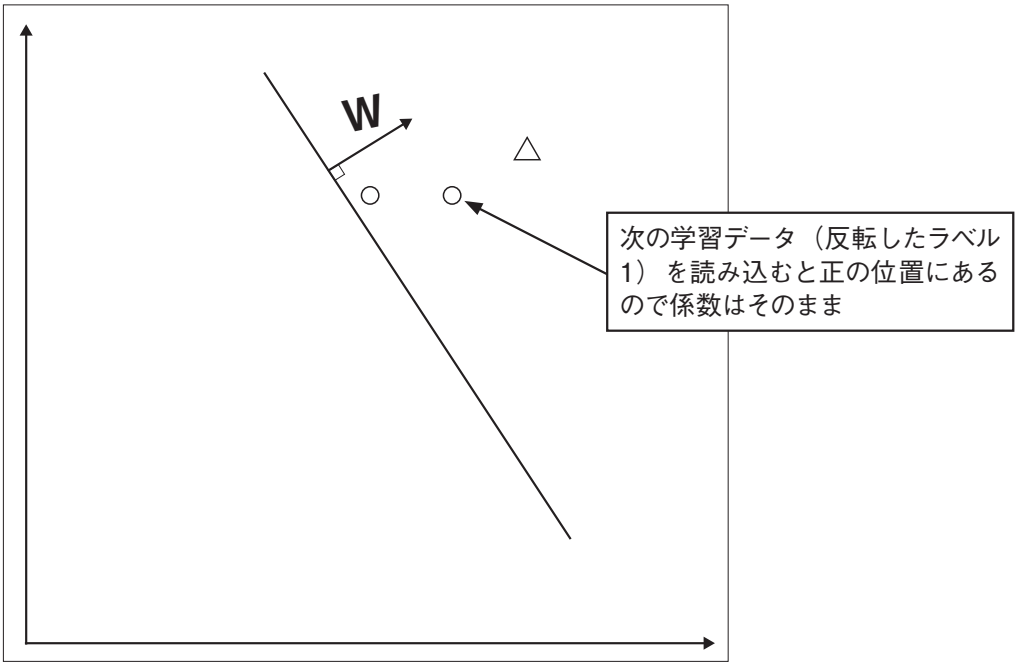
重みベクトルの係数を変更して、傾きを回転させてデータが正の位置に来るようにします（**図10**）。

図10●決定境界の考え方（その4）



こうして、学習データを読み込みながら、重みベクトルの係数を逐次調整します（**図11**）。

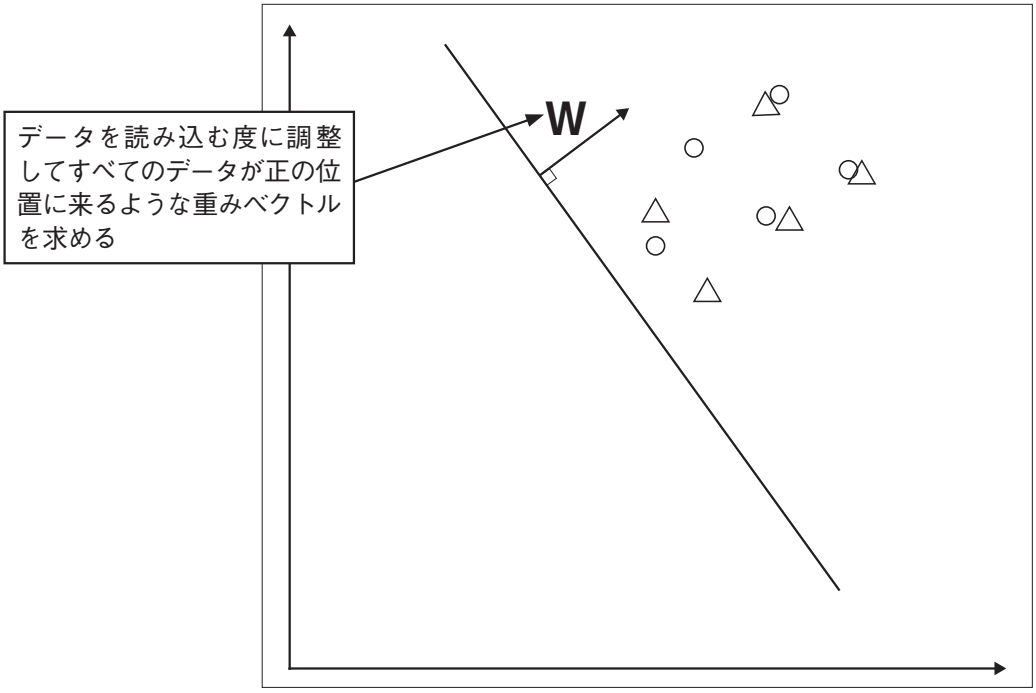
図11●決定境界の考え方（その5）



データの読み込みが完了すると、すべてのデータが正の位置に来ます。これにより「重みベクトルを放線ベクトルとする直線」が引けることになります（**図12**）。

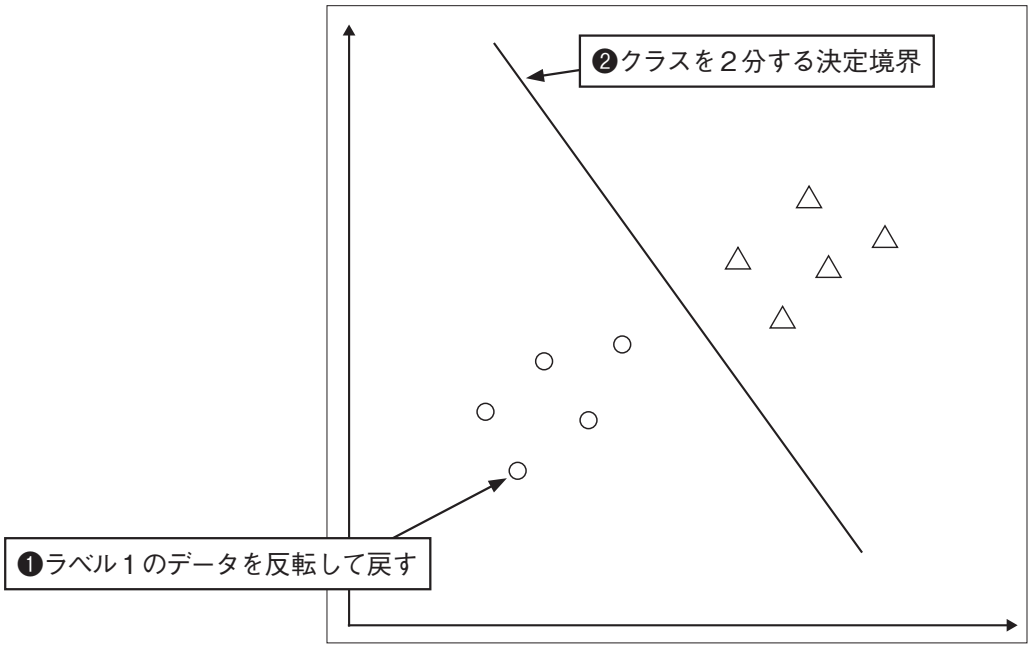
反転してあったラベル1のデータを再び反転させると、この直線は2つのデータを2分す

図12●決定境界の考え方（その6）



る決定境界になっています（**図13**）。

図13●線形分離完了

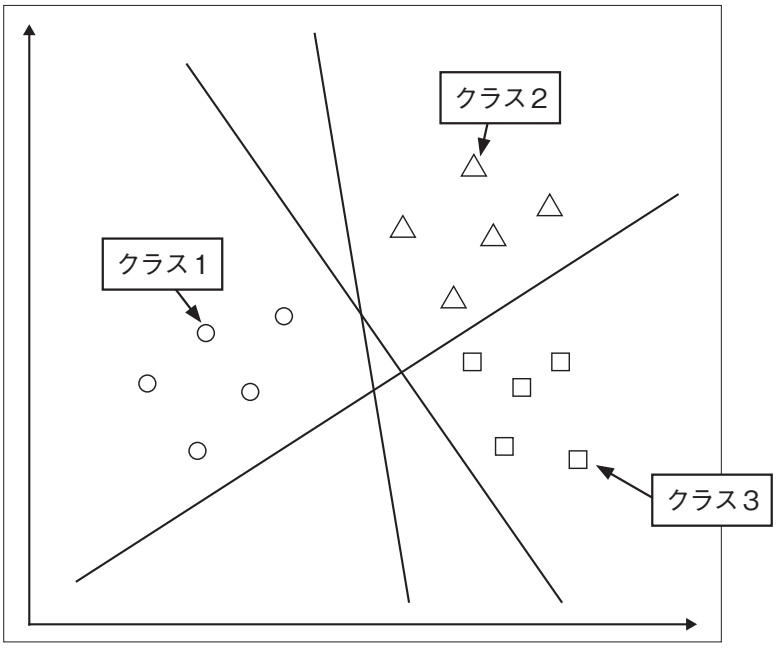


これが、単純パーセプトロンの学習方法になります。

▶クラスが複数ある場合

ここまでは、クラスが2つの場合でしたが、クラスが3つ、4つと増えた場合は、決定境界をたくさん引くことになります。例えば、**図14**のように線形分離が可能な場合があります。

図14●クラスが3つの場合



4日目 Part 2 scikit-learnのパーセプトロン

パーセプトロンの考え方がわかったところで、irisのデータセットを機械学習してみましょう。scikit-learnには、パーセプトロンを実装したlinear_model.Perceptronがあります。書式は、**図1**のようになります。

図1●Perceptronの書式。引数は抜粋

```
Perceptron(penalty = None 、alpha = 0.0001 、fit_intercept = T
rue 、max_iter = None 、tol = None 、shuffle = True 、verbose =
0 、eta0 = 1.0 、n_jobs = 1 、random_state = 0 、class_weight =
None 、warm_start = False 、n_iter = None)
```

引数	説明
max_iter	トレーニングデータに対する試行の最大回数（別名エポック）。バージョン0.19の新機能（推奨）
n_iter	トレーニングデータに対する試行回数（非推奨）
eta0	更新時に乗算される定数
shuffle	トレーニングデータをシャッフルするか否か
random_state	データをシャッフルするときの疑似乱数ジェネレータのシード

このPerceptronから学習モデルを生成して、irisデータセットの識別が可能かどうか試します。まずは、irisデータセットを読み込みます。

```
In [2]: from sklearn.datasets import load_iris
        iris_dataset = load_iris()
```

前回と同じく、train_test_split関数を使い、データを次のように分割します。

```
X_train 訓練用のデータ（7割）
```

```
X_test  評価用のデータ（3割）
y_train 訓練用の正解ラベル（7割）
y_test  評価用の正解ラベル（3割）

In [3]: from sklearn.model_selection import train_test_split
        X_train, X_test, y_train, y_test = train_test_split(ir
is_dataset['data'], iris_dataset['target'], test_size=0.3,  ra
ndom_state=0)
```

今回は、データを「正規化」（標準化）してみます。

データを正規化することで、値のバラツキが押さえられ、扱いやすくなります。ここでは、平均を0に分散を1に整えています。したがって、一番多いデータがX座標の0付近になり、−1から1の間に分散するデータになります。

```
In [4]: from sklearn.preprocessing import StandardScaler
        sc = StandardScaler()

        # データセットをStandardScalerにセット
        sc.fit(X_train)

        # データセットを正規化
        X_train = sc.transform(X_train)
        X_test = sc.transform(X_test)
```

パーセプトロンの学習モデルを生成します。パーセプトロンでは、重みベクトルであるwが少しずつ変更され、決定境界の直線が最適化されていきます。パラメータeta0は、1度にどれくらい傾かせるかを決める値なので、この値が小さいほど、最適値に到達しやすくなります。最適値に到達した状態を「収束した」と言います。

ただし、一度に傾く量がわずかだと、何度も試行することになり、試行回数が増えて処理が遅くなります。ここでは、最大試行回数であるmax_iterを100、eta0を0.1にしていますが、何度か実行して効率の良いパラメータ値にすべきかもしれません。

```
In [5]: from sklearn.linear_model import Perceptron
        ppn = Perceptron(max_iter=100, eta0=0.1, random_state=
0, shuffle=True)
```

```
        ppn.fit(X_train, y_train)
Out[5]: Perceptron(alpha=0.0001, class_weight=None, eta0=0.1,
fit_intercept=True,
              max_iter=100, n_iter=None, n_jobs=1, penalty=Non
e, random_state=0,
              shuffle=True, tol=None, verbose=0, warm_start=Fa
lse)
```

これで学習が完了したので、精度を評価します。

```
In [6]: y_pred = ppn.predict(X_test)
        import numpy as np
        np.mean(y_pred == y_test)

Out[6]: 0.97777777777777775
```

パーセプトロンでもk近傍法と変わらない精度です。

次に、誤分類した数を求めてみます。

```
In [7]: print('誤分類:%d' % (y_test != y_pred).sum())
        誤分類:1
```

目視で比べたときと同じように、1つ分類に失敗したようですね。精度の計算にNumPyのmean関数を使いましたが、sklearn.metricsにaccuracy_score関数というモデルの正解率を返す関数があるので、こちらも試しに使ってみました。

```
In [8]: from sklearn.metrics import accuracy_score
        print('正解率: %.2f' % accuracy_score(y_test, y_pred))
        正解率: 0.98
```

98%の正解率になっています。この数値を見る限り、irisデータセットならパーセプトロンで十分なのかもしれません。

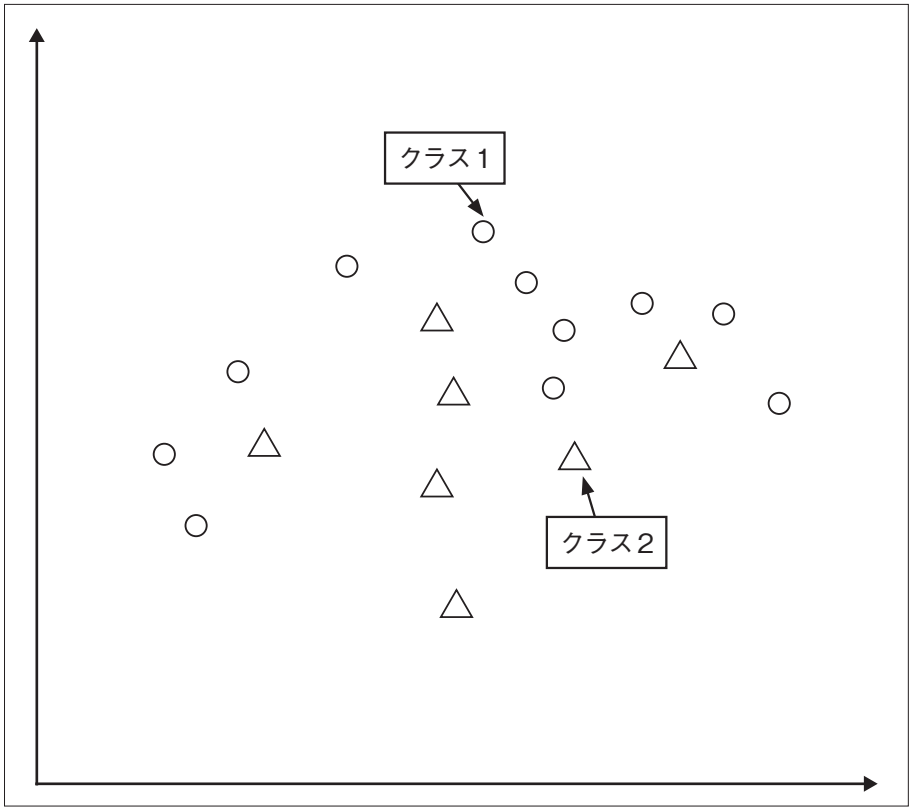
▶非線形分離可能

パーセプトロンにより、クラスを分離できることはわかりましたが、ここである疑問が浮かびます。

「学習データは、いつも『線形分離可能』とは限らない。直線で分離できない場合は、どうするの?」

例えば、**図2**のようなデータです。

図2●非線形分離可能



このように決定境界が曲線だったり、データが混ざっている場合、直線を使ってクラスを分類することはできません。このような状態を「非線形分離可能」(線形分離不可能)と呼びます。

そこで、非線形分離可能な問題に対処できるアルゴリズムということで、「ロジスティック回帰」と「サポートベクターマシン」に挑戦です。

4日目 Part 3 ロジスティック回帰

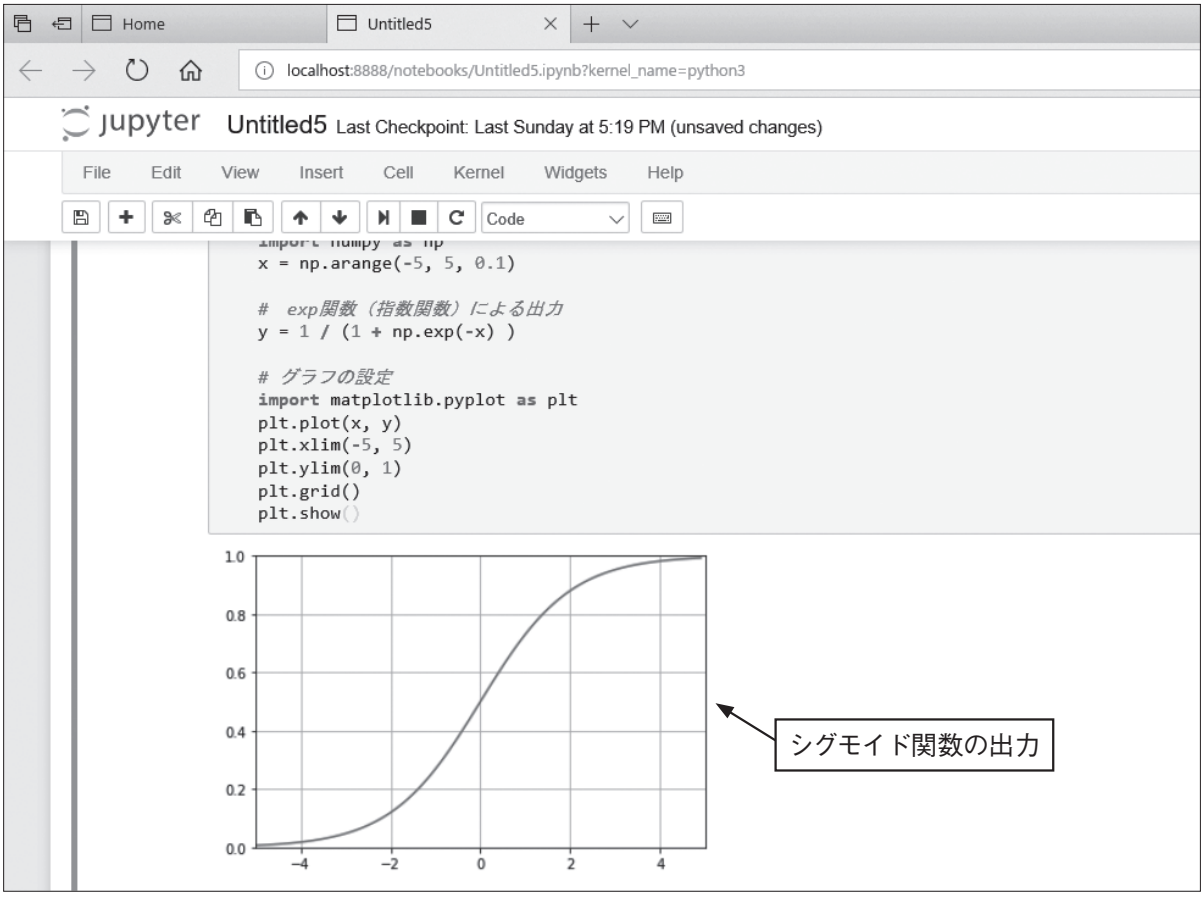
ロジスティック回帰は、クラスを分離する判断に「0」か「1」（正か負）ではなく、「確率」を用います。例えば、与えられたデータがクラス1である確率は80%、そうでない確率は20%のように判断します。つまり、たとえ正の可能性があったとしても、確率が低い場合は決定境界の重みベクトルを修正できます。

ロジスティック回帰では、このように確率でクラスを分離するわけですが、実際のデータで確率をそのまま計算すると、−150%の確率や1200%の確率などが求められてしまいます。

そこで、0%～ 100%の範囲内にデータを押し込めてしまう関数「シグモイド関数」を利用します。シグモイド関数は、次のような関数です（**図1**）。

```
In [10]: # xの値（-5 ～ 5の間で0.1刻み）
```

図1●シグモイド関数



```
import numpy as np
x = np.arange(-5, 5, 0.1)
# exp関数（指数関数）による出力
y = 1 / (1 + np.exp(-x) )

# グラフの設定と表示
import matplotlib.pyplot as plt
plt.plot(x, y)
plt.xlim(-5, 5)
plt.ylim(-0.1, 1.1)
plt.grid()
plt.show()
```

シグモイド関数の出力は、0 ～ 1 の間をなめらかに移動します。つまり、この関数を間に入れることで、常に出力を 0 ～ 1 の間にして、0.5なら50%、0.8なら80%のように判断するわけです。

▶LogisticRegression

ロジスティック回帰で機械学習を行うには、scikit-learnのLogisticRegressionを使います。書式は**図2**のようになります。

図2●LogisticRegressionの書式。引数は抜粋

```
LogisticRegression(penalty='l2', dual=False, tol=0.0001, C=1.0,
fit_intercept=True, intercept_scaling=1, class_weight=None, ran
dom_state=None, solver='liblinear', max_iter=100, multi_class='ov
r', verbose=0, warm_start=False, n_jobs=1)
```

引数	説明
penalty	正則化の種類（デフォルトはL2ノルム正則化）
C	Regularization（正則化）の強さ

LogisticRegressionの引数にある「penalty」と「C」は「正則化項」と呼ばれます。正

則化とは、モデルの「複雑さ」が過剰に増えないように「ペナルティ」を設ける手法なのですが、ここで正則化の話をする前に、「過学習」の話をしておきます。

過学習とは、学習データに限りなくフィットするように学習させた結果、逆に評価データ（未知のデータ）を与えた時に精度が落ちてしまう現象のことです。そして過学習にならないようにするには、正則化を行うと良いらしいのです。

モデルの「複雑さ」を表す指標には「L1ノルム正則化」と「L2ノルム正則化」があり、係数である引数Cの値を多くすると正則化が強く働きます。

今回は、これらのパラメータは使わず（正則化は行わず）機械学習を行ってみます。利用するデータは、irisデータセットではなく、digitsデータセットにします。

▶LogisticRegressionによる学習と評価

基本的な手順は、これまでとまったく同じです。

```
In [11]: # digitsデータセットの読み込み
         from sklearn.datasets import load_digits
         digits = load_digits()
```

digitsデータセットを次のように分割します。

```
X_train 訓練用のデータ（7割）
X_test  評価用のデータ（3割）
y_train 訓練用の正解ラベル（7割）
y_test  評価用の正解ラベル（3割）

In [12]: from sklearn.model_selection import train_test_split
         X_train, X_test, y_train, y_test = train_test_split(d
igits['data'], digits['target'], test_size=0.3,  random_state=
0)
```

ロジスティック回帰の学習モデルを生成し、学習を行います。

```
In [13]: from sklearn.linear_model import LogisticRegression
         logreg = LogisticRegression()
         logreg_model = logreg.fit(X_train, y_train)
```

評価用データで予測を行います。

```
In [14]: pred = logreg.predict(X_test)
```

これで学習が完了したので、精度を評価します。

```
In [15]: pred = logreg.predict(X_test)
         import numpy as np
         np.mean(pred == y_test)

Out[15]: 0.95370370370370372
```

評価用の正解ラベルを表示してみます。

```
In [16]: print(y_test)
        [2 8 2 6 6 7 1 9 8 5 2 8 6 6 6 6 1 0 5 8 8 7 8 4 7 5 4
9 2 9 4 7 6 8 9 4 3
         1 0 1 8 6 7 7 1 0 7 6 2 1 9 6 7 9 0 0 5 1 6 3 0 2 3 4
1 9 2 6 9 1 8 3 5 1
         2 8 2 2 9 7 2 3 6 0 5 3 7 5 1 2 9 9 3 1 7 7 4 8 5 8 5
5 2 5 9 0 7 1 4 7 3
         4 8 9 7 9 8 2 6 5 2 5 8 4 8 7 0 6 1 5 9 9 9 5 9 9 5 7
5 6 2 8 6 9 6 1 5 1
         5 9 9 1 5 3 6 1 8 9 8 7 6 7 6 5 6 0 8 8 9 8 6 1 0 4 1
6 3 8 6 7 4 5 6 3 0
         3 3 3 0 7 7 5 7 8 0 7 8 9 6 4 5 0 1 4 6 4 3 3 0 9 5 9
2 1 4 2 1 6 8 9 2 4
         9 3 7 6 2 3 3 1 6 9 3 6 3 2 2 0 7 6 1 1 9 7 2 7 8 5 5
7 5 2 3 7 2 7 5 5 7
         0 9 1 6 5 9 7 4 3 8 0 3 6 4 6 3 2 6 8 8 8 4 6 7 5 2 4
5 3 2 4 6 9 4 5 4 3
         4 6 2 9 0 1 7 2 0 9 6 0 4 2 0 7 9 8 5 4 8 2 8 4 3 7 2
6 9 1 5 1 0 8 2 1 9
         5 6 8 2 7 2 1 5 1 6 4 5 0 9 4 1 1 7 0 8 9 0 5 4 3 8 8
6 5 3 4 4 4 8 8 7 0
         9 6 3 5 2 3 0 8 3 3 1 3 3 0 0 4 6 0 7 7 6 2 0 4 4 2 3
7 8 9 8 6 8 5 6 2 2
```

```
          3 1 7 7 8 0 3 3 2 1 5 5 9 1 3 7 0 0 7 0 4 5 9 3 3 4 3
1 8 9 8 3 6 2 1 6 2
          1 7 5 5 1 9 2 8 9 7 2 1 4 9 3 2 6 2 5 9 6 5 8 2 0 7 8
0 5 8 4 1 8 6 4 3 4
          2 0 4 5 8 3 9 1 8 3 4 5 0 8 5 6 3 0 6 9 1 5 2 2 1 9 8
4 3 3 0 7 8 8 1 1 3
          5 5 8 4 9 7 8 4 4 9 0 1 6 9 3 6 1 7 0 6 2 9]
```

予測したラベルを表示します。

```
In [17]: print(pred)
          [2 8 2 6 6 7 1 9 8 5 2 8 6 6 6 6 1 0 5 8 8 7 8 4 7 5
4 9 2 9 4 7 6 8 9 4 3
           8 0 1 8 6 7 7 1 0 7 6 2 1 9 6 7 9 0 0 5 1 6 3 0 2 3
4 1 9 2 6 9 1 8 3 5 1
           2 8 2 2 9 7 2 3 6 0 5 3 7 5 1 2 9 9 3 1 4 7 4 8 5 9
5 5 2 5 9 0 7 1 4 1 3
           4 8 9 7 8 8 2 1 5 2 5 8 4 1 7 0 6 1 5 5 9 9 5 9 9 5
7 5 6 2 8 6 9 6 1 5 1
           5 9 9 1 5 3 6 1 8 9 8 7 6 7 6 5 6 0 8 8 9 8 6 1 0 4
1 6 3 8 6 7 4 1 6 3 0
           3 3 3 0 7 7 5 7 8 0 7 1 9 6 4 5 0 1 4 6 4 3 3 0 9 5
3 2 1 4 2 1 6 9 9 2 4
           9 3 7 6 2 3 3 1 6 9 3 6 3 2 2 0 7 6 1 1 9 7 2 7 8 5
5 7 5 3 3 7 2 7 5 5 7
           0 9 1 6 5 9 7 4 3 8 0 3 6 4 6 3 2 6 8 8 8 4 6 7 5 2
4 5 3 2 4 6 9 4 5 4 3
           4 6 2 9 0 6 7 2 0 9 6 0 4 2 0 7 8 8 5 4 8 2 8 4 3 7
2 6 9 1 5 1 0 8 2 8 9
           5 6 2 2 7 2 1 5 1 6 4 5 0 9 4 1 1 7 0 8 9 0 5 4 3 8
8 6 5 3 4 4 4 8 8 7 0
           9 6 3 5 2 3 0 8 2 3 1 3 3 0 0 4 6 0 7 7 6 2 0 4 4 2
3 7 1 9 8 6 8 5 6 2 2
           3 1 7 7 8 0 9 3 2 6 5 5 9 1 3 7 0 0 3 0 4 5 9 3 3 4
```

```
3 1 8 9 8 3 6 3 1 6 2
          1 7 5 5 1 9 2 8 9 7 2 8 4 9 3 2 6 2 5 9 6 5 8 2 0 7
8 0 6 8 4 1 8 6 4 3 4
          2 0 4 5 8 3 9 1 8 3 4 5 0 8 5 6 3 0 6 9 1 5 2 2 1 9
8 4 3 3 0 7 8 8 1 1 3
          5 5 8 4 9 7 8 4 4 9 0 1 6 9 3 6 1 7 0 6 2 9]
```

irisデータセットと違って予想の正解、不正解を目視で確認するのは大変ですね。そこで、どのくらい正解したのか、scikit-learnのconfusion_matrix関数で調べましょう。

```
In [18]: from sklearn.metrics import confusion_matrix
         confusion_matrix(y_test, pred, labels=digits['target_
names'])
Out[18]: array([[45,  0,  0,  0,  0,  0,  0,  0,  0,  0],
                [ 0, 47,  0,  0,  0,  0,  2,  0,  3,  0],
                [ 0,  0, 51,  2,  0,  0,  0,  0,  0,  0],
                [ 0,  0,  1, 52,  0,  0,  0,  0,  0,  1],
                [ 0,  0,  0,  0, 48,  0,  0,  0,  0,  0],
                [ 0,  1,  0,  0,  0, 55,  1,  0,  0,  0],
                [ 0,  1,  0,  0,  0,  0, 59,  0,  0,  0],
                [ 0,  1,  0,  1,  1,  0,  0, 50,  0,  0],
                [ 0,  3,  1,  0,  0,  0,  0,  0, 55,  2],
                [ 0,  0,  0,  1,  0,  1,  0,  0,  2, 53]], dty
pe=int64)
```

縦軸が 0 ～ 9 までの正解です。横軸がその数字を学習モデルが予測した数です。0は0と予測したものが45個、その他の数字に予測したものはありません。1は、1と予測したものが47個、6と予測したものが2個、8と予測したものが3個ありました。

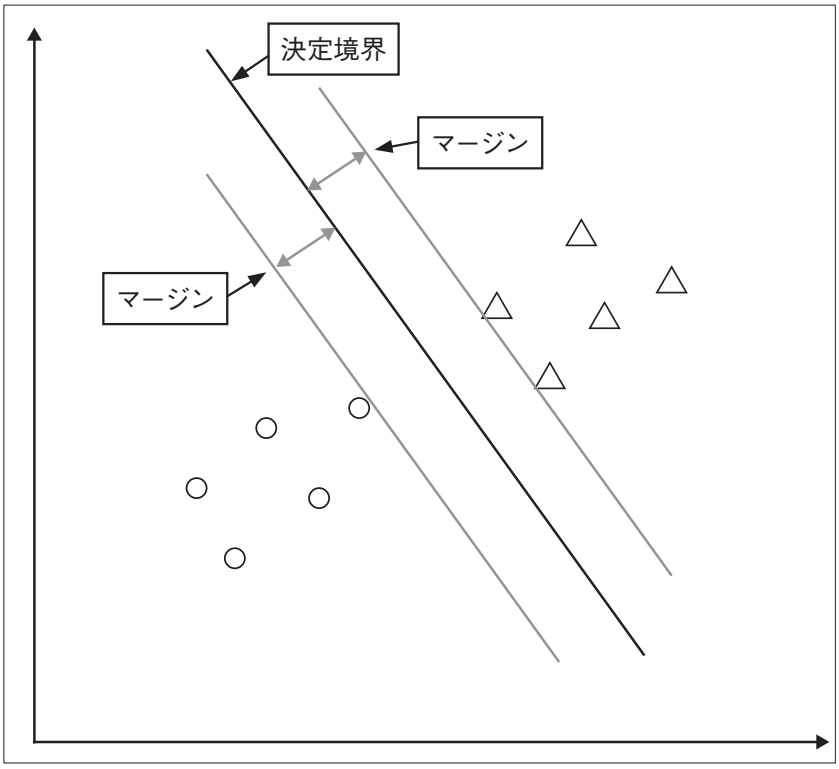
Part 4 4日目 サポートベクターマシン

サポートベクターマシン（SVM）は、2クラスの識別アルゴリズムでは「強力」と言われているようですが、アルゴリズムが難しいです。正直、私はまだよくわかっておりません。

ただ、理解できた範囲内で説明すると、SVMの特徴には「マージン最大化」と「カーネルトリック」があるということです。

「マージン」とは、識別面と2つのクラス間の距離のことです。このマージンが最大になるようにすることで、「汎化能力を最大にしよう」ということのようです（**図1**）。

図1●マージン最大化



それほど学習しなくても評価データ（未知のデータ）の判別精度が高い学習モデルを「汎化性能が高い」と言います。つまり、SVMはマージン最大化によって、未知のデータに対して判別精度が高いアルゴリズムだというわけですね。

ただし、マージン最大化は、あくまでも線形分離可能なデータに対してのみ有効です。そして、SVMは「カーネルトリック」という手法を取り入れることで、線形分離が難しい問題に対処しています。

「カーネルトリック」とは、高次元空間上で線形分析を行う機能です。これまでのような2

次元の分析ではなく、3次元、4次元へと写像して線形分析を行う…らしいのですが、これ以上は、この原稿の締め切りまでに理解できませんでした。

とりあえず、学習モデルが生成できれば、何とかなるのではないかということで、実装を始めましょう。

▶svm.SVC

非線形な識別を可能にする学習モデルを作るには、svm.SVCを使います。書式は**図2**のようになります。

図2●svm.SVCの書式。引数は抜粋

```
svm.SVC(C = 1.0 、kernel = 'rbf' 、degree = 3 、gamma = 'auto' 、
coef0 = 0.0 、shrinking = true 、確率= false 、tol = 0.001 、cach
e_size = 200 、class_weight = None 、verbose = False 、max_iter
= -1 、decision_function_shape = 'ovr' 、random_state = None )
```

引数	説明
C	どれだけ誤分類を許容するか
kernel	カーネルの種類（linear、poly、rbf、sigmoid、precomputed）
gamma	カーネルの種類が、rbf、poly、sigmoid、のとき使用

▶SVMで画像の識別

機械学習のテストは、ロジスティック回帰と同じ手順です。ただし、学習モデルはsvm.SVCを使います。パラメータのgamma=0.001は、kernelが'rbf'（デフォルト）の場合の決定境界の複雑度合いを表し、大きいほど複雑になります。

また、C=100.は誤分類を許容する尺度です。これらの引数の値は、digitsデータセットに対する適正値がわからなかったので、とりあえず入門書などのサンプル数値を参考にしています。

それでは、svmをインポートして、svm.SVCの学習モデルを生成しましょう。

```
In [20]: from sklearn import svm
```

```
        clf = svm.SVC(gamma=0.001, C=100.)
```

学習モデルが生成できたら、学習させます。

```
In [21]: clf.fit(X_train, y_train)
         SVC(C=100.0, cache_size=200, class_weight=None, coef0
=0.0,
         decision_function_shape='ovr', degree=3, gamma=0.001,
kernel='rbf',
         max_iter=-1, probability=False, random_state=None, sh
rinking=True,
         tol=0.001, verbose=False)
```

学習が完了したので、精度を評価してみましょう。

```
In [22]: pred = clf.predict(X_test)
         import numpy as np
         np.mean(pred == y_test)

Out[22]: 0.9907407407407407
```

精度が高い気がします。それぞれの数字の正解、不正解を確認します。

```
In [23]: from sklearn.metrics import confusion_matrix
         confusion_matrix(y_test, pred, labels=digits['target_
names'])
Out[23]: array([[45,  0,  0,  0,  0,  0,  0,  0,  0,  0],
                [ 0, 52,  0,  0,  0,  0,  0,  0,  0,  0],
                [ 0,  0, 52,  0,  0,  0,  0,  1,  0,  0],
                [ 0,  0,  0, 54,  0,  0,  0,  0,  0,  0],
                [ 0,  0,  0,  0, 48,  0,  0,  0,  0,  0],
                [ 0,  0,  0,  0,  0, 55,  1,  0,  0,  1],
                [ 0,  0,  0,  0,  0,  0, 60,  0,  0,  0],
                [ 0,  0,  0,  0,  0,  0,  0, 53,  0,  0],
                [ 0,  1,  0,  0,  0,  0,  0,  0, 60,  0],
                [ 0,  0,  0,  0,  0,  1,  0,  0,  0, 56]], dty
```

```
pe=int64)
```

成績が大きく向上しています。SVM最強説は、ホントかもしれません。

もっとも、パラメータの調整やデータの種類でスコアは異なるので、他のアルゴリズムでもパターンを変えてテストする必要はありそうです。

明日は、ニューラルネットワークと教師なし学習を試してみたいと思います。