ラビット・チャレンジ

＜実装演習レポート　機械学習＞

メールアドレス：mtop.jp@gmail.com
受講者名：山崎　英和
受講種別：ラビット・チャレンジ

＜線形回帰モデル＞
要点まとめ

■回帰問題とは
回帰問題とは数値（連続値）を予想する問題のこと。与えられた入力データ（連続値または離散値）と出力データから対応する規則を学習し、未知の入力データに対して適切な出力結果を生成する手法。
教師あり学習に分類される。
線形回帰とは直線で予測する手法であり、曲線で予測する手法は非線形回帰と呼ばれる

■回帰で扱うデータ
入力データ：m 次元のベクトル ※説明変数または特徴量と呼ばれる
出力データ：スカラ値 ※目的変数と呼ばれる

■線形回帰モデル
線形回帰モデル： $\hat{y} = \boldsymbol{w^T x} + w_0 = \sum_{j=1}^{m} w_j x_j + w_0$
説明変数： $\boldsymbol{x} = (x_1, x_2, \dots, x_m)^T \in \mathbb{R}^m$
目的変数： $y \in \mathbb{R}^1$
パラメータ： $\boldsymbol{w} = (w_1, w_Z, \dots, w_m)^T \in \mathbb{R}^m$

※説明変数が一つのものを単回帰、二つ以上を重回帰と呼ぶ

■線形回帰モデルのパラメータ推定
データとモデルの出力値の誤差を最小にするために、最小二乗法で推定する

回帰係数： $\hat{w} = \left(X^{(tr)T} X^{(tr)}\right)^{-1} X^{(tr)^T} y^{(tr)}$

予測値： $\hat{\boldsymbol{y}} = X_* \left(X^{(tr)T} X^{(tr)}\right)^{-1} X^{(tr)^T} y^{(tr)}$

■実装演習結果キャプチャ又はサマリーと考察

コード：skl_regression.ipynb

CRIM と RM による 2 変数重回帰分析

CRIM=0.2、RM=6 の場合、21000 ドルの予想結果となる

重回帰分析(2変数)

```
[29] #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

```
[30] # 説明変数
     data2 = df.loc[:, ['CRIM', 'RM']].values
     # 目的変数
     target2 = df.loc[:, 'PRICE'].values
```

```
[31] # オブジェクト生成
     model2 = LinearRegression()
```

```
[32] # fit関数でパラメータ推定
     model2.fit(data2, target2)
```

```
LinearRegression(copy_X=True, fit_intercept=True, n_jobs=None, normalize=False)
```

```
model2.predict([[0.2, 6]])
```

```
array([21.04870738])
```

＜非線形回帰モデル＞
要点まとめ

■非線形回帰モデルとは

非線形回帰モデル：　$\hat{y} = \boldsymbol{w^T}\phi(\boldsymbol{x_i}) + w_0 = \sum_{j=1}^{n} w_j\phi_j(x_i) + w_0$

※求める w については線形の式

説明変数：　$\boldsymbol{x}_i = \left(x_{i_1}, x_{i_2}, \dots, x_{i_m}\right) \in \mathbb{R}^m$

非線形関数ベクトル：　$\phi(\boldsymbol{x}_i) = \left(\phi_1(\boldsymbol{x}_i), \phi_2(\boldsymbol{x}_i), \dots, \phi_k(\boldsymbol{x}_i)\right)^T \in \mathbb{R}^k$

※基底関数$\phi(\boldsymbol{x}_i)$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を使用

目的変数：　$y \in \mathbb{R}^1$

パラメータ：　$\boldsymbol{w} = (w_1, w_Z, \dots, w_m)^T \in \mathbb{R}^m$

■非線形回帰モデルのパラメータ推定

データとモデルの出力値の誤差を最小にするために、線形回帰モデル同様、最小二乗法または最尤法で推定する

回帰係数：　$\widehat{w} = \left(\boldsymbol{\Phi}^{(tr)T}\boldsymbol{\Phi}^{(tr)}\right)^{-1}\boldsymbol{\Phi}^{(tr)T}\boldsymbol{y}^{(tr)}$

予測値：　$\hat{y} = \Phi_*\left(\boldsymbol{\Phi}^{(tr)T}\boldsymbol{\Phi}^{(tr)}\right)^{-1}\boldsymbol{\Phi}^{(tr)T}\boldsymbol{y}^{(tr)}$

■未学習と過学習

・未学習：学習に対して、十分小さな誤差が得られないモデル
　対策：表現力の高いモデルを利用する
・過学習：小さな誤差は得られたけど、テスト集合誤差との差が大きいモデル
　対策１：学習データの数を増やす
　対策２：不要な基底関数（変数）を削除して表現力を抑止
　対策３：正則化法を利用して表現力を抑止

■正則化法

モデルの複雑さに伴い、その値が大きくなる正則化項（罰則項）を課した関数を最小化
$\min MSE \; s.t. \; R(\boldsymbol{w}) \leq r$
最適化 KKT 条件より　$\min MSE + \lambda R(\boldsymbol{w}) \quad \lambda > 0$

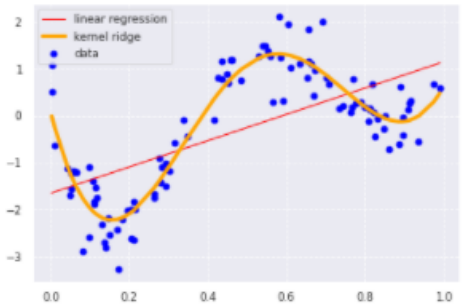
■実装演習結果キャプチャ又はサマリーと考察

コード：skl_nonlinear regression.ipynb

非線形回帰モデルの方が、線形回帰モデルより表現力が高い

```
from sklearn.kernel_ridge import KernelRidge

clf2 = KernelRidge(alpha=0.0002, kernel='rbf')
clf2.fit(data, target)

p_kridge = clf2.predict(data)

plt.scatter(data, target, color='blue', label='data')
plt.plot(data, p_lin, color='red', marker='', linestyle='-', linewidth=1, markersize=6, label='linear regression')
plt.plot(data, p_kridge, color='orange', linestyle='-', linewidth=3, markersize=6, label='kernel ridge')
plt.legend()
#plt.plot(data, p, color='orange', marker='o', linestyle='-', linewidth=1, markersize=6)

print('linear regression score:', clf.score(data, target))
print('kernl  ridge      score:', clf2.score(data, target))
```

```
linear regression score: 0.3958971373603204
kernl  ridge      score: 0.8916875836966959
```



＜ロジスティック回帰モデル＞
要点まとめ

■分類問題における識別的アプローチと生成的アプローチ
・識別的アプローチ
$p(C_k|\boldsymbol{x})$を直接モデル化
・生成的アプローチ
$p(C_k)$と$p(\boldsymbol{x}|C_k)$をモデル化し、Bayes の定理を用いて$p(C_k|\boldsymbol{x})$を求める

■ロジスティック回帰モデル
分類問題を解くための教師あり機械学習モデル
分類にシグモイド関数などの識別関数を使った手法
m 次元入力パラメータの線形結合をシグモイド関数に入力とし、
その出力は y=1 となる確率の値となる
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

ロジスティック回帰モデル: $P(Y=1|\boldsymbol{x}) = \sigma(\boldsymbol{w^T x} + w_0) = \sigma(\sum_{j=1}^{m} w_j x_j + w_0)$

説明変数: $\boldsymbol{x} = (x_1, x_2, \dots, x_m)^T \in \mathbb{R}^m$
目的変数: $P(Y=1|\boldsymbol{x})$
パラメータ: $\boldsymbol{w} = (w_1, w_Z, \dots, w_m)^T \in \mathbb{R}^m$

シグモイド関数: $\sigma(x) = \frac{1}{1+\exp(-ax)}$

入力は実数・出力は 0～1 の値
クラス1に分類される確率を表現
単調増加関数
a を増加させると、x=0 付近での勾配が増加。極めて大きくすると単位ステップ関数に近づく

■シグモイド関数の性質
シグモイド関数の微分は、シグモイド関数自身で表現することが可能

$$\frac{\partial \sigma(x)}{\partial x} = a\sigma(x)\big(1-\sigma(x)\big)$$

■ロジスティック回帰モデルのパラメータ推定
データから最もらしいベルヌーイ分布（パラメータ）を推定（最尤推定）したい
尤度関数を最大化するようなパラメータを求めたい

尤度関数：$P(y_1, y_2, \ldots, y_n | w_0, w_1, \ldots, w_m) = \prod_{i=1}^{n} p_i^{y_i}(1 - P_i)^{1-y_i}$

$$= \prod_{i=1}^{n} \sigma(\boldsymbol{w}^T \boldsymbol{x}_i)^{y_i}\left(1 - \sigma(\boldsymbol{w}^T \boldsymbol{x}_i)\right)^{1-y_i} = L(\boldsymbol{w})$$

一般的に、尤度関数にマイナスを掛けたものに対数をとって最小となるパラメータを求める

$$E(\boldsymbol{w}) = -\log L(\boldsymbol{w})$$

$$= -\sum_{i=1}^{n} (y_i \log p_i + (1 - y_i) \log(1 - p_i))$$

■勾配下降法

・反復学習によりパラメータを逐次的に更新するアプローチの一つ

・η は学習率と呼ばれるハイパーパラメータでモデルのパラメータの収束しやすさを調整

・ロジスティック回帰モデルでは勾配下降法によりパラメータを求める

$$\boldsymbol{w}^{(k+1)} = \boldsymbol{w}^{(k)} - \eta \frac{\partial E(w)}{\partial w} = \boldsymbol{w}^{(k)} + \eta \sum_{i=1}^{n} (y_i - p_i) x_i$$

■確率的勾配下降法（SGD）

・勾配下降法ではパラメータ更新ごとに n 個のデータを計算する必要がある。データ個数が多くなると大量のメモリ、計算量が必要になるため、確率的勾配下降法でパラメータを更新

・データを一つずつランダムに(「確率的」に)選んでパラメータを更新

・勾配降下法でパラメータを 1 回更新するのと同じ計算量でパラメータを n 回更新できるので効率よく最適な解を探索可能

$$\boldsymbol{w}^{(k+1)} = \boldsymbol{w}^{(k)} + \eta (y_i - p_i) x_i$$

■実装演習結果キャプチャ又はサマリーと考察

コード：skl_logistic_regression.ipynb

ロジスティック回帰モデルで運賃から生死確率を確認

チケット価格が 61 ドルだと生存確率が 50%切り、62 ドルだと生存確率が 50%を上回る

```
[11] from sklearn.linear_model import LogisticRegression
```

```
[12] model=LogisticRegression()
```

```
[20] model.fit(data1, label1)
```

```
/usr/local/lib/python3.7/dist-packages/sklearn/utils/validation.py:760: DataConvers
  y = column_or_1d(y, warn=True)
LogisticRegression(C=1.0, class_weight=None, dual=False, fit_intercept=True,
                   intercept_scaling=1, l1_ratio=None, max_iter=100,
                   multi_class='auto', n_jobs=None, penalty='l2',
                   random_state=None, solver='lbfgs', tol=0.0001, verbose=0,
                   warm_start=False)
```

```
[44] model.predict([[61]])
```

```
array([0])
```

```
[45] model.predict_proba([[61]])
```

```
array([[0.50358033, 0.49641967]])
```

```
model.predict([[62]])
```

```
array([1])
```

```
[48] model.predict_proba([[62]])
```

```
array([[0.49978123, 0.50021877]])
```

＜主成分分析＞
要点まとめ

■主成分分析とは
相関のある多数の変数から相関のない少数で全体のばらつきを最もよく表す主成分と呼ばれる変数を合成する多変量解析手法。学習のためのデータの次元を削減するために用いられる
主成分を与える変換は、第一主成分の分散を最大化し、続く主成分はそれまでに決定した主成分と直交するという拘束条件の下で分散を最大化するようにして選ばれる

多くの場合、多変量データは次元が大きく、各変数を軸にとって視覚化することは難しいが、主成分分析によって情報をより少ない次元に集約することでデータを視覚化できる
集約によって得られる情報は、データセットを元のデータ変数の空間から主成分ベクトルのなす空間へ射影したものであり、元のデータから有用な情報を抜き出したものになっている

■実装演習結果キャプチャ又はサマリーと考察
コード：skl_pca.ipynb
30 個のデータをロジスティック回帰で学習、テストすると精度 97%

```
# 学習用とテスト用でデータを分離
X_train, X_test, y_train, y_test = train_test_split(X, y, random_state=0)
#X_train
#X_test

# 標準化
scaler = StandardScaler()
X_train_scaled = scaler.fit_transform(X_train)
X_test_scaled = scaler.transform(X_test)

# ロジスティック回帰で学習
from sklearn.linear_model import LogisticRegression
logistic = LogisticRegressionCV(cv=10, random_state=0,max_iter=1000)
#logistic = LogisticRegression()
logistic.fit(X_train_scaled, y_train)

# 検証
print('Train score: {:.3f}'.format(logistic.score(X_train_scaled, y_train)))
print('Test score: {:.3f}'.format(logistic.score(X_test_scaled, y_test)))
print('Confustion matrix:¥n{}'.format(confusion_matrix(y_true=y_test, y_pred=logistic.predict(X_test_scaled))))
```

```
Train score: 0.988
Test score: 0.972
Confustion matrix:
[[89  1]
 [ 3 50]]
```

主成分分析で第2主成分まで次元圧縮した説明変数による学習、テストで 94%の精度となり、30 次元から 2 次元まで圧縮したが、まずまずの精度が得られた

```
# パイプラインの作成
from sklearn.pipeline import Pipeline
pca_pipeline = Pipeline([
    ('scale', StandardScaler()),
    ('decomposition', PCA(n_components=2)),
    ('model', LogisticRegressionCV(cv=10, random_state=0))
])

# 標準化・次元圧縮・学習
pca_pipeline.fit(X_train, y_train)

# 検証
print('Train score: {:.3f}'.format(pca_pipeline.score(X_train, y_train)))
print('Test score: {:.3f}'.format(pca_pipeline.score(X_test, y_test)))
print('Confustion matrix:¥n{}'.format(confusion_matrix(y_true=y_test, y_pred=pca_pipeline.predict(X_test))))
```

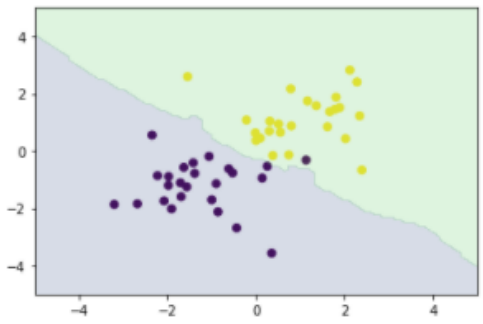
```
Train score: 0.965
Test score: 0.937
Confustion matrix:
[[84  6]
 [ 3 50]]
```

参考：https://ohke.hateblo.jp/entry/2017/08/11/230000

＜アルゴリズム＞
＜k 近傍法＞
与えられたデータに対して、最近傍のデータを k 個とってきて、それらが最も多く属するクラスに分類
教師なし学習

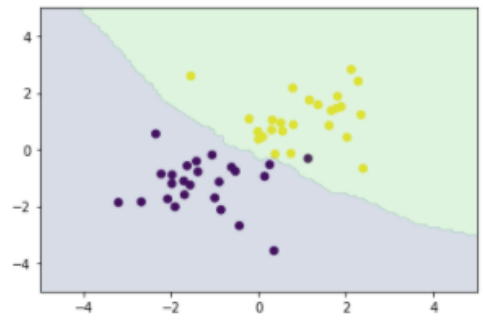
■実装演習結果キャプチャ又はサマリーと考察
コード：np_knn.ipynb
グループA（青色の点）とグループB(黄色の点)に対して、
0.1x0.1 のグリッド点に対し、3近傍で分類した結果

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



10 近傍にすることで境界が滑らかになることを確認

```
n_neighbors = 10

xx0, xx1 = np.meshgrid(np.linspace(-5, 5, 100), np.linspace(-5, 5, 100))
X_test = np.array([xx0, xx1]).reshape(2, -1).T
#print(X_test)
y_pred = knc_predict(n_neighbors, X_train, ys_train, X_test)
plt_resut(X_train, ys_train, y_pred)
```



＜k-means 法＞
要点まとめ
与えられたデータを k 個のクラスタに分類
教師なし学習

■アルゴリズム
1）k 個の各クラスタ中心の初期値を設定する
2）全てのデータ点に対して、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）クラスタ中心が収束するまで 2, 3 の処理を繰り返す

■実装演習結果キャプチャ又はサマリーと考察
コード：skl_kmeans.ipynb
wine データに対して 3 種類のラベルに分類した結果

```
[13]  pd.crosstab(df['labels'], df['species'])
      #クロス集計
```

species	class_0	class_1	class_2
labels			
0	0	50	19
1	46	1	0
2	13	20	29

wine データにラベルを付与

```
[15]  wine_df['labels']=labels
      wine_df.head()
```

	alcohol	malic_acid	ash	alcalinity_of_ash	magnesium	total_phenols	flavanoids	nonflavanoid_phenols	proanthocyanins	color_intensity	hue	od280/od315_of_diluted_wines	proline	labels
0	14.23	1.71	2.43	15.6	127.0	2.80	3.06	0.28	2.29	5.64	1.04	3.92	1065.0	1
1	13.20	1.78	2.14	11.2	100.0	2.65	2.76	0.26	1.28	4.38	1.05	3.40	1050.0	1
2	13.16	2.36	2.67	18.6	101.0	2.80	3.24	0.30	2.81	5.68	1.03	3.17	1185.0	1
3	14.37	1.95	2.50	16.8	113.0	3.85	3.49	0.24	2.18	7.80	0.86	3.45	1480.0	1
4	13.24	2.59	2.87	21.0	118.0	2.80	2.69	0.39	1.82	4.32	1.04	2.93	735.0	2

ラベル0、1、2の各種パラメータの平均値を表示した結果
ラベルの分類には proline が大きく影響している

```
cluster_info = pd.DataFrame()
for i in range(3):
    cluster_info['label' + str(i)] = wine_df[wine_df['labels'] == i].mean()
cluster_info = cluster_info.drop('labels')

my_plot = cluster_info.T.plot(kind='bar', stacked=True, title="Mean Value of 3 Clusters")
#my_plot.set_xticklabels(my_plot.xaxis.get_majorticklabels(), rotation=0)
my_plot.legend(loc='upper right',
           bbox_to_anchor=(1.08, 0.9, 0.5, .100),
           borderaxespad=0.,)
```

```
<matplotlib.legend.Legend at 0x7f227c00edd0>
```

＜SVM＞
要点まとめ

■SVM
要点まとめ
信号処理医療アプリケーションや自然言語処理、音声および画像認識などの多くの分類と回帰の問題に使用される
教師あり学習

＜ハードマージン SVM＞
以降は2クラス分類に適用する場合について記載する。
2クラス分類問題では、特徴ベクトル$\boldsymbol{x}$がどちらのクラスに属するか判定するために次の線形判別関数と呼ばれる関数$f(\boldsymbol{x})$を使う

$$f(\boldsymbol{x}) = \boldsymbol{w}^T\boldsymbol{x} + b \qquad \cdots(1)$$

線形判別関数（1）と最も近いデータ点(サポートベクトル)との距離(マージン)が最大となる線形判別関数、つまりそのような$\boldsymbol{w}$とbを求めることが目的です。
パラメータ$\boldsymbol{w}$とbは訓練データから学習して推定します。

■目的関数
各データ点と決定境界の距離

$$\frac{|\boldsymbol{w}^T\boldsymbol{x} + b|}{||\boldsymbol{w}||} = \frac{t_i(\boldsymbol{w}^T\boldsymbol{x} + b)}{||\boldsymbol{w}||}$$

※点と直線との距離の公式より
https://mathtrain.jp/tentotyokusen

マージンは決定境界と最も距離の近いデータ点との距離から、

$$min_i \frac{t_i(\boldsymbol{w}^T\boldsymbol{x} + b)}{||\boldsymbol{w}||}$$

求めたいのはマージンを最大化することであるため、SVM の目的関数は、

$$max_{w,b}\left[min_i \frac{t_i(\boldsymbol{w}^T\boldsymbol{x} + b)}{||\boldsymbol{w}||}\right]$$

今、この式を以下と置くと

$$max_{w,b}\frac{M(\boldsymbol{w},b)}{||\boldsymbol{w}||} \quad \text{s.t. } t_i(\boldsymbol{w}^T\boldsymbol{x}+b) \geq M(\boldsymbol{w},b)\ (i=1,2,\ldots,n)$$

$\frac{1}{|\boldsymbol{w}|}$の最大化は逆数の$|\boldsymbol{w}|$の最小化と等価であること、$|\boldsymbol{w}|$の最小化は$\frac{1}{2}|\boldsymbol{w}|^2$の最小化と等価であることから、上記最適化問題はさらに扱いやすい形に置き換えることができます。

$$\min_{w,b}\frac{1}{2}||\boldsymbol{w}||^2 \quad \text{s.t. } t_i(\boldsymbol{w}^T\boldsymbol{x}+b) \geq 1\ (i=1,2,\ldots,n)$$

$\boldsymbol{x}$:説明変数
$\boldsymbol{w}$:パラメータ
b:切片
$\boldsymbol{t}$:分類ラベル(0,1)

■目的関数の最適化
ラグランジュ乗数法を使うと、上の最適化問題はラグランジュ乗数$\boldsymbol{a}(\geq 1)$を用いて、以下の目的関数を最小化する問題となる。

$$L(\boldsymbol{w},b,\boldsymbol{a}) = \frac{1}{2}\left|\left|\boldsymbol{w}\right|\right|^2 - \sum_{i=1}^{n} a_i(\,t_i(\boldsymbol{w}^T x_i + b) - 1) \qquad \cdots(2)$$

目的関数が最小となるのは、$\boldsymbol{w}$、bに関して偏微分した値が 0 となるときなので、

$$\frac{\partial L}{\partial \boldsymbol{w}} = \boldsymbol{w} - \sum_{i=1}^{n} a_i t_i x_i = 0 \ \rightarrow\ \boldsymbol{w} = \sum_{i=1}^{n} a_i t_i x_i \qquad \cdots(3)$$

$$\frac{\partial L}{\partial b} = \sum_{i=1}^{n} a_i t_i = \boldsymbol{a}^{\mathrm{T}} t = 0 \qquad \cdots(4)$$

$$a_i = 0 \quad \textit{または、}\ t_i(\boldsymbol{w}^T x_i + b) - 1) = 0 \qquad \cdots(5)$$

しかしこれらの式だけでは$\boldsymbol{a}$を決めることができません。そこでこれまでの最適化問題とは別の$\boldsymbol{a}$に関する最適化問題(双対問題)を導き、これを解くことにより$\boldsymbol{a}$を求めます。
$\boldsymbol{w} = \sum_{i=1}^{n} a_i t_i x_i$なので、これを$L(\boldsymbol{w},b,\boldsymbol{a})$に代入した関数を

$$\tilde{L}(\boldsymbol{a}) = L\left(\sum_{i=1}^{n} a_i\, t_i x_i, b, \boldsymbol{a}\right)$$

と定義する。

■最適化問題（双対問題）
目的関数$\tilde{L}(\boldsymbol{a})$を以下の制約条件の下で、$\boldsymbol{a}$に関して最大化することを考える。
（3）の

$$\boldsymbol{w} = \sum_{i=1}^{n} a_i t_i x_i$$

と、（4）の制約条件

$$\sum_{i=1}^{n} a_i y_i = \boldsymbol{a}^{\mathrm{T}} \boldsymbol{y} = 0 \quad \text{s.t. } \boldsymbol{a} \geqq 0, \ \ i = 0, \ldots, n$$

を式(2)に代入することで、最適化問題は結局以下の目的関数の最大化となる。

$$L(\boldsymbol{w}, b, \boldsymbol{a}) = \frac{1}{2}\left(\sum_{i=1}^{n} a_i t_i x_i\right)^{\mathrm{T}} \left(\sum_{i=1}^{n} a_i t_i x_i\right) - \sum_{i=1}^{n} a_i \left(t_i \left(\left(\sum_{i=1}^{n} a_i t_i x_i\right)^{\mathrm{T}} x_i + b \right) - 1 \right)$$

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

$$= \boldsymbol{a}^{\mathrm{T}}\boldsymbol{1} - \frac{1}{2}\boldsymbol{a}^{\mathrm{T}}\boldsymbol{H}\boldsymbol{a} \qquad \cdots (6)$$

ただし、行列 $\boldsymbol{H}$ の i 行 j 列成分は$H_{ij} = t_i t_j {x_i}^{\mathrm{T}} x_i$、$1 = (1, \ldots, 1)^T$である。
また制約条件は、

$$\boldsymbol{a}^{\mathrm{T}}\boldsymbol{t} = 0 \left(\frac{1}{2}\left|\left|\boldsymbol{a}^T\boldsymbol{t}\right|\right|^2 = 0\right)$$

である。
この最適化問題を最急降下法で解く。目的関数と制約条件を、$\boldsymbol{a}$で微分すると、

$$\frac{d\tilde{L}}{d\boldsymbol{a}} = \boldsymbol{1} - \boldsymbol{H}\boldsymbol{a}$$

$$\frac{d}{d\boldsymbol{a}}\left(\frac{1}{2}\left|\left|\boldsymbol{a}^T\boldsymbol{t}\right|\right|^2\right) = (\boldsymbol{a}^T\boldsymbol{t})\boldsymbol{t}$$

なので、$\boldsymbol{a}$ を以下の二つの式で更新する。

$$\boldsymbol{a} \leftarrow \boldsymbol{a} + \eta_1(\boldsymbol{1} - \boldsymbol{H}\boldsymbol{a})$$

$\boldsymbol{a} \leftarrow \boldsymbol{a} - \eta_2(\boldsymbol{a}^T\boldsymbol{t})\boldsymbol{t}$

双対問題を解いて得られたラングランジュ未定乗数の最適解を、$\tilde{a} = (\widetilde{a_1}, \ldots, \widetilde{a_n})^{\mathrm{T}}$とすると、(3)より最適解は次のように求められます。ここで、(5)より、x_iがサポートベクトルでない場合は、$\widetilde{a_i}$が0となるため和をとるのはサポートベクトルのみとなる。

$$w = \sum_{i=1}^{n} \widetilde{a_i} t_s x_s$$

(5)より、x_iがサポートベクトルの場合、$t_i(\boldsymbol{w}^T x_i + b) - 1) = 0$であるため、

$$b_s = \frac{1}{t_s} - w^T x_s$$

ここで、x_s, t_sはサポートベクターのデータを表す。
切片の誤差を小さくするためすべてのサポートベクトルで平均をとると、

$$b = \frac{1}{|V_s|} \sum_{s \in V_s} b_s = \frac{1}{|V_s|} \sum_{s \in V_s} (t_s - \boldsymbol{w}^T x_s)$$

ここで、V_sはサポートベクトルの数を表す。

＜ソフトマージン SVM＞

データを完璧に線形分離できない時、誤差ξ_iを許容して誤差に対してペナルティを与える。

$$\xi_i = 1 - t_i(\boldsymbol{w}^T\boldsymbol{x} + b)$$

線形分離できない場合でも対応でき、パラメータCの大小で決定境界が変化する。

パラメータCが大きいと誤差をあまり許容せず、小さいと誤差をより許容する。

■目的関数

$$\min_{w,b}\{\frac{1}{2}||\boldsymbol{w}||^2 + C\sum_{i=1}^{n}\xi_i\} \quad \text{s.t. } t_i(\boldsymbol{w}^T\boldsymbol{x} + b) \geq 1 - \xi_i, \quad \xi_i \geq 0 \ (i = 1,2,\ldots,n)$$

$\frac{1}{2}||w||^2$：マージンの大きさ

C：誤差に対するペナルティを制御するパラメータ

$\sum_{i=1}^{n}\xi_i$：誤差に対するペナルティ

■最適化問題（双対問題）

目的関数の最大化

$$\tilde{L}(\boldsymbol{a}) = \sum_{i=1}^{n} a_i - \frac{1}{2}\sum_{i=1}^{n}\sum_{j=1}^{n} a_i a_j t_i t_j x_i{}^{\mathrm{T}} x_i \qquad \cdots(7)$$

制約条件：

$$\sum_{i=1}^{n} a_i\, t_i = 0, \quad 0 \leqq a_i \leqq C, \quad i = 1,2,\ldots n$$

＜非線形分離＞
線形分離できない時、特徴空間に写像してその空間で線形に分離する。
特徴空間上では線形に分離した結果は、元の空間上では非線形な分離となる。

■カーネルトリック
非線形分離を可能とする手法。
ソフトマージンでは必ずしも良い性能になるとは限らないが、カーネルトリックは特徴ベクトルを非線形変換してその空間上で線形の識別を行う。

カーネル関数を使い、高次元ベクトルの内積をスカラ関数で表現する。
RBF カーネル(ガウシアンカーネル)、多項式カーネル、シグモイドカーネルなどを用いる。

■非線形判別関数

$$f(x) = \boldsymbol{w}^T \phi(\boldsymbol{x}) + b$$

■最適化問題（双対問題）
目的関数の最大化

$$\tilde{L}(a) = \sum_{i=1}^{n} a_i - \frac{1}{2}\sum_{i=1}^{n}\sum_{j=1}^{n} a_i a_j t_i t_j \phi(x_i)^{\mathrm{T}} \phi(x_j) \qquad \cdots (8)$$

制約条件:

$$\sum_{i=1}^{n} a_i\, t_i = 0, \quad 0 \leqq a_i \leqq C, \quad i = 1,2, \ldots n$$

■カーネルトリック
非線形に写像した空間での二つの要素$\phi(x_i)^{\mathrm{T}}$と$\phi(x_j)$の内積が、

$$\phi(x_i)^{\mathrm{T}} \phi(x_j) = \boldsymbol{K}(x_i, x_j)$$

のように、入力特徴 x_iと x_jのみから計算できるなら、非線形写像によって変換された特徴空間での特徴$\phi(x_i)^{\mathrm{T}}$や$\phi(x_j)$を計算する代わりに、$\boldsymbol{K}(x_i, x_j)$から最適な非線形写像を構成できる。ここで、このような$\boldsymbol{K}$のことをカーネルと呼ぶ。
このように高次元に写像しながら、実際には写像された空間での特徴の計算を避けて、カーネルの計算のみで最適な識別関数を構成するテクニックのことを「カーネルトリック」と呼ぶ

よく使用する3つのカーネル関数

・多項式カーネル　$K(x_i, x_j) = (x_i^T x_j + c)^d$

・ガウスカーネル　$K(x_i, x_j) = exp\left\{ -\frac{||x_i - x_j||^2}{2\sigma^2} \right\}$

・シグモイドカーネル　$K(x_i, x_j) = tanh(bx_i^T x_j + c)$

カーネルトリックを使用することで、(7)は以下に変形できる。

$$\tilde{L}(a) = \sum_{i=1}^{n} a_i - \frac{1}{2}\sum_{i=1}^{n}\sum_{j=1}^{n} a_i a_j t_i t_j \boldsymbol{K}(x_i, x_j) \qquad \cdots (9)$$

■実装演習結果キャプチャ又はサマリーと考察

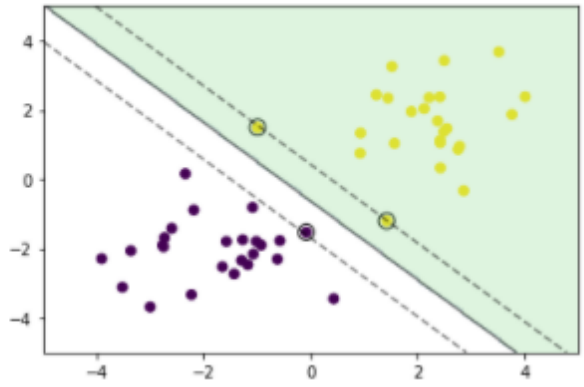
コード：np_svm.ipynb

1．線形分離可能な場合

式（6）の最適化問題を解くことによりマージンと決定境界が求まる

```
# 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')

support_vectors
```

```
array([[-0.0843424 , -1.53020394],
       [ 1.42302539, -1.20317626],
       [-0.99657157,  1.49451368]])
```



2. 線形分離不可能な場合

式(9)のカーネルとして RBF カーネル(ガウシアンカーネル)を使い、特徴空間上での線形分離を行う。

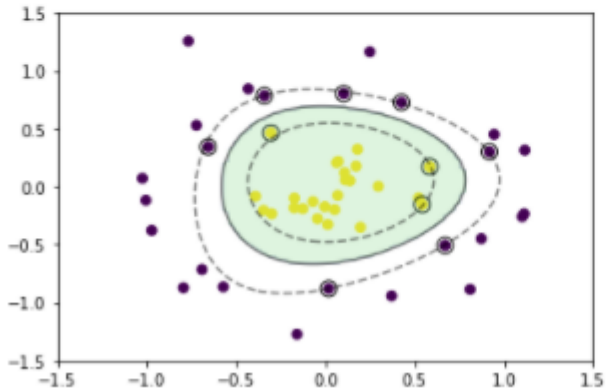
```
[23]  def rbf(u, v):
              sigma = 0.8
              return np.exp(-0.5 * ((u - v)**2).sum() / sigma**2)

      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)
```


```
[23]  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)
```

式(9)の最適化問題を解くことによりマージンと決定境界が求まる

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

```
<matplotlib.contour.QuadContourSet at 0x7f28e9ed2ad0>
```



3. 誤判定を許容し線形分離する場合

誤判定を許容するため、式(7)の制約条件、$0 \leqq a_i \leqq C$となる誤差を制御するパラメータCを設定する。

```
[32] 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)
     #a = np.where(a > 0, a, 0)
```

制約条件の下で式(7)の最適化問題を解くことによりマージンと決定境界が求まる

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

```
<matplotlib.contour.QuadContourSet at 0x7f4321c36790>
```