実装演習レポート（機械学習）

owadatk edited this page 1 minute ago · 18 revisions

実装演習レポート（機械学習）

線形回帰モデル

要点のまとめ

全体像

- 機械学習モデリングプロセス：どのような課題を機械学習で解くかという入り口での吟味が重要
- 教師あり学習（予測と分類）・教師なし学習（クラスタリングと次元削減）：それぞれモデルと解くべき推定問題との対応た
- モデルの評価（ホールドアウト法と交差検証法）

線形回帰モデル

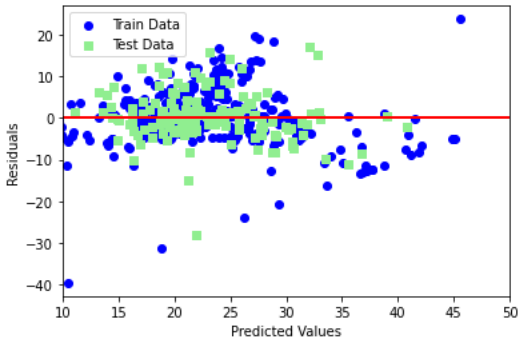
- 入力とm次元パラメータの線形結合を出力するモデル（単回帰m=1と重回帰m>1がある）
- 線形結合とは入力とパラメータの内積である（出力は1次元のスカラ）
- 未知パラメータは最小二乗法により推定（残差平方和が最小となるパラメータを探索：勾配0の点）
- データは学習用と検証用に分割する→モデルの汎化性能を測定するため

実装演習結果

- **E資格ではnumpyでの実装を問われるので、sklearnだけでなくnumpyの実装例も確認すること！！**
- **コードと数式の対応関係を考えながら実行することが重要**
- データの中身の確認 **（要約統計量をとって外れ値を確認したり、傾向を確認したりする）**
  - print(boston) # データの全部を表示
  - print(boston['DESCR']) # DESCRの中身を表示（辞書型リストからDESCRをキーとして呼び出す）JSON形式
  - 'feature_names','data','target'についても同様
- データフレームの作成
  - df(data=,columns=)でデータフレームにデータを流し込み、データ項目名を付与する
  - df['PRICE']=np.array(boston.target)で'target'のデータ配列を'PRICE'という列として追加
  - df.head(5) #先頭の5行を表示して中身を確認
- 線形単回帰分析
  - data=df.loc[:,['RM']].values # 列ラベルを使って値を取得し、dataに格納
  - targetも同様
  - LinearRegressionをインポートしてオブジェクト生成（from sklearn.linear_model import LinearRegression）
  - model.fit(data, target) # fit関数でパラメータ推定
  - model.predict(1) # predict関数で予測：1部屋の場合の価格は$-25,569と予測（外挿のため変な数字になったか？）
- 重回帰分析（2変数）

    - 変数が2つになる以外は単回帰と同様
    - **説明変数を動かして、モデルが想定通りの挙動をするかを確かめることが重要**
- 回帰係数と切片の値を確認
    - 'coef_'と'intercept_'で表示
- モデルの検証
    - 訓練用データとテスト用データを分割するにはtrain_test_splitを使う
    - 書き方：X_train, X_test, y_train, y_test = train_test_split(data, target, test_size = 0.3, random_state = 666)
    - matplotlibを用いて残差をグラフ表示
    - MSEとR^2の表示

```
# 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', marker = 'o', label = 'Train Data')
plt.scatter(y_test_pred, y_test_pred - y_test, c = 'lightgreen', marker = '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()
```




```
* pyplotのscatterを使用
* y軸は y_train_pred - y_train と残差を計算している
```

非線形回帰モデル

要点のまとめ

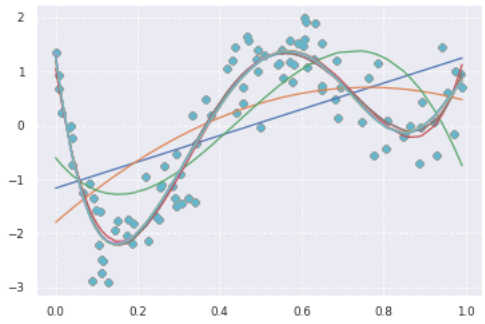
- 基底展開法：回帰関数として、基底関数と呼ばれる既知の非線形関数φ(x)とパラメータベクトルの線形結合を使用（重みにつ形:＝linear-in-parameter）
- 未知のパラメータは線形回帰モデルと同様に最小二乗法や最尤法により推定
    - 多項式関数
    - ガウス型基底関数
    - スプライン関数／Bスプライン関数
- 未学習と過学習（変数を削除したり（特徴選択：ドメイン知識、AICによるモデル選択、交差検証法での選択）**正則化**を利用して表現力を抑止する方法がある）
    - L1ノルム（Lasso推定量）いくつかのパラメータを正確に0に推定

  - L2ノルム（Ridge推定量）パラメータを0に近づけるよう推定
  - 正則化パラメータをどうするか？
- 適切なモデル（汎化性能が高いモデル）は**交差検証法**で決定（ホールドアウト法と比較すると外れ値の影響を受けにくい）※**ータを使った精度（例えばMSE）を報告する**（訓練データを使った精度では汎化性能を評価できない）
- グリッドサーチ：最もよい評価値をもつハイパーパラメータの組合せを探索する（※実装の練習はした方がよいが……実践でズ最適化がNNで使われることが多い）

実装演習結果

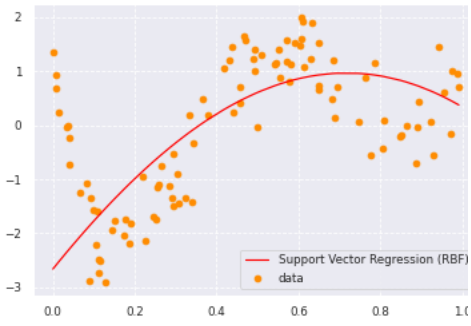
- true_func()はxに関する4次式：　1 - 48x + 218x^2 - 315x^3 + 145x^4
- clf (classifier) はLinearRegression()
- reshape(-1,1)は2次元の縦ベクトルに変形している。これは、**fitやpredictに渡す引数は（要素の数,1）の形にする必要がある**
- LinearRegression.score()は相関係数を出力
- KernelRidge回帰はL2正則化付き最小二乗線形回帰。パラメータalphaで正則化の強度を指定し、kernelで適用する関数の形を。rbfはradial basis function kernel（放射基底関数カーネル）
- 【参考】Lasso回帰による線形回帰分析とカーネル法による非線形データへの適用

```
#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))
```



- deg変数は多項式の次数
- 【参考】機械学習パイプラインによる多項式回帰
- 多項式の特徴を線形回帰に適用する部分はpipelineを使ってラップ **[後で調べる！]**

```
from sklearn import model_selection, preprocessing, linear_model, svm

# SVR-rbf
clf_svr = svm.SVR(kernel='rbf', C=1e3, gamma=0.1, epsilon=0.1)
clf_svr.fit(data, target)
y_rbf = clf_svr.fit(data, target).predict(data)

# plot

plt.scatter(data, target, color='darkorange', label='data')
plt.plot(data, y_rbf, color='red', label='Support Vector Regression (RBF)')
plt.legend()
plt.show()
```



- サポートベクトル回帰（SVR）
- 引数：C（正則化）、gamma（）、epsilon（誤差のチューブ）　**[後で調べる！]**
- これらのハイパーパラメータはグリッドサーチで最適な組み合わせを探る

```
from keras.callbacks import EarlyStopping, TensorBoard, ModelCheckpoint

cb_cp = ModelCheckpoint('/content/drive/My Drive/study_ai_ml/skl_ml/out/checkpoints/weights.{epoch:02d}-{val_loss:.2f}.hdf5', verbose=1, save_weights_on
cb_tf  = TensorBoard(log_dir='/content/drive/My Drive/study_ai_ml/skl_ml/out/tensorBoard', histogram_freq=0)
```

```
def relu_reg_model():
    model = Sequential()
    model.add(Dense(10, input_dim=1, activation='relu'))
    model.add(Dense(1000, activation='relu'))
    model.add(Dense(1000, activation='relu'))
    model.add(Dense(1000, activation='relu'))
    model.add(Dense(1000, activation='relu'))
    model.add(Dense(1000, activation='relu'))
    model.add(Dense(1000, activation='relu'))
    model.add(Dense(1000, activation='relu'))
    model.add(Dense(1000, activation='linear'))
#     model.add(Dense(100, activation='relu'))
#     model.add(Dense(100, activation='relu'))
#     model.add(Dense(100, activation='relu'))
#     model.add(Dense(100, activation='relu'))
    model.add(Dense(1))

    model.compile(loss='mean_squared_error', optimizer='adam')
    return model
```

```
from keras.models import Sequential
from keras.layers import Input, Dense, Dropout, BatchNormalization
from keras.wrappers.scikit_learn import KerasRegressor

# use data split and fit to run the model
estimator = KerasRegressor(build_fn=relu_reg_model, epochs=100, batch_size=5, verbose=1)

history = estimator.fit(x_train, y_train, callbacks=[cb_cp, cb_tf], validation_data=(x_test, y_test))
```

- 【何をやっているかわからない！学習が進んでから戻って考える！】
- ReLUを使ったディープラーニングで非線形回帰をしているが、SVRよりも良い結果にはなっていないような気がする

ロジスティック回帰モデル

要点のまとめ

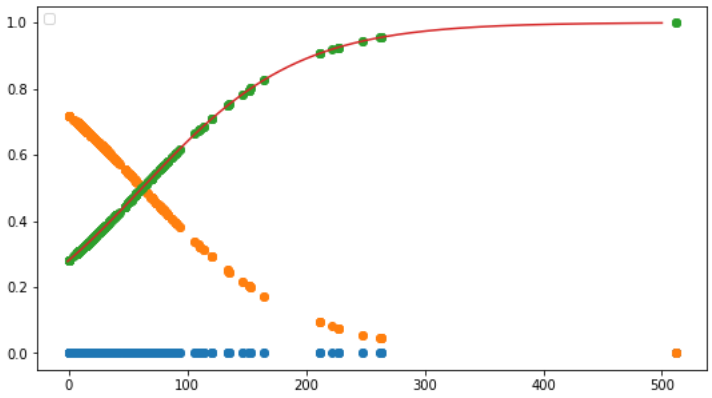
- Using ML to Predict Parking Difficulty (Google AI Blog, Feb. 2017)
- 分類タスク：入力はm次元ベクトル、出力は 0 or 1 の値
  - 識別的アプローチ（例：ロジスティック回帰） 識別完成の構成もある（SVMなど）
  - 生成的アプローチ（ベイズの定理を一回かませる）
- シグモイド関数：出力をy=1になる確率に変換する **（実数全体から[0,1]へつぶす）**
- シグモイド関数の微分はシグモイド関数自身で表現可能⇒尤度関数の微分を行う際に計算が容易
- 最尤推定
  - データからそのデータを生成したであろう尤もらしい分布（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
  - 計算の便宜のため（**桁落ちを防ぐため**）対数をとって対数尤度関数とし、**「尤度関数にマイナスを掛けたものを最小化し二乗法の最小化と合わせる」**
- 勾配降下法：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ひとつ。学習率ηで収束しやすさを調整
- 確率的勾配降下法
- 分類の評価方法：混同行列が基本。タスクに応じて適切な評価方法を選択する。
- 正解率、再現率（recall=TP/(TP+FN)）、適合率（Precision=TP/(TP+FP)）、F値（再現率と適合率の調和平均）
- Recall：本当にPositiveなものの中からPositiveと予測できる割合。False Positiveが多少あっても抜け漏れは少ない予測をしたい利用（例：病気の診断）
- Precision：モデルがPositiveと予測したものの中で本当にPositiveである割合。見逃しが多くてもより正確な予測をしたい場合（例：スパムメール検知）

- F値：タスクが分からない場合

実装演習結果

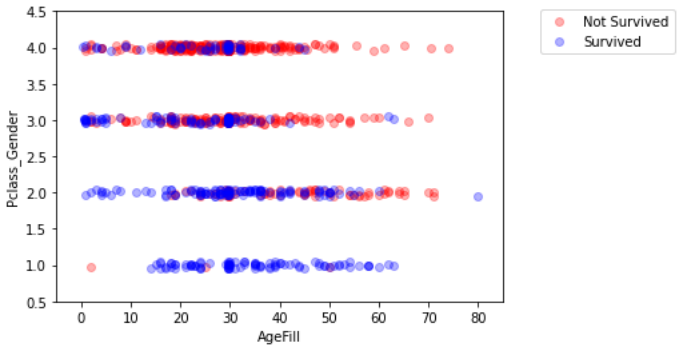
- %matplotlib inline #plt.show()しなくてもinlineでグラフを表示できる。覚えていると時短になるので便利。
- pd.read_csv('FILE_PATH')* 不要なデータの削除：dropメソッド（axis=1で列方向の削除、inplace=trueは上書きオプション（デフォルトはfalse））
- isnull()は欠損値の確認。any(axis=1)で行方向に 1 つでも欠損値があればTrueを返す。
- fillna()　引数の値で欠損値を埋める
- 【参考】『前処理大全』
- logistic回帰はfrom sklearn.linear_model import LogisticRegression
  - 裏で回っている関数のデフォルト値に注意（何が前提とされているか？）
  - 例えば、C=1.0（正則化）、penalty='l2'（L2ノルムを指定）、solver='lbfgs'（最適化）とか
- _proba()で確率を返す（※数学的背景を理解していれば、このようなメソッドがあることは予想できるはず）
- map({'female':0, 'male':1})　質的変数（性別）を 0 or 1 にエンコーディング

```
w_0 = model.intercept_[0]
w_1 = model.coef_[0,0]

# def normal_sigmoid(x):
#     return 1 / (1+np.exp(-x))

def sigmoid(x):
    return 1 / (1+np.exp(-(w_1*x+w_0)))

x_range = np.linspace(-1, 500, 3000)

plt.figure(figsize=(9,5))
#plt.xkcd()
plt.legend(loc=2)


# plt.ylim(-0.1, 1.1)
# plt.xlim(-10, 10)

# plt.plot([-10,10],[0,0], "k", lw=1)
# plt.plot([0,0],[-1,1.5], "k", lw=1)
plt.plot(data1,np.zeros(len(data1)), 'o')
plt.plot(data1, model.predict_proba(data1), 'o')
plt.plot(x_range, sigmoid(x_range), '-')
#plt.plot(x_range, normal_sigmoid(x_range), '-')
#
```



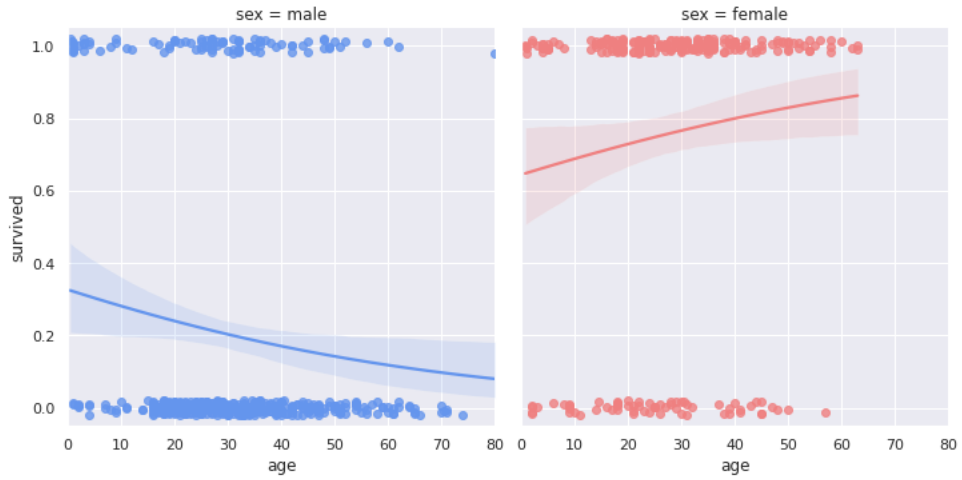
* 変数w0, w1を初期化
* シグモイド関数の定義：ネイピア数の肩に乗っている部分が w0 + w1*x という線形式になっている
* -1 から 500 までの間で3000個の数を生成する
* plot(data1, np.zeros(len(data1)),'o') # チケット価格をxとし、y=0にプロット
* plot(data1, model.predict_proba(data1), 'o') # モデルから予測される確率をプロット
* plot(x_range, sigmoid(x_range), '-') # 上で定義したシグモイド関数をプロット。モデルの確率値と重なっている。

```
np.random.seed = 0

xmin, xmax = -5, 85
ymin, ymax = 0.5, 4.5

index_survived = titanic_df[titanic_df["Survived"]==0].index
index_notsurvived = titanic_df[titanic_df["Survived"]==1].index

from matplotlib.colors import ListedColormap
fig, ax = plt.subplots()
cm = plt.cm.RdBu
cm_bright = ListedColormap(['#FF0000', '#0000FF'])
sc = ax.scatter(titanic_df.loc[index_survived, 'AgeFill'],
                titanic_df.loc[index_survived, 'Pclass_Gender']+(np.random.rand(len(index_survived))-0.5)*0.1,
                color='r', label='Not Survived', alpha=0.3)
sc = ax.scatter(titanic_df.loc[index_notsurvived, 'AgeFill'],
                titanic_df.loc[index_notsurvived, 'Pclass_Gender']+(np.random.rand(len(index_notsurvived))-0.5)*0.1,
                color='b', label='Survived', alpha=0.3)
ax.set_xlabel('AgeFill')
ax.set_ylabel('Pclass_Gender')
ax.set_xlim(xmin, xmax)
ax.set_ylim(ymin, ymax)
ax.legend(bbox_to_anchor=(1.4, 1.03))
```



* index_survived = titanic_df[titanic_df["Survived"==0].index # Survivedの値が0であるデータのインデックスをindex_survivedに
る
* matplotlibでは100種類以上のカラーマップが利用できる。ListedColormpaは円グラフや棒グラフのように数種類の色分けを行いたい場合に
るカラーマップのクラス
* subplots()は返り値figとaxes。中身はfig = figure()した後fig.add_subplot(111)と同じ
* scatterの引数：xはAgeFill、yはPclass_Gender（階級と性別の組合せ。上流階級女性は生存しやすく、下層階級男性は死亡しやすい傾向を
定。）
* (np.random.rand(len(index_survived))-0.5)*0.1 を付け加えることでy軸方向に適度なばらつきをもたせている
* alphaは透明度を指定する引数（0 < alpha < 1）
* Pclass_Genderという特徴量を作ることによって、特徴空間上でうまく線形分離できるようになるケース（※feature engineering：ドメイン
を使った特徴量探索）

```
#Faceted logistic regression

import seaborn as sns
sns.set(style="darkgrid")

# Load the example titanic dataset
df = sns.load_dataset("titanic")

# Make a custom palette with gendered colors
pal = dict(male="#6495ED", female="#F08080")

# Show the survival probability as a function of age and sex
g = sns.lmplot(x="age", y="survived", col="sex", hue="sex", data=df,
               palette=pal, y_jitter=.02, logistic=True)
g.set(xlim=(0, 80), ylim=(-.05, 1.05))
plt.show()
```



* seabornを使っての描画はコードがシンプル
* lmplotの引数：hueは色分け出力、colで表示枠を分ける、y_jitterはy軸方向のランダムノイズ
* logistic=Trueでロジスティック回帰モデルを適用している。デフォルトで信頼区間95%の範囲を併せて表示。

主成分分析

要点のまとめ

- 高次元のデータを低次元に圧縮
  - 情報の損失を小さくするには？⇒分散が最大になるように次元削減する
  - 次元を減らした分析や、可視化（2次元、3次元の場合）
  - 寄与率：第k主成分の分散の全分散に対する割合（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

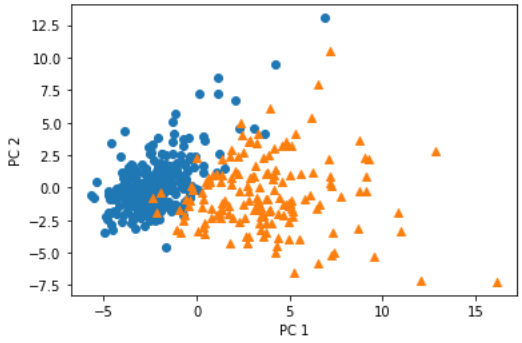
実装演習結果

- 【参考】scikit-learnで乳がんデータセットを主成分分析する
- まず、30次元の特徴量空間でロジスティック回帰を回している
- 目的変数を抽出する際、診断結果がMならば1, M以外であれば0となるように変換している（apply(lambda d:1 if d=='M' else
- StandardScaler()を用いた標準化
  - fit_transform はパラメータ計算とデータ変換をまとめて実行
  - transform はパラメータをもとにデータ変換

    - fit はパラメータ（平均や標準偏差 etc）を計算
- 主成分ごとの寄与率（pca.explained_variance_ratio_）をグラフ化
- 30次元を2次元に圧縮することにより予測性能は落ちるが、可視化することで議論を前に進めることができるのであれば、説

```
# PCA
# 次元数2まで圧縮
pca = PCA(n_components=2)
X_train_pca = pca.fit_transform(X_train_scaled)
print('X_train_pca shape: {}'.format(X_train_pca.shape))
# X_train_pca shape: (426, 2)

# 寄与率
print('explained variance ratio: {}'.format(pca.explained_variance_ratio_))
# explained variance ratio: [ 0.43315126  0.19586506]

# 散布図にプロット
temp = pd.DataFrame(X_train_pca)
temp['Outcome'] = y_train.values
b = temp[temp['Outcome'] == 0]
m = temp[temp['Outcome'] == 1]
plt.scatter(x=b[0], y=b[1], marker='o') # 良性は○でマーク
plt.scatter(x=m[0], y=m[1], marker='^') # 悪性は△でマーク
plt.xlabel('PC 1') # 第1主成分をx軸
plt.ylabel('PC 2') # 第2主成分をy軸
```

```
X_train_pca shape: (426, 2)
explained variance ratio: [0.43315126 0.19586506]
Text(0, 0.5, 'PC 2')
```



として使えることもある。

アルゴリズム

要点のまとめ

- k近傍法：クラスタリング。最近傍のデータをk個とってきて、それらが最も多く所属するクラスに識別
- kを変化させると結果も変わる
- kを大きくすると決定境界は滑らかになる
- k-means法：クラスタリング。与えられたデータをk個のクラスタに分類する
- 中心の初期値を設定⇒各データ点と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⇒収束するまで繰り
- 初期値が近いとうまくクラスタリングできない　⇒k-means++法という方法を使うと初期値がうまく離れる
- 【参考】『見て試してわかる機械学習アルゴリズムの仕組み 機械学習図鑑』

実装演習結果

- scikit-learnのtoy datasetの中からload_wine()を使ってワインの種類の分類タスクを実行する
- data attributeはshape(178,13)のpandas dataframeを返す。target attributeはshape(178,)のpandas配列を返す
- target_names attributeは目的クラスの名称を返す

- model = KMeans(n_clusters=3) クラスター数3 でオブジェクトを生成

```
[8]  model = KMeans(n_clusters=3)
```

```
[9]  labels = model.fit_predict(X)
```

```
[10] df = pd.DataFrame({'labels': labels})
     type(df)
```

```
     pandas.core.frame.DataFrame
```

```
[11] def species_label(theta):
         if theta == 0:
             return wine.target_names[0]
         if theta == 1:
             return wine.target_names[1]
         if theta == 2:
             return wine.target_names[2]
```

```
[12] df['species'] = [species_label(theta) for theta in wine.target]
```

```
pd.crosstab(df['labels'], df['species'])
```

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

サポートベクターマシーン

要点のまとめ

- 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二値分類（確率は出ないので決定論的）
- 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いい、マージンが最大となる線形判別関数を求める
- ラグランジュの未定乗数法：制約付き最適化問題を解くための手法
- KKT条件（カルーシュ・キューン・タッカーの相補性条件）：制約付き最適化問題において最適解が満たす条件
- **【数学的議論は高度すぎるので、余裕があれば後から書籍やネット検索でフォロー】**
- マージンの外側のデータは予測に影響を与えない。マージン上のデータは予測に影響を与える。
- 分離超平面を構成する学習データはサポートベクターだけで、残りのデータは不要（外側のデータはサポートベクターではな
- ソフトマージンSVM：誤差を許容し、誤差に対してペナルティを与える。誤差を表す変数ξ（グザイ）を導入する。
- ペナルティ項のきき具合を制御するパラメータC（小さいときは誤差をより許容し、大きいときは誤差をあまり許容しない）
- 線形分離できないときは特徴空間に写像し、その空間で線形に分離する
- カーネルトリック：高次元ベクトルの内積をスカラー関数で表現。
- 非線形カーネルを用いた分離（RBFカーネル、ガウシアンカーネル）：非線形な分離が可能

実装演習結果

線形分離可能な場合

- 学習
  - t = np.where(ys_train == 1.0, 1.0, -1.0) # ys_trainが1.0の場合はt=1.0, それ以外はt=-1.0
  - n_samples = len(X_train) # サンプル数
  - K = X_train.dot(X_train.T) # X_trainとX_train転置との内積
  - η1 = 0.01, η2 = 0.001, 最急降下法の繰返し回数=500
  - H = np.outer(t, t) * K # 次の式に相当
  - 行列Hのi行j列成分は$H_{ij} = t_i t_j \phi(\boldsymbol{x}_i)^{\mathrm{T}} \phi(\boldsymbol{x}_j) = t_i t_j k(\boldsymbol{x}_i, \boldsymbol{x}_j)$である。
  - a = np.ones(n_samples) # サンプル数と同数の要素をもつ配列aを生成。全要素を1で初期化
  - for文の中は最急降下法であり、次の式に相当
  - $\frac{d\tilde{L}}{d\boldsymbol{a}} = \mathbf{1} - H\boldsymbol{a}$
    $\frac{d}{d\boldsymbol{a}}(\frac{1}{2}||\boldsymbol{a}^{\mathrm{T}}\boldsymbol{t}||^2) = (\boldsymbol{a}^{\mathrm{T}}\boldsymbol{t})\boldsymbol{t}$
    なので、$\boldsymbol{a}$を以下の二式で更新する。
    $\boldsymbol{a} \leftarrow \boldsymbol{a} + \eta_1(\mathbf{1} - H\boldsymbol{a})$
    $\boldsymbol{a} \leftarrow \boldsymbol{a} - \eta_2(\boldsymbol{a}^{\mathrm{T}}\boldsymbol{t})\boldsymbol{t}$
  - ```
    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)
    ```

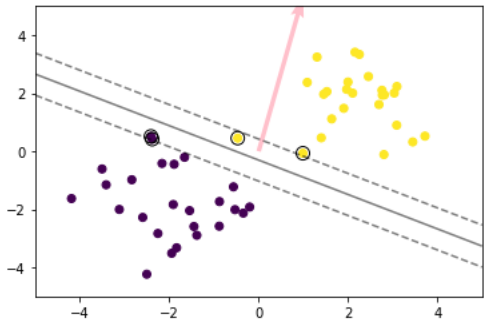
- 予測
  - a > 0 に対応するデータ点のみ保持し、bを求める
  - ここで、最適化の結果得られた$a_i (i = 1, 2, \ldots, n)$の中で$a_i = 0$に対応するデータ点は予測に影響を与えないので、$a_i > 0$に対応するデータ点（サポートベクトル）のみ保持しておく。bはサポートベクトルのインデックスの集合をSとすると、
    $b = \frac{1}{S}\sum_{s \in S}(t_s - \sum_{i=1}^{n} a_i t_i k(\boldsymbol{x}_i, \boldsymbol{x}_s))$によって求める。
  - ```
    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()
    ```
  - 次の式の計算：最終行の y_pred には正負の情報が入る
  - $y(\boldsymbol{x}) = \boldsymbol{w}\phi(\boldsymbol{x}) + b = \sum_{i=1}^{n} a_i t_i k(\boldsymbol{x}, \boldsymbol{x}_i) + b$の正負によって分類する。

```
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)
```

- グラフによる表現

```
# 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')
```

```
<matplotlib.quiver.Quiver at 0x7fc64aa0b8d0>
```



線形分離不可能な場合

- 学習
  - RBFカーネル（ガウシアンカーネル）を用いて特徴空間上で線形分離する
  - 冒頭でrbfカーネルを定義
  - 上述の線形分離可能なSVMでは線形カーネルだった部分がRBFカーネルに置き換わっている
  - 残りの部分は線形カーネルを使った場合と同じ流れ

```
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)
```

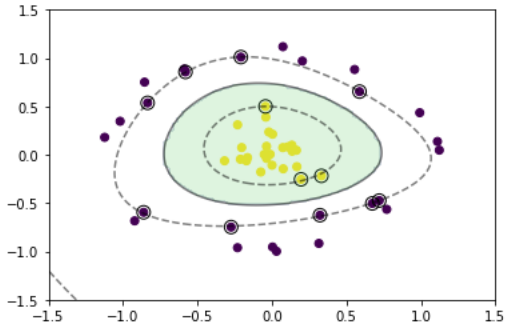
- 予測
  - 流れは線形分離可能な場合と同じ

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

```
<matplotlib.contour.QuadContourSet at 0x7fc6426c9d90>
```



-

ソフトマージンSVM

- 学習
  - スラック変数ξ（グザイ）：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
  - 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C
  - 最終行の制約条件　a = np.clip(a, 0, C) がこれまでと異なる # 配列a に関して最小値0、最大値C
    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。
    $\tilde{L}(\boldsymbol{a}) = \sum_{i=1}^{n} a_i - \frac{1}{2}\sum_{i=1}^{n}\sum_{j=1}^{n} a_i a_j t_i t_j \phi(\boldsymbol{x}_i)^{\mathrm{T}}\phi(\boldsymbol{x}_j)$
  - 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。

```
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)
```

- 予測
  - 予測の流れはハードマージンSVMと同じ

```
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))
```