

機械学習レポート

- 線形回帰モデル

➤ 要点まとめ

そもそも...何かを予測するモデルは大きく「分類」と「回帰」に分けられる。目的変数がカテゴリデータの場合が分類、数値データの場合が回帰。線形回帰モデルは回帰予測をするモデルの一つ。説明変数が一つの場合を単回帰分析、説明変数が複数の場合を重回帰分析という。誤差関数を最小化、あるいは尤度関数を最大化するような回帰係数を探すこととする。

➤ 実装演習結果・考察

単回帰分析

```
In [23]: from sklearn.datasets import load_boston
from sklearn.linear_model import LinearRegression
import pandas as pd
import numpy as np

In [32]: # データを取得
data_raw = load_boston()
df = pd.DataFrame(data=data_raw.data, columns = data_raw.feature_names)
df['PRICE'] = np.array(data_raw.target)

In [34]: # 単回帰用に特徴量を絞る
x = data_raw.data[:,5:6]
y = data_raw.target

# 1次元配列から2次元配列に変換
# x = x.reshape(-1, 1)
# y = y.reshape(-1, 1)

In [36]: # オブジェクト生成
model = LinearRegression()
model.get_params()

Out[36]: {'copy_X': True, 'fit_intercept': True, 'n_jobs': None, 'normalize': False}

In [37]: # パラメータ推定
model.fit(x,y)

Out[37]: LinearRegression(copy_X=True, fit_intercept=True, n_jobs=None, normalize=False)

In [38]: # 予測
model.predict([[7]])

Out[38]: array([41.28836565])
```

重回帰分析(3変数)

```
In [47]: # 重回帰分析
x2 = df.loc[:,['CRIM','RM','TAX']].values
y2 = df.loc[:,['PRICE']].values

In [48]: model2 = LinearRegression()

In [49]: model2.fit(x2,y2)

Out[49]: LinearRegression(copy_X=True, fit_intercept=True, n_jobs=None, normalize=False)

In [51]: model2.predict([[0.4,8,6]])

Out[51]: array([41.28836565])
```

考察

重回帰分析で用いた3変数について、どのように解釈されているのか確認した。

CRIM：小さいほど家賃が高い。犯罪率と家賃の相関イメージと合致。

RM：大きいほど家賃が高い。部屋数と家賃の相関イメージと合致。

TSX：小さいほど家賃が高い。10,000ドル当たりの固定資産税率と家賃の相関イ

イメージがわからぬので何とも言えないところ。ここを理解するにはドメイン知識が足りない。

- 非線形回帰モデル

- 要点まとめ

実際の問題解決の中で、線形回帰モデルの表現力で解決できる問題は限られる。非線形回帰モデルは線形回帰モデルのように説明変数をそのまま X_i として使うのではなく、 X_i^2 や $\sin(X_i)$ など、非線形関数に説明変数を食わせた形で使用する。重みを w_i としたとき、 $Y = w_i * f(X_i)$ で、 $f(x)$ が非線形関数になっているイメージ。これにより、モデルの表現力が高まる。

- 実装演習結果・考察

表現力の高いモデルを用いてパラメータの学習を行う際には、過学習を防ぐための仕組みとして正則化を用いる場合がある。正則化の係数を大きくするほど、複雑なモデルに対し罰則が強く働く。これにより、訓練データに対し過学習をしにくくなる。以下では、サンプルコードとともに正則化の係数を変化させ、どのような予測を行うモデルが作成されるか比較した。

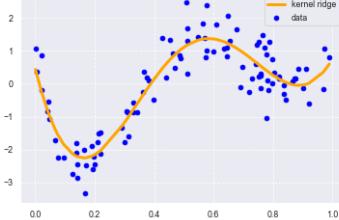
- ✧ KernelRidge alpha = 0.0002 のとき

```
In [28]: from sklearn.kernel_ridge import KernelRidge
clf = KernelRidge(alpha=0.0002, kernel='rbf')
clf.fit(data, target)

p_kridge = clf.predict(data)

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

Out[28]: <matplotlib.legend.Legend at 0x2610d695e10>
```



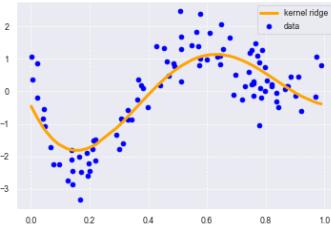
- ✧ KernelRidge alpha = 0.002 のとき

```
In [28]: from sklearn.kernel_ridge import KernelRidge
clf = KernelRidge(alpha=0.002, kernel='rbf')
clf.fit(data, target)

p_kridge = clf.predict(data)

plt.scatter(data, target, color='blue', label='data')
plt.plot(data, p_kridge, color='orange', linestyle='-', linewidth=3, markersize=0, label='kernel ridge')
# plt.plot(data, p, color='orange', marker='o', linestyle='--', linewidth=1, markersize=6)
```

Out [28]: <matplotlib.legend.Legend at 0x2610d7096a0>



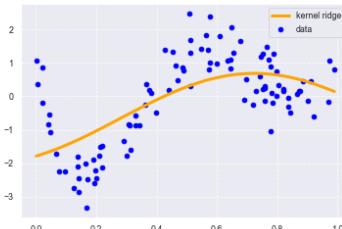
✧ KernelRidge alpha = 0.2 のとき

```
In [30]: from sklearn.kernel_ridge import KernelRidge
clf = KernelRidge(alpha=0.2, kernel='rbf')
clf.fit(data, target)

p_kridge = clf.predict(data)

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

Out [30]: <matplotlib.legend.Legend at 0x2610d533e48>



以上より、KernelRidge alpha を大きくするほど、予測した関数がシンプルなものになっていることがわかる。

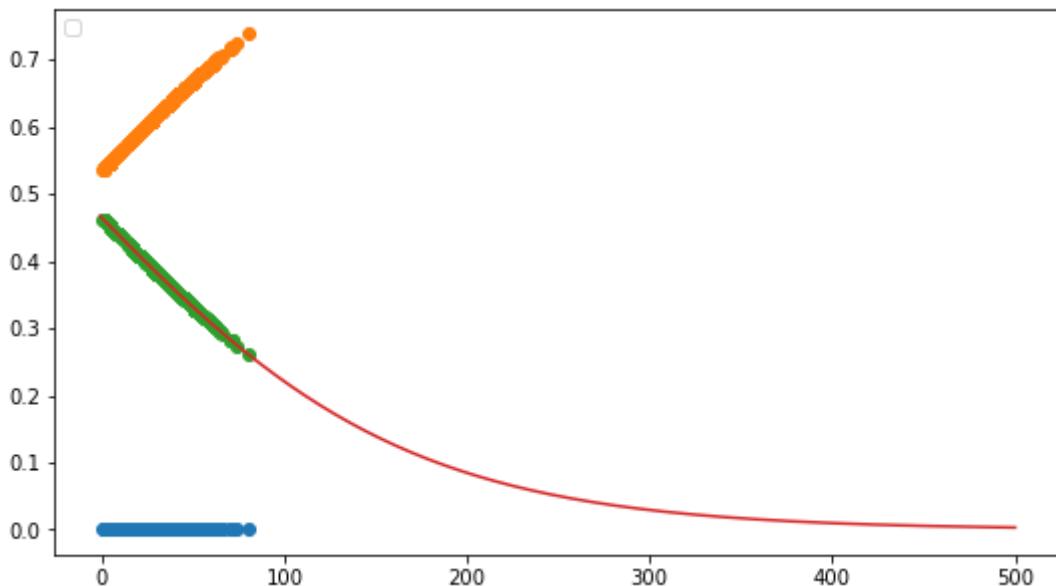
- ロジスティック回帰モデル

- 要点まとめ

ロジスティック回帰モデルは、分類タスクに用いるモデル。二値分類、多項分類どちらにも対応可能。入力と重みの線形結合をシグモイド関数に入力することで、0 ~ 1 の出力を得る。この出力を確率として解釈することで分類タスクを解くことができる。学習の際には尤度関数を最大化するようなパラメータが探索される。解析的なアプローチで最適なパラメータを求めることが困難なため、勾配降下法や確率的勾配降下法という手法によりパラメータの探索が行われる。確率的勾配降下法では、非常に量の多いデータセットを扱う場合でも最低限の計算量で学習を進めることが可能。

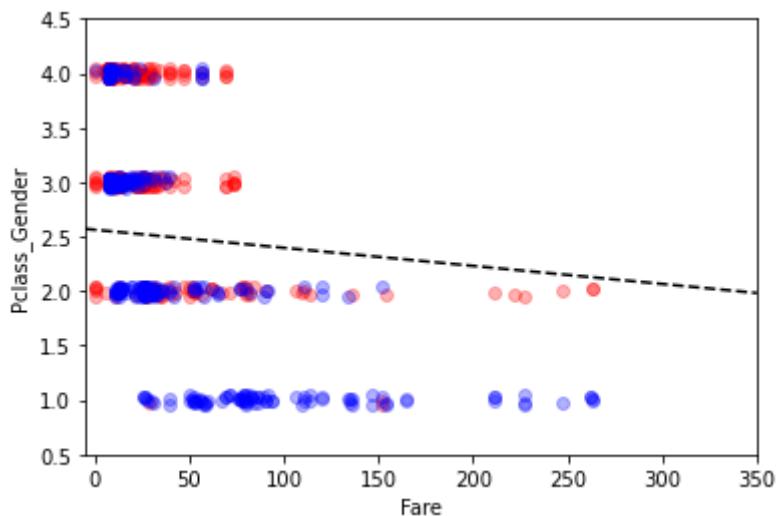
➤ 実装演習結果・考察

一変数のロジスティック回帰で用いる項目を年齢(AgeFill)に変更して実施。



年齢についても高いほど生存率が高いという結果になった。タイタニックの映画を見る限りでは、正しいように思える。

二変数を用いたロジスティック回帰の項目を、「客室等級+性別」「年齢」から「客室等級+性別」「運賃」に変更した。(客室等級と運賃には相関があるのであまり良い例ではないですが・・・)



- 主成分分析

➤ 要点まとめ

次元削減手法の一つ。データの特徴をできる限り失わずに次元数を減らすことが

できる。入力データのばらつきが最も大きくなる低次元空間に入力データを射影する。データの特徴を残したまま次元削減ができれば、計算量の削減につなげることができる。

➤ 実装演習結果・考察

演習の例では乳がんに関する 30 次元のデータを 2 次元に圧縮した。

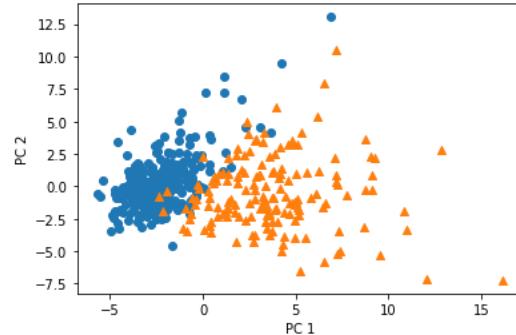
```
In [14]: # PCA
# 次元数2まで圧縮
pca = PCA(n_components=2)
X_train_pca = pca.fit_transform(X_train_scaled)
print('X_train_pca shape: {}'.format(X_train_pca.shape))
# X_train_pca shape: (426, 2)

# 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O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]
```

Out[14]: Text(0, 0.5, 'PC 2')



2 次元にまで次元圧縮した場合の検証精度は 91.6 パーセント

```
In [17]: # テストデータも次元削減
X_test_pca = pca.fit_transform(X_test_scaled)

In [18]: # 次元削減後のデータでロジスティック回帰で学習
logistic = LogisticRegressionCV(cv=10, random_state=0)
logistic.fit(X_train_pca, y_train)

# 検証
print('Train score: {:.3f}'.format(logistic.score(X_train_pca, y_train)))
print('Test score: {:.3f}'.format(logistic.score(X_test_pca, y_test)))
print('Confusion matrix:\n{}'.format(confusion_matrix(y_true=y_test, y_pred=logistic.predict(X_test_pca))))
```

Train score: 0.965

Test score: 0.916

Confusion matrix:

[[89 7]]

[5 48]]

15 次元の場合は 95.1% となった。

```
In [13]: # テストデータも次元削減
X_test_pca = pca.fit_transform(X_test_scaled)

In [14]: # 次元削減後のデータでロジスティック回帰で学習
logistic = LogisticRegressionCV(cv=10, random_state=0)
logistic.fit(X_train_pca, y_train)

# 検証
print('Train score: {:.3f}'.format(logistic.score(X_train_pca, y_train)))
print('Test score: {:.3f}'.format(logistic.score(X_test_pca, y_test)))
print('Confusion matrix:\n{}'.format(confusion_matrix(y_true=y_test, y_pred=logistic.predict(X_test_pca))))
```

Train score: 0.988
Test score: 0.951
Confusion matrix:
[[86 4]
[3 50]]

必要な計算量に対して十分な計算力と時間がある場合は次元削減せず、そのままのデータを使ったほうが精度がよいようである。

- アルゴリズム

- 要点まとめ(k-近傍法・k-mean)

k 近傍法は分類タスクを解くための教師あり学習モデル。分類したい新データに順に k 個のデータを取得し、それらがどの分類に属するか調べる。最も多い分類に新データも属していると判定する。k の数を大きくするほど、分類の境界はなめらかになる。

k-mean はクラスタリングタスクを解くための教師なし学習モデル。与えられたデータを k 個のクラスタに分類する。学習のステップは以下 4 ステップ

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

※講義資料より

初期値によって、うまくクラスタリングできない場合がある。

この問題を解決した kernel k-means というアルゴリズムもある。

- 実装演習結果・考察

演習問題の内容に加え、wine のデータを標準化してから k-means にかけた。

Out[14]:

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

In [21]:

```
# データを標準化して精度の変化を見る
from sklearn.preprocessing import StandardScaler

scaler = StandardScaler()
kmeans = KMeans(n_clusters=3)
from sklearn.pipeline import make_pipeline
pipeline = make_pipeline(scaler,kmeans)
pipeline.fit(X)
labels = pipeline.predict(X)
df = pd.DataFrame({'labels':labels})
df['species'] = [species_label(theta) for theta in wine.target]
ct = pd.crosstab(df['labels'],df['species'])
```

Out[21]:

species	class_0	class_1	class_2
labels			
0	59	3	0
1	0	65	0
2	0	3	48

Out[14]が標準化前の結果。Out[21]が標準化後の結果。標準化により、クラスタリングの精度が大きく上昇していることがわかる。

- サポートベクターマシン

- 要点まとめ

2010年代に深層学習がブレークスルーを迎えるまでは機械学習の中で最も注目を集めていた教師あり学習モデル。回帰、分類両方のタスクに使うことができ、汎化性能が高い。学習データをもっともよく分離する超平面を導くことで、タスクを解決する。もっともよく分離する超平面を導くことを、マージン最大化と呼ぶ。線形分離できない問題についても、データを高次元に拡張解釈し分離することができる。その場合には計算量を削減するため、カーネルトリックと呼ばれる手法が用いられる。

- 実装演習結果・考察

配布された SVR のコードにグリッドサーチを追加した。C,gamma,epsilonについて探索を行い、最も精度が高かったのは C=100000, epsilon=0.1, gamma=1 の時であった。

```
from sklearn import model_selection, preprocessing, linear_model, svm
from sklearn.metrics import f1_score
from sklearn.model_selection import GridSearchCV

#トレーニングデータ、テストデータの分離
train_X, test_X, train_y, test_y = train_test_split(data, target, random_state=0)

# 次元を調整
train_y = np.reshape(train_y, (-1))
test_y = np.reshape(test_y, (-1))

#条件設定
params = {"C": [10 ** i for i in range(-5, 6)],
           "gamma": [10 ** i for i in range(-5, 6)],
           "epsilon": [10 ** i for i in range(-5, 6)]}

# グリッドサーチ実行
clf = GridSearchCV(svm.SVR(), params)
clf.fit(train_X, train_y)
print("学習モデル=", clf.best_estimator_)

# 検証
pred = clf.predict(test_X)
score = clf.score(test_X, test_y)
print("正解率=", score)
```

学習モデル= SVR(C=100000, cache_size=200, coef0=0.0, degree=3, epsilon=0.1, gamma=1,
kernel='rbf', max_iter=-1, shrinking=True, tol=0.001, verbose=False)
正解率= 0.9300848444764295