機械学習　レポート

機械学習　要点まとめ

・線形回帰モデル

　最小二乗法を用いて線形近似することは、エクセル等でもなじみがある。機械学習のアルゴリズムとして重要なのは、二次元空間から拡張しても用いることができることで、行列を用いた偏微分方程式を解くことで、誤差が極小を取る重みを決定することができる。

・非線形回帰モデル

　曲線とフィッティングすることができる一方、正しくモデルを設定する必要があるため、扱う側にも十分な知識が必要である。また、表現力の高いモデルを用いるほど過学習の危険があるため、正則化項を用いたり、パッチで学習したりする際は、学習誤差と訓練誤差の推移を確認したりすることが重要である。また、ある程度モデルが決まれば、交差検証等も有効である。

・ロジスティック回帰モデル

　回帰問題と異なるのは、0か1に分類するところだ。線形回帰問題の操作に加えて、シグモイド関数によって確率として評価し、その確率が最も訓練データと近い重みを導出することでモデルを作る。これは、あるデータの分布をベルヌーイ分布等のモデルで仮定することにより尤度関数を求め、その関数の最大を取る重みを取ることで計算され、対数を取り正負を逆転することにより、最小二乗法を用いることができる。データの数が多い場合、全てのデータで重みを更新せず一部のデータだけで更新することを繰り返すことで、徐々に最小に近づける方法もある。

　性能の評価には様々な指標があり、取り方によって数値が大きく変わる場合もあるため、何を評価しているのかをしっかりと確認する必要がある。

・主成分分析

　特徴量を空間上にマッピングしたときに、広がりが大きい方向（分散が大きいベクトル）を探し出し、射影することによって、データがもっている特徴を要約する。学習する特徴量が多い時に、次元を圧縮することができる。

・アルゴリズム

　k 近傍法はあらかじめラベルがわかっているデータを用意して、分類したいデータに対して空間的に近い数点のデータの多数決によって分類する手法である。

　k-meansは、教師がなくても分類でき、他のでーたとの空間的な距離が近いもので固まりを作ってラベリングする。モデルの評価が難しいが、一般的には同じくラベリングされたものどうしの密集度と他にラベリングされたものとの乖離度をあわせた指標を用いて評価される場合が多いようだ。

・サポートベクターマシン

　サポートベクターを探し出すために、分離平面からの距離が最小であるデータを探索する。一方で、そのデータに対して、その距離が最大となる分離平面を探さなくてはいけない。これを解くためにラグランジュ未定乗数法を用いる。サポートベクターを探し出すまでに勾配降下法で係数を決定するので、学習率や工ポック数を適切に設定する必要があった。現実には誤差無く完全に線形で分離できる問題は少なく、ある程度の誤差を許容するソフトマージンだったり、カーネルトリックを用いて線形分離できるように工夫する場合もある。

線形回帰モデル　実装演習

In [ ]:
```
from sklearn.datasets import load_boston
from sklearn.linear_model import LinearRegression
from sklearn.metrics import mean_squared_error
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
```

In [ ]:
```
boston = load_boston()

boston_df = pd.DataFrame(boston.data, columns = boston.feature_names)
boston_df['MEDV'] = boston.target
```

In [ ]:
```
boston_df.columns.values
```

Out[ ]:
```
array(['CRIM', 'ZN', 'INDUS', 'CHAS', 'NOX', 'RM', 'AGE', 'DIS', 'RAD',
       'TAX', 'PTRATIO', 'B', 'LSTAT', 'MEDV'], dtype=object)
```

In [21]:
```
#相関関係の模索
plt.figure(figsize=(20, 20))
cm = np.corrcoef(boston_df.values.T)
sns.set(font_scale=1)
hm = sns.heatmap(cm,
                 cbar=True,
                 annot=True,
                 square=True,
                 fmt='.2f',
                 annot_kws={'size': 25},
                 yticklabels=boston_df.columns.values,
                 xticklabels=boston_df.columns.values)

plt.tight_layout()
plt.show()
```

In [22]:

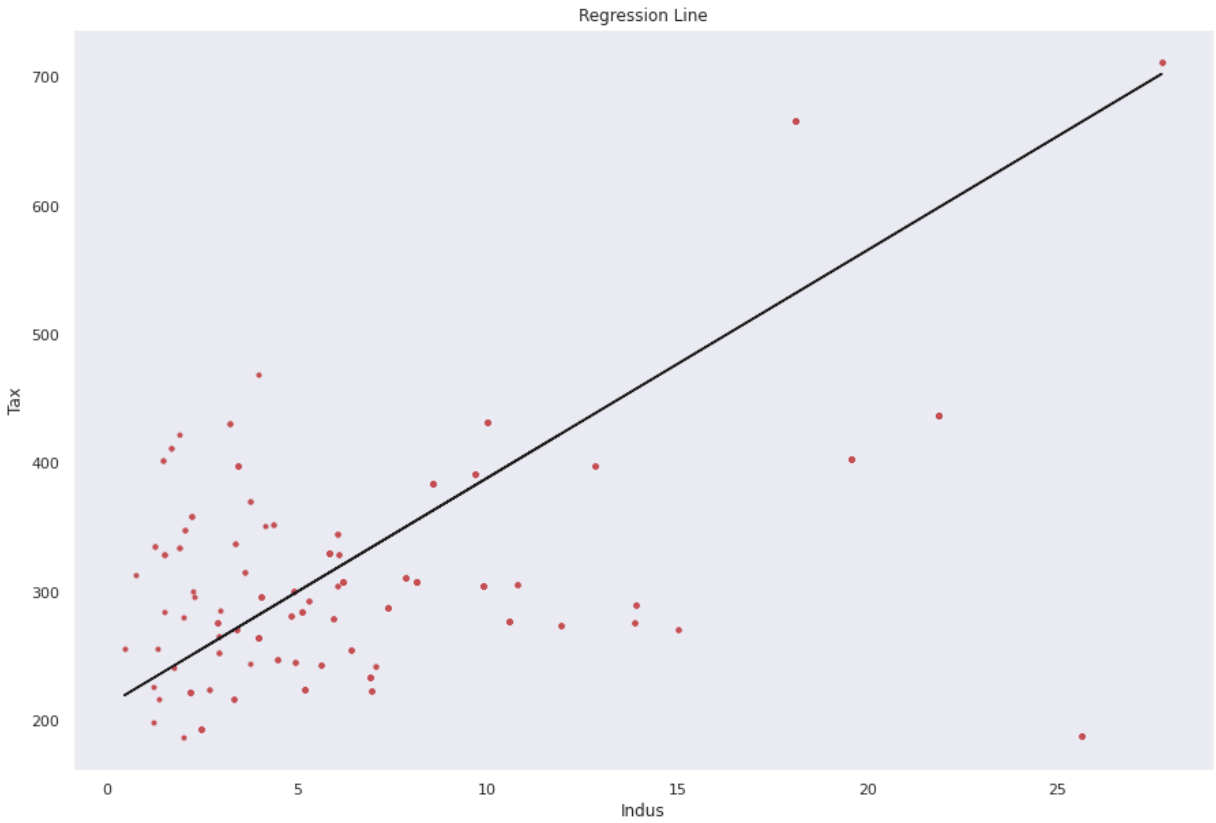
```
#taxとindusの線形回帰モデルの作成
lr = LinearRegression()

X = boston_df[['INDUS']].values
Y = boston_df['TAX'].values

lr.fit(X, Y)
```

Out[22]: LinearRegression(copy_X=True, fit_intercept=True, n_jobs=None, normalize=False)

In [23]:

```
#plot
Y_pred = lr.predict(X)
fig = plt.figure(figsize=(15,10))
plt.plot(X, Y, 'r.')
plt.plot(X, Y_pred, 'k-')

plt.title('Regression Line')
plt.xlabel('Indus')
plt.ylabel('Tax')
plt.grid()
```

```
plt.show()
```



In [24]:

```
#評価
print('MSE: ', mean_squared_error(Y, Y_pred))
```

```
MSE:  13621.64866409399
```

In [25]:

```
#課題
X1 = boston_df[['CRIM','RM']].values
Y1 = boston_df['MEDV'].values

lr.fit(X1, Y1)
print(lr.predict([[0.3,4]]))
```

```
[4.24007956]
```

ボストンデータセットから相関の強いデータを探し、線形回帰を試みた。
MSEで評価を行ったが、単体の値だけでは意味が薄く、パラメトリックスタディなどをして比較する際に求めるべきだったと感じた。
また、相関が強いからと言って必ずしも線形回帰が適切であるとは限らず、外れ値に関しての工学的な判断も必要であると感じた。

非線形回帰モデル　実装演習

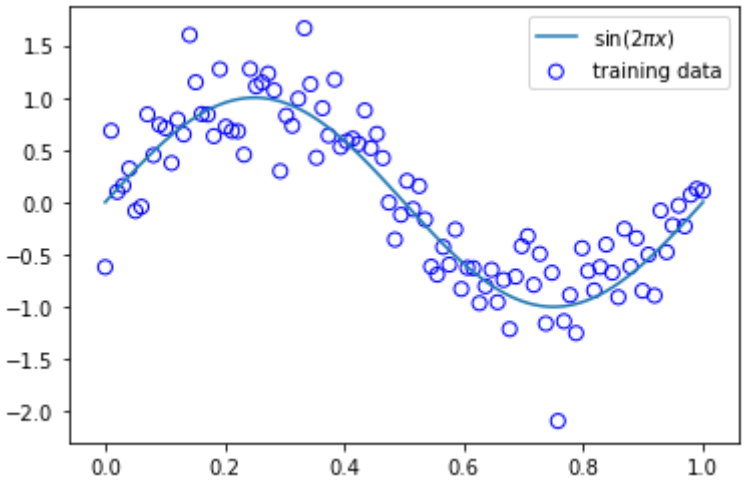
In [2]:
```
import numpy as np
import matplotlib.pyplot as plt
```

In [3]:
```
def sin_func(x):
    return np.sin(2 * np.pi * x)

def add_noise(y_true, var):
    return y_true + np.random.normal(scale=var, size=y_true.shape)

def plt_result(xs, ys_true, ys):
    plt.scatter(xs, ys,facecolor="none", edgecolor="b", s=50,  label="training data")
    plt.plot(xs, ys_true, label="$¥sin(2¥pi x)$")
    plt.legend()

def polynomial_features(xs, degree=3):
    """多項式特徴ベクトルに変換
         X = [[1, x1, x1^2, x1^3],
                 [1, x2, x2^2, x2^3],
                   ...
                 [1, xn, xn^2, xn^3]]"""
    X = np.ones((len(xs), degree+1))
    X_t = X.T #(100, 4)
    for i in range(1, degree+1):
        X_t[i] = X_t[i-1] * xs
    return X_t.T
```

In [4]:
```
#データの作成
n_sample = 100
var = .300
xs = np.linspace(0, 1, n_sample)
ys_true = sin_func(xs)
ys = add_noise(ys_true, var)

print("xs: {}".format(xs.shape))
print("ys_true: {}".format(ys_true.shape))
print("ys: {}".format(ys.shape))
```

```
xs: (100,)
ys_true: (100,)
ys: (100,)
```

In [5]:
```
#結果の描画
plt_result(xs, ys_true, ys)
```

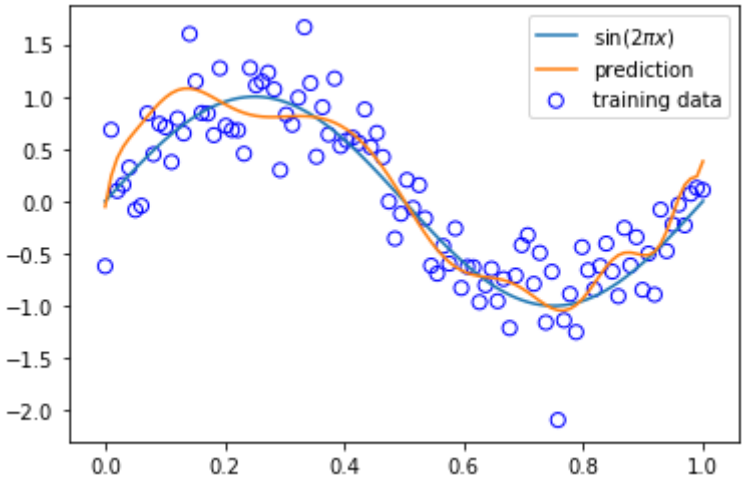


In [16]:

```
#学習
Phi = polynomial_features(xs,31)
Phi_inv = np.dot(np.linalg.inv(np.dot(Phi.T, Phi)), Phi.T)
w = np.dot(Phi_inv, ys)

#予測
ys_pred = np.dot(Phi, w)
```

In [17]:

```
plt.scatter(xs, ys, facecolor="none", edgecolor="b", s=50, label="training data")
plt.plot(xs, ys_true, label="$¥sin(2¥pi x)$")
plt.plot(xs, ys_pred, label="prediction")
# for i in range(0, 4):
#     plt.plot(xs, Phi[:, i], label="basis")
plt.legend()
plt.show()
```



サイン関数をもとにしたデータの非線形回帰を行った。
今回のデータの範囲的には大体３次程度の表現力があれば表現できそうと推察されるが、
実際にも3~5程度の次数で良好なけっかとなった。
値を極端に大きくすると過学習の傾向がみられることも確認できた。

ロジスティック回帰モデル

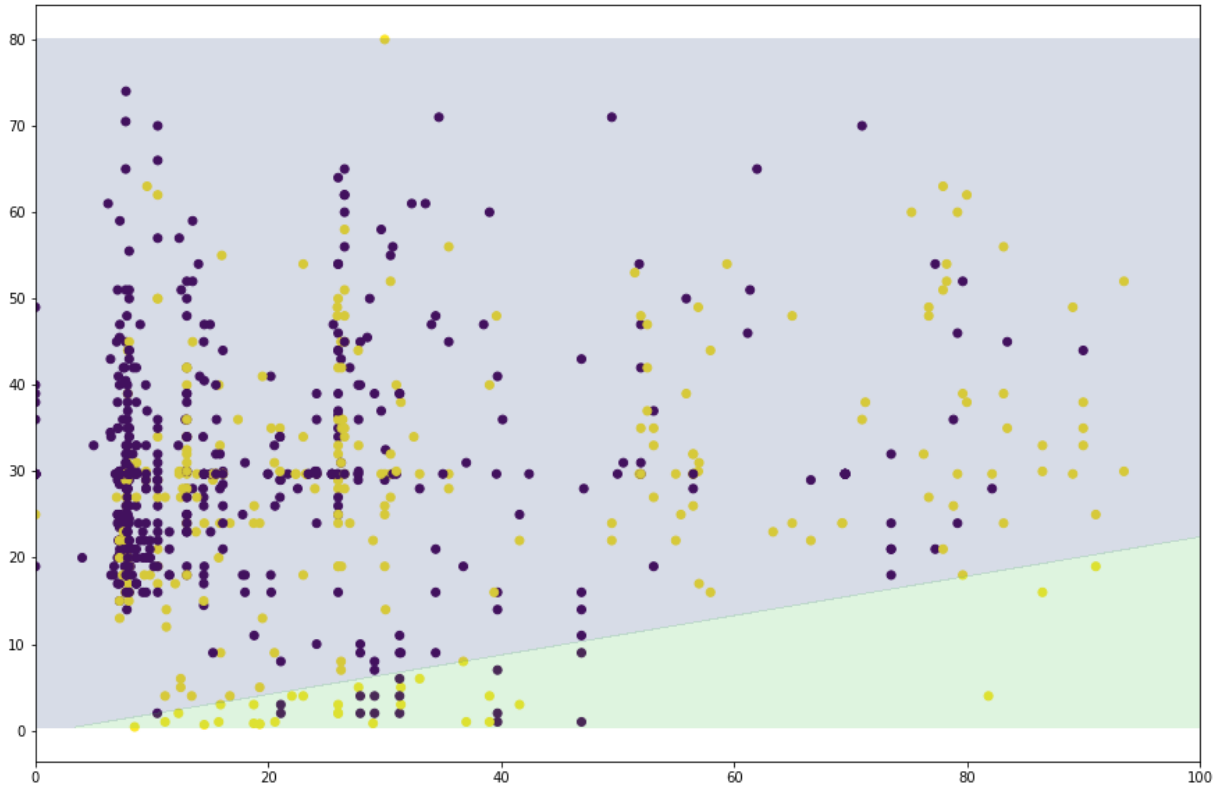
In [1]:
```
import pandas as pd
import matplotlib.pyplot as plt
import numpy as np
from sklearn.metrics import confusion_matrix
import seaborn as sns
from seaborn import load_dataset
```

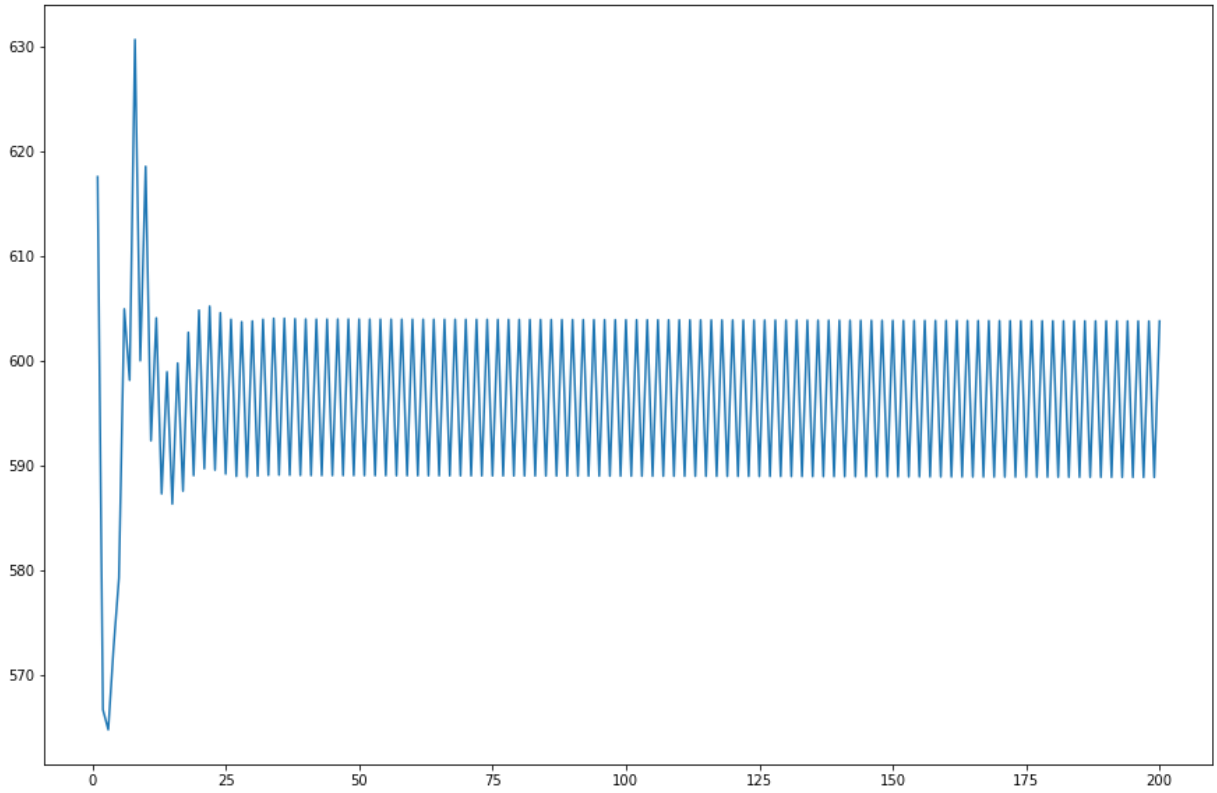
In [2]:
```
data = load_dataset("titanic")
# data30 = data[data['age'] < 35][data['age'] > 25][data['sex'] == 'male']
# cor = data30.corr()
ave_age = data['age'].mean()
data['age'].fillna(ave_age,inplace=True)
X = data[['fare','age']].values
y = data['survived'].values
```

In [9]:
```
#正則化　コスト関数の実装
def sigmoid(x):
    return 1 / (1 + np.exp(-x))

def sgd(X_train, y_train, max_iter, eta,c):
    w = np.zeros(X_train.shape[1])
    cost = []
    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))*c
#         print(grad+ w_prev**2/2)
        w -= eta * (grad + w_prev**2/2)
        cost.append(-np.dot(y_train,np.log(sigma))-np.dot((1-y_train),np.log(1-sigma)
        if np.allclose(w, w_prev):
            return w
    return w,cost
```

In [10]:
```
max_iter=200
Xb = np.hstack((np.ones((len(X[:,0]), 1)), X))

w ,cost= sgd(Xb, y, max_iter, 0.00001,0.5)

xx0, xx1 = np.meshgrid(np.linspace(min(X[:,0]), max(X[:,0]), 100), np.linspace(min(X[
xx = np.array([xx0, xx1]).reshape(2, -1).T
xxb = np.hstack((np.ones((len(xx[:,0]), 1)), xx))
proba = sigmoid(np.dot(xxb,w))
y_pred = (proba > 0.5).astype(np.int)
fig = plt.figure(figsize=(15,10))
plt.scatter(X[:, 0], X[:, 1], c=y)
plt.contourf(xx0, xx1, proba.reshape(100, 100), alpha=0.2, levels=np.linspace(0, 1, 3
plt.xlim([0,100])
fig = plt.figure(figsize=(15,10))
plt.plot(np.linspace(1,max_iter,max_iter),cost)
```

Out[10]:
```
[<matplotlib.lines.Line2D at 0x151766bdfc8>]
```





In [11]:

```
# 課題　30歳の生存率
from sklearn.linear_model import LogisticRegression
sex = pd.get_dummies(data['sex'])
mAge = np.hstack((data[['age']].values,sex[['male']]))
model=LogisticRegression(fit_intercept=True)

model.fit(mAge, y)
print( 'survived = ',model.predict_proba([[30,1]])[0,1])

score = model.score(mAge, y)
print(score, 1-y.mean())
cm=confusion_matrix(y, model.predict(X))
```

```
survived =  0.1933594115895271
0.7867564534231201 0.6161616161616161
C:¥Users¥zawaz¥Anaconda3¥lib¥site-packages¥sklearn¥linear_model¥logistic.py:432: FutureWarning: Default solver will be changed to 'lbfgs' in 0.22. Specify a solver to silence this warning.
  FutureWarning)
```

タイタニックのデータセットからロジスティック回帰によって生存かどうかの推論を行った。
その際、学習に用いるコスト関数に正則化の項を追加した。
目論見通り重みが過剰に大きな値になることはおさえられたが、収束に関してはあまりうまくいかなかった。
課題の演習では30歳男性の生存率の推定を行った。

主成分分析

In [1]:
```
import warnings
import pandas as pd
import matplotlib.pyplot as plt
import numpy as np
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler
from sklearn.linear_model import LogisticRegressionCV
from sklearn.metrics import confusion_matrix
from sklearn.decomposition import PCA
from sklearn.datasets import load_breast_cancer
import seaborn
warnings.simplefilter('ignore')
```

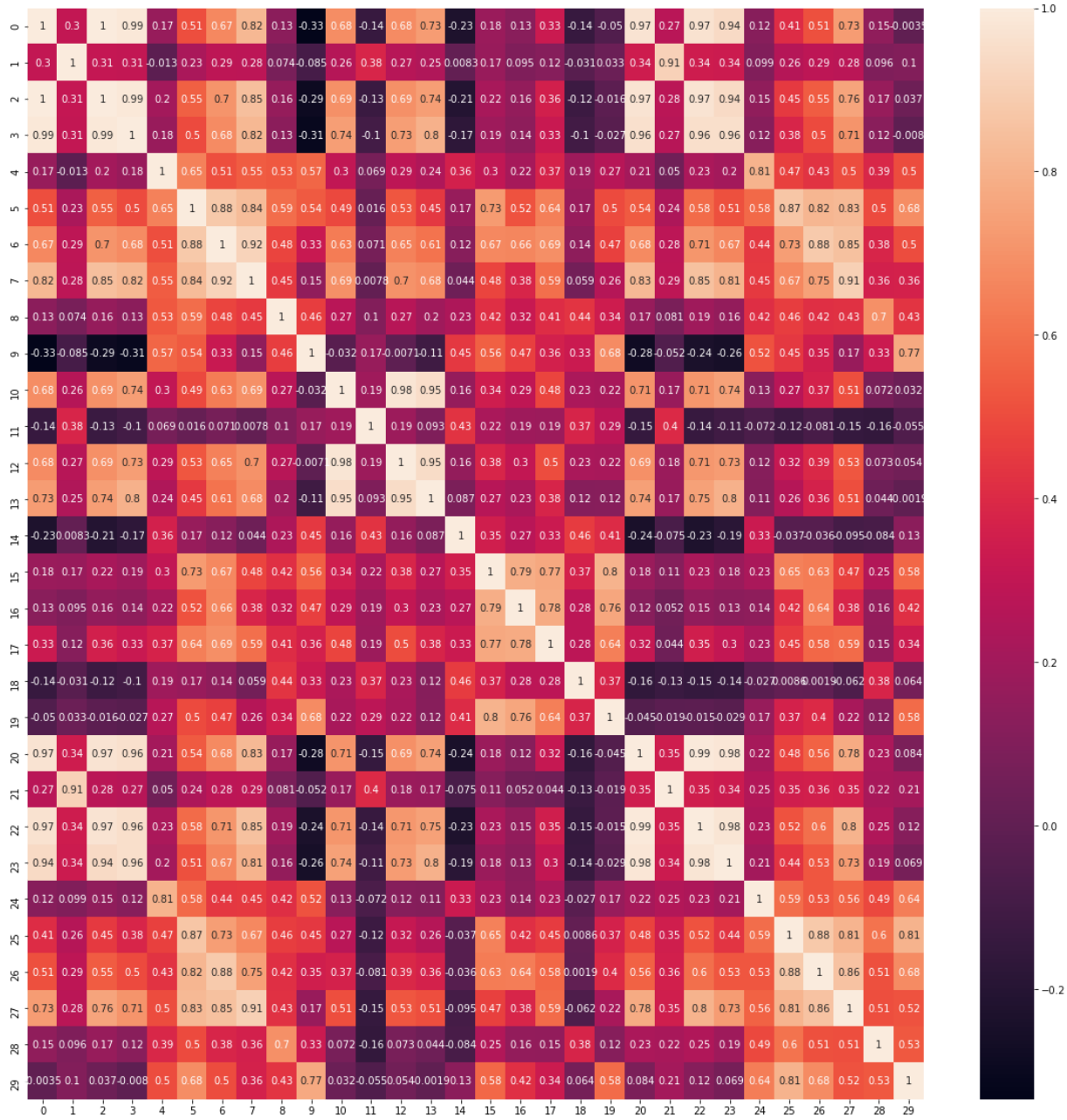
In [2]:
```
data_breast_cancer = load_breast_cancer()
X = pd.DataFrame(data_breast_cancer["data"],columns=data_breast_cancer["feature_names
y = pd.DataFrame(data_breast_cancer["target"],columns=["target"])

X_train, X_test, y_train, y_test = train_test_split(X, y, random_state=0)

sc = StandardScaler()
X_train_std = sc.fit_transform(X_train)
X_test_std = sc.transform(X_test)
```

In [3]:
```
lr = LogisticRegressionCV(cv=10, random_state=0)
lr.fit(X_train_std,y_train)
print('train score:',lr.score(X_train_std,y_train))
print('test score:',lr.score(X_test_std,y_test))
```

```
train score: 0.9882629107981221
test score: 0.972027972027972
```

In [4]:
```
plt.figure(figsize=(20, 20))
seaborn.heatmap(pd.DataFrame(X_train_std).corr(), annot=True)
```

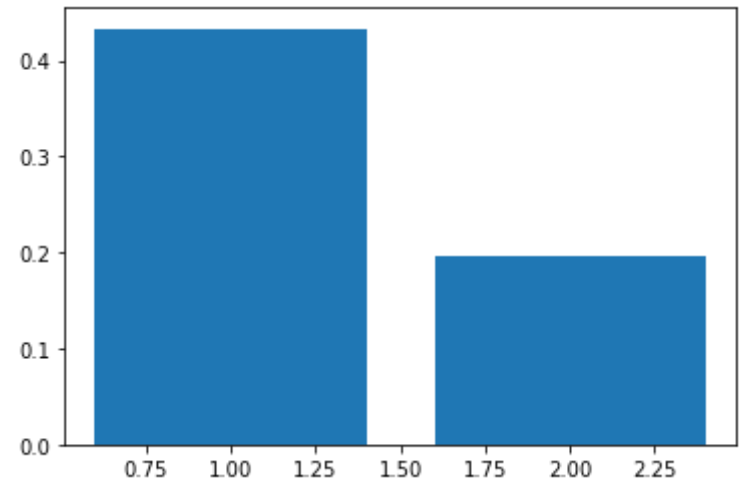
Out[4]:
```
<matplotlib.axes._subplots.AxesSubplot at 0x26207f29908>
```



In [9]:

```
pca = PCA(n_components=2)
pca.fit(X_train_std)
plt.bar([n for n in range(1, len(pca.explained_variance_ratio_)+1)], pca.explained_va
```

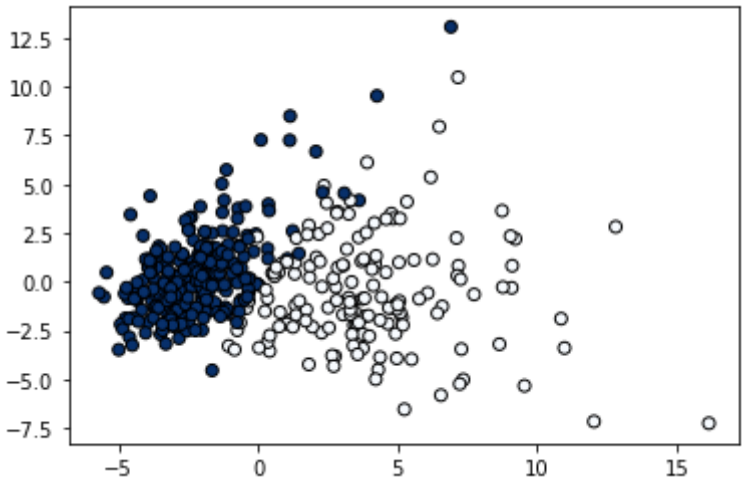
Out[9]: <BarContainer object of 2 artists>



In [10]:
```
X_train_pca = pca.fit_transform(X_train_std)
X_test_pca = pca.fit_transform(X_test_std)

lr = LogisticRegressionCV(cv=10, random_state=0)
lr.fit(X_train_pca,y_train)
print('train score:',lr.score(X_train_pca,y_train))
print('test score:',lr.score(X_test_pca,y_test))
plt.scatter(X_train_pca[:, 0], X_train_pca[:, 1],c = y_train.values.flatten(),edgecol
```

```
train score: 0.9647887323943662
test score: 0.916083916083916
```

Out[10]:
```
<matplotlib.collections.PathCollection at 0x26208d13408>
```



乳がんデータを主成分分析を行って次元削減したデータを、ロジスティック回帰で分類した。
2つの主成分だけで、データの分散の6割近くを担っていたため、すべてのデータを用いて行った場合とほぼ遜色なく学習ができた。
3つに増やした場合も行ったが、それほどスコアに変化はなかった。
欠点としては、より学習の中身がブラックボックス化してしまうため、説明性がひくくなる。

アルゴリズム

In [7]:
```
import numpy as np
import matplotlib.pyplot as plt
from matplotlib import cm
from sklearn.metrics import silhouette_samples
from sklearn.datasets import load_wine
from sklearn.cluster import KMeans
from sklearn.model_selection import train_test_split
import pandas as pd
import seaborn as sns
```

In [2]:
```
data_wine = load_wine()
X = data_wine.data
y = data_wine.target
```

In [16]:
```
wine_df=pd.DataFrame(X,columns = data_wine.feature_names )
wine_df['target']=y
print(wine_df)
sns.pairplot(wine_df,hue = 'target')
```

```
     alcohol  malic_acid   ash  alcalinity_of_ash  magnesium  total_phenols  ¥
0      14.23        1.71  2.43               15.6      127.0           2.80
1      13.20        1.78  2.14               11.2      100.0           2.65
2      13.16        2.36  2.67               18.6      101.0           2.80
3      14.37        1.95  2.50               16.8      113.0           3.85
4      13.24        2.59  2.87               21.0      118.0           2.80
..       ...         ...   ...                ...        ...            ...
173    13.71        5.65  2.45               20.5       95.0           1.68
174    13.40        3.91  2.48               23.0      102.0           1.80
175    13.27        4.28  2.26               20.0      120.0           1.59
176    13.17        2.59  2.37               20.0      120.0           1.65
177    14.13        4.10  2.74               24.5       96.0           2.05

     flavanoids  nonflavanoid_phenols  proanthocyanins  color_intensity   hue  ¥
0          3.06                  0.28             2.29             5.64  1.04
1          2.76                  0.26             1.28             4.38  1.05
2          3.24                  0.30             2.81             5.68  1.03
3          3.49                  0.24             2.18             7.80  0.86
4          2.69                  0.39             1.82             4.32  1.04
..          ...                   ...              ...              ...   ...
173        0.61                  0.52             1.06             7.70  0.64
174        0.75                  0.43             1.41             7.30  0.70
175        0.69                  0.43             1.35            10.20  0.59
176        0.68                  0.53             1.46             9.30  0.60
177        0.76                  0.56             1.35             9.20  0.61

     od280/od315_of_diluted_wines  proline  target
0                            3.92   1065.0       0
1                            3.40   1050.0       0
2                            3.17   1185.0       0
3                            3.45   1480.0       0
4                            2.93    735.0       0
..                            ...      ...     ...
173                          1.74    740.0       2
174                          1.56    750.0       2
175                          1.56    835.0       2
176                          1.62    840.0       2
177                          1.60    560.0       2

[178 rows x 14 columns]
```

Out[16]:
```
<seaborn.axisgrid.PairGrid at 0x132c02eb348>
```

In [3]:

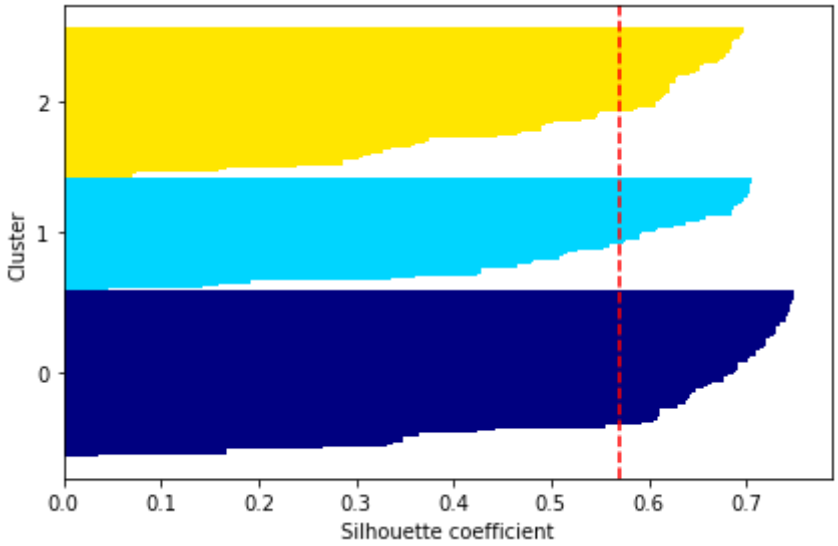
```
#kmeans
km = KMeans(n_clusters=3)
y_km = km.fit_predict(X)
df = pd.DataFrame({'y_km':y_km})
df['y_wine'] = y

print( pd.crosstab(df['y_km'],df['y_wine']) )
```

```
y_wine   0   1   2
y_km
0        0  50  19
1       46   1   0
2       13  20  29
```

In [17]:

```
#kmeans シルエット図
cluster_labels = np.unique(y_km)
n_clusters = cluster_labels.shape[0]
silhouette_vals = silhouette_samples(X, y_km, metric='euclidean')
y_ax_lower, y_ax_upper = 0, 0
yticks = []
for i, c in enumerate(cluster_labels):
    c_silhouette_vals = silhouette_vals[y_km == c]
    c_silhouette_vals.sort()
    y_ax_upper += len(c_silhouette_vals)
    color = cm.jet(float(i) / n_clusters)
    plt.barh(range(y_ax_lower, y_ax_upper), c_silhouette_vals, height=1.0,
```

```
                edgecolor='none', color=color)

    yticks.append((y_ax_lower + y_ax_upper) / 2.)
    y_ax_lower += len(c_silhouette_vals)

silhouette_avg = np.mean(silhouette_vals)
plt.axvline(silhouette_avg, color="red", linestyle="--")

plt.yticks(yticks, cluster_labels )
plt.ylabel('Cluster')
plt.xlabel('Silhouette coefficient')

plt.tight_layout()
plt.show()
```



In [5]:

```
#K近傍法
from sklearn.neighbors import KNeighborsClassifier
X_train, X_test, y_train, y_test = train_test_split(X, y, random_state=1)
knc = KNeighborsClassifier(n_neighbors=10).fit(X_train, y_train)
y_test_knc = knc.predict(X_test)
df2 = pd.DataFrame({'y_knc':y_test_knc})
df2['y_wine'] = y_test
print( pd.crosstab(df2['y_knc'],df2['y_wine']) )
```

```
y_wine   0   1  2
y_knc
0       14   1  0
1        0  12  6
2        4   4  4
```

Kmeansを用いたデータのクラスタリングと評価を行った。
データの特徴マップをみると、閾値で切るタイプの方がきれいに分類できそうな雰囲気もあるが、実際にアルゴリズムにかけるとそれなりに分類できたが、元のデータのクラス２（分類後のクラス２）はあまり良好ではない印象がある。
そういう意味で言うと、教師がある場合は素直に使った方がいいといえるかもしれない。
シルエット図を見ると、0は比較的シルエット係数が平均値を超えている。しかし、1,2は半数程度シルエット係数が低く、モデルとしては良好とはいえないかもしれない。
K近傍法での分類も行った。利用する近傍データの数によって分類のされ方が変わることも確認できた。
こちらの結果からもわかるように、元のデータのクラス２（分類後のクラス２）は他のクラスと空間的に近いことがわかる。

実際に教師データがないとしたら、既知の知見から特徴量を絞るといった工夫が必要かもしれない。

サポートベクターマシン　実装演習

In [11]:

```
import numpy as np
import matplotlib.pyplot as plt
import pandas as pd
from sklearn.datasets import load_iris
import seaborn as sns
```

In [12]:

```
iris = load_iris()

X = pd.DataFrame(iris.data, columns=iris.feature_names)
y = iris.target
X["target"] = iris.target
# sns.pairplot(X,hue = "target")

X_train = X.iloc[:,[1,3]].values

t = np.where(y == 0, 1.0, -1.0)
print(t)

n_samples = len(X_train)
```

```
[ 1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.
  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.
  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1. -1. -1. -1. -1.
 -1. -1. -1. -1. -1. -1. -1. -1. -1. -1. -1. -1. -1. -1. -1. -1. -1. -1.
 -1. -1. -1. -1. -1. -1. -1. -1. -1. -1. -1. -1. -1. -1. -1. -1. -1. -1.
 -1. -1. -1. -1. -1. -1. -1. -1. -1. -1. -1. -1. -1. -1. -1. -1. -1. -1.
 -1. -1. -1. -1. -1. -1. -1. -1. -1. -1. -1. -1. -1. -1. -1. -1. -1. -1.
 -1. -1. -1. -1. -1. -1. -1. -1. -1. -1. -1. -1. -1. -1. -1. -1. -1. -1.
 -1. -1. -1. -1. -1. -1.]
```

In [13]:

```
K = X_train.dot(X_train.T)
print(K)

eta1 = 0.001
eta2 = 0.0001
n_iter = 100000

H = np.outer(t, t) * K

a = np.ones(n_samples)
print(a)
```

```
[[12.29 10.54 11.24 ... 10.9  12.36 10.86]
 [10.54  9.04  9.64 ...  9.4  10.66  9.36]
 [11.24  9.64 10.28 ... 10.   11.34  9.96]
 ...
 [10.9   9.4  10.   ... 13.   14.8  12.6 ]
 [12.36 10.66 11.34 ... 14.8  16.85 14.34]
 [10.86  9.36  9.96 ... 12.6  14.34 12.24]]
[1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1.
 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1.
 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1.
 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1.
 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1.
 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1.
 1. 1. 1. 1. 1. 1.]
```

In [14]:

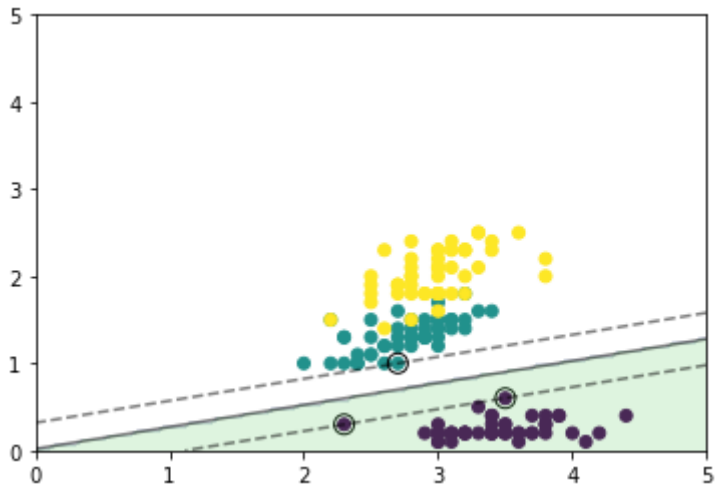
```
for _ in range(n_iter):
    grad = 1 - H.dot(a)
    a += eta1 * grad
    a -= eta2 * a.dot(t) * t
    a = np.where(a > 0, a, 0)
```

```
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(0, 5, 100), np.linspace(0, 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)
```

In [16]:

```
# 訓練データを可視化
plt.scatter(X_train[:, 0], X_train[:, 1], c=y)
# 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,
#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=['--', '-', '--'])
```

Out[16]: <matplotlib.contour.QuadContourSet at 0x7faa3508b750>



irisデータセットをSVMで分離することを試みた。
学習率とイテレーション回数を適切に設定しないと、サポートベクターが適切に決定されず結果的には発散してしまった。
今回は、学習率を低くしてかなりの回数学習させることにより収束させたが、それでも、理論通りのサポートベクター（境界から最も小さい要素）を取得できているわけではない。
このデータよりも重いデータを学習させるような場合には、学習率の更新のさせ方に工夫が必要だろう。

参考図書「達人データサイエンティストによる理論と実践　Python 機械学習プログラミング」10章より

データの読み込み

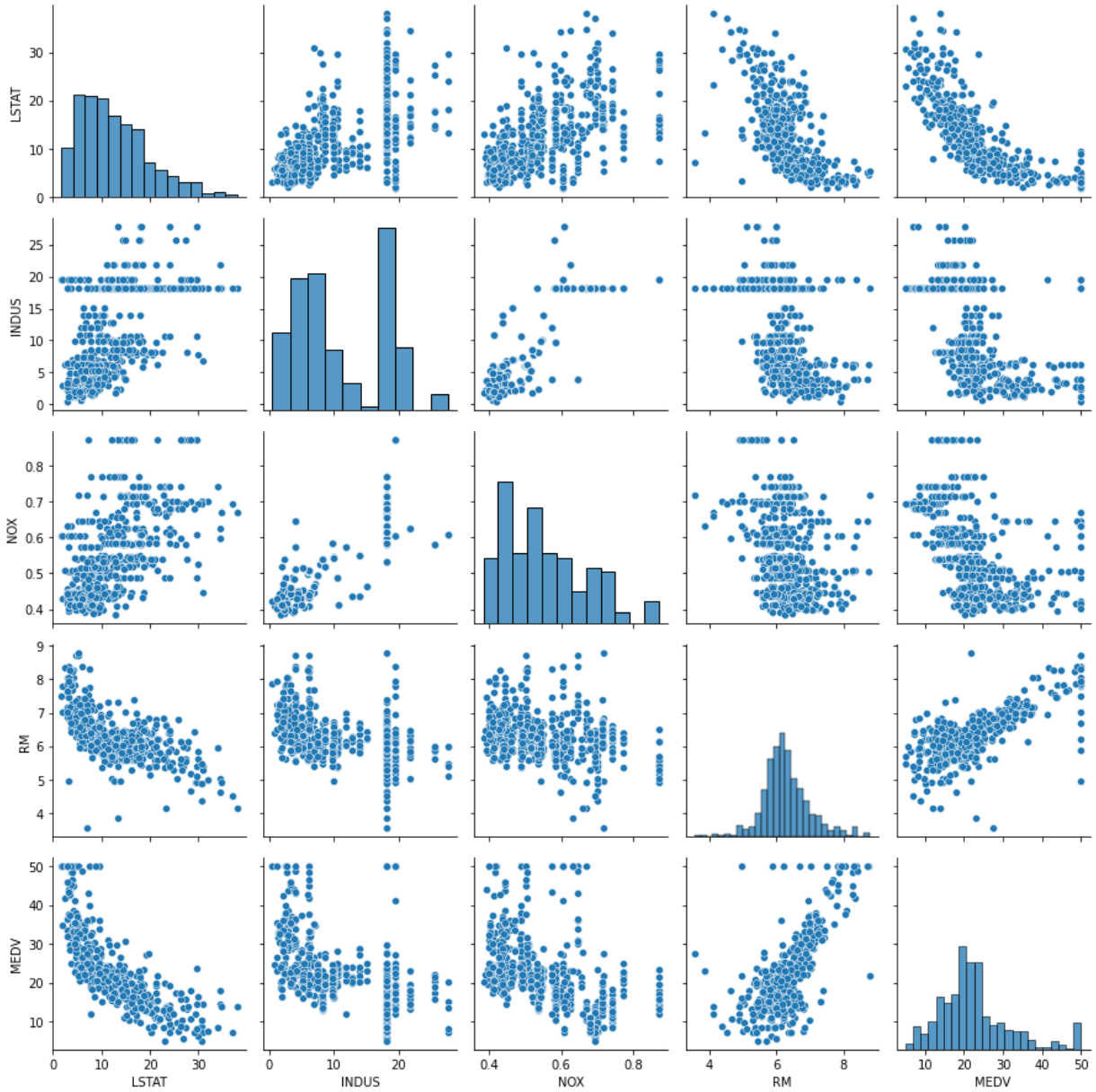
In [1]:
```
from sklearn.datasets import load_boston
from sklearn.linear_model import LinearRegression
from sklearn.metrics import mean_squared_error
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns

boston = load_boston()

boston_df = pd.DataFrame(boston.data, columns = boston.feature_names)
boston_df['MEDV'] = boston.target

boston_df.head()
```

Out[1]:

	CRIM	ZN	INDUS	CHAS	NOX	RM	AGE	DIS	RAD	TAX	PTRATIO	B	LSTAT	M
0	0.00632	18.0	2.31	0.0	0.538	6.575	65.2	4.0900	1.0	296.0	15.3	396.90	4.98	
1	0.02731	0.0	7.07	0.0	0.469	6.421	78.9	4.9671	2.0	242.0	17.8	396.90	9.14	
2	0.02729	0.0	7.07	0.0	0.469	7.185	61.1	4.9671	2.0	242.0	17.8	392.83	4.03	
3	0.03237	0.0	2.18	0.0	0.458	6.998	45.8	6.0622	3.0	222.0	18.7	394.63	2.94	
4	0.06905	0.0	2.18	0.0	0.458	7.147	54.2	6.0622	3.0	222.0	18.7	396.90	5.33	

In [2]:
```
cols = ['LSTAT', 'INDUS', 'NOX', 'RM', 'MEDV']

sns.pairplot(boston_df[cols], size=2.5)
plt.tight_layout()
plt.show()
```

```
/usr/local/lib/python3.7/dist-packages/seaborn/axisgrid.py:1969: UserWarning: The `size` parameter has been renamed to `height`; please update your code.
  warnings.warn(msg, UserWarning)
```



ヒートマップ作製

In [5]:

```
cm = np.corrcoef(boston_df[cols].values.T)
sns.set(font_scale=1)
hm = sns.heatmap(cm,
                 cbar=True,
                 annot=True,
                 square=True,
                 fmt='.2f',
                 annot_kws={'size': 15},
                 yticklabels=cols,
                 xticklabels=cols)

plt.tight_layout()
plt.show()
```

回帰モデルの実装

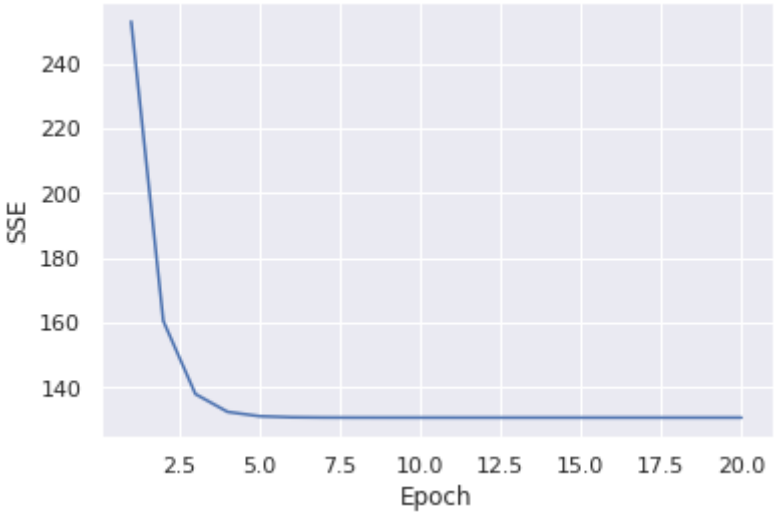
ADALINEの実装（勾配降下法）

- コスト関数：誤差平方和
- 前処理：正規化

In [53]:

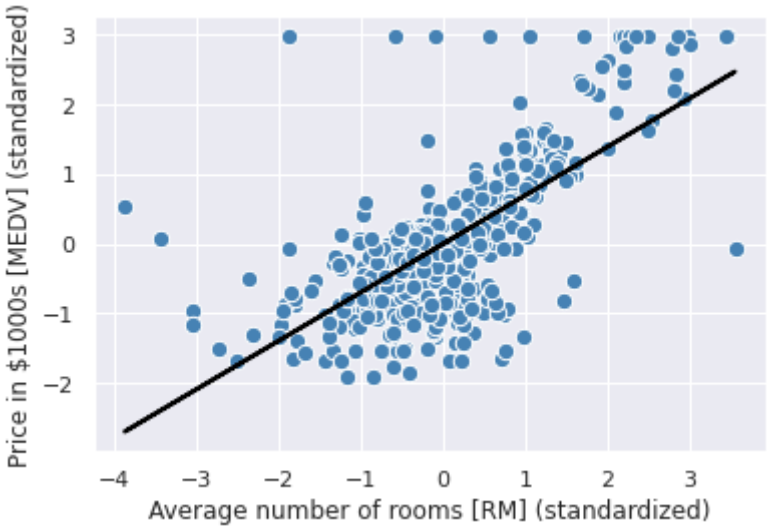
```
class LinearRegressionGD(object):

    def __init__(self, eta=0.001, n_iter=20):
        self.eta = eta
        self.n_iter = n_iter

    def fit(self, X, y):
        self.w_ = np.zeros(1 + X.shape[1])
        self.cost_ = []

        for i in range(self.n_iter):
            output = self.net_input(X)
            errors = (y - output)
            self.w_[1:] += self.eta * X.T.dot(errors)
            self.w_[0] += self.eta * errors.sum()
            # print(self.w_)
            cost = (errors**2).sum() / 2.0
            self.cost_.append(cost)
        return self

    def net_input(self, X):
        return np.dot(X, self.w_[1:]) + self.w_[0]

    def predict(self, X):
        return self.net_input(X)

X = boston_df[['RM']].values
y = boston_df[['MEDV']].values

from sklearn.preprocessing import StandardScaler
sc_x = StandardScaler()
sc_y = StandardScaler()
X_std = sc_x.fit_transform(X)
y_std = sc_y.fit_transform(y).flatten()

lr = LinearRegressionGD()
lr.fit(X_std, y_std)
```

```
plt.plot(range(1, lr.n_iter+1), lr.cost_)
plt.ylabel('SSE')
plt.xlabel('Epoch')
plt.show()
```



In [52]:

```
def lin_regplot(X, y, model):
    plt.scatter(X, y, c='steelblue', edgecolor='white', s=70)
    plt.plot(X, model.predict(X), color='black', lw=2)
    return

lin_regplot(X_std, y_std, lr)
plt.xlabel('Average number of rooms [RM] (standardized)')
plt.ylabel('Price in $1000s [MEDV] (standardized)')
plt.show()
```



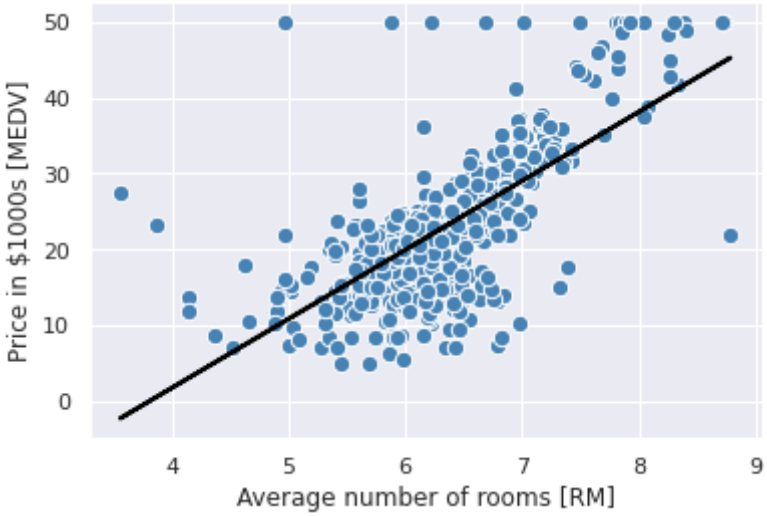
scikit-learnを使った回帰モデル

In [54]:

```
slr = LinearRegression()
slr.fit(X, y)
y_pred = slr.predict(X)
print('Slope: %.3f' % slr.coef_[0])
print('Intercept: %.3f' % slr.intercept_)

lin_regplot(X, y, slr)
```

```
plt.xlabel('Average number of rooms [RM]')
plt.ylabel('Price in $1000s [MEDV]')

plt.show()
```

```
Slope: 9.102
Intercept: -34.671
```



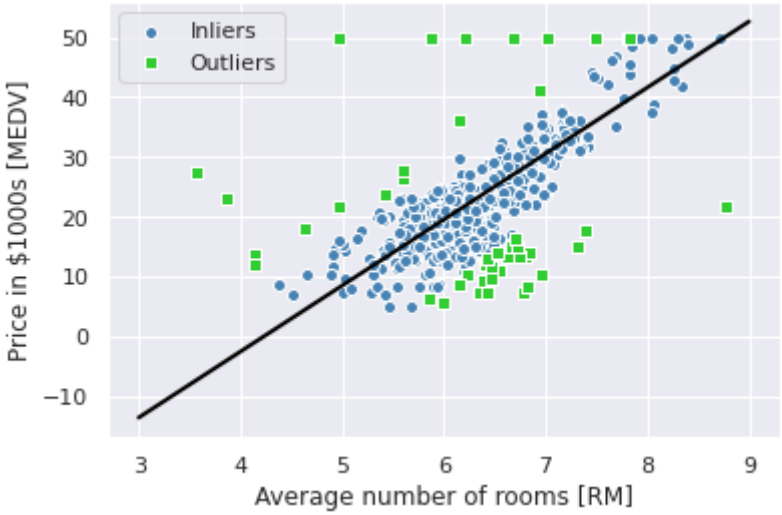
RANSACを使ったロバスト回帰

- ランダムな数のサンプルからモデルを学習させる。
- 許容範囲の誤差にある他のデータを選択し、再び学習させることを繰り返す。
- 外れ値を避けて学習が可能。

In [62]:

```
from sklearn.linear_model import RANSACRegressor
ransac = RANSACRegressor(LinearRegression(),
                         max_trials=100,
                         min_samples=50,
                         loss='absolute_loss',
                         residual_threshold=10.0,
                         random_state=0)


ransac.fit(X, y)
print('Slope: %.3f' % ransac.estimator_.coef_[0])
print('Intercept: %.3f' % ransac.estimator_.intercept_)

inlier_mask = ransac.inlier_mask_
outlier_mask = np.logical_not(inlier_mask)

line_X = np.arange(3, 10, 1)
line_y_ransac = ransac.predict(line_X[:, np.newaxis])
plt.scatter(X[inlier_mask], y[inlier_mask],
            c='steelblue', edgecolor='white',
            marker='o', label='Inliers')
plt.scatter(X[outlier_mask], y[outlier_mask],
            c='limegreen', edgecolor='white',
            marker='s', label='Outliers')
plt.plot(line_X, line_y_ransac, color='black', lw=2)
plt.xlabel('Average number of rooms [RM]')
plt.ylabel('Price in $1000s [MEDV]')
plt.legend(loc='upper left')
plt.show()
```

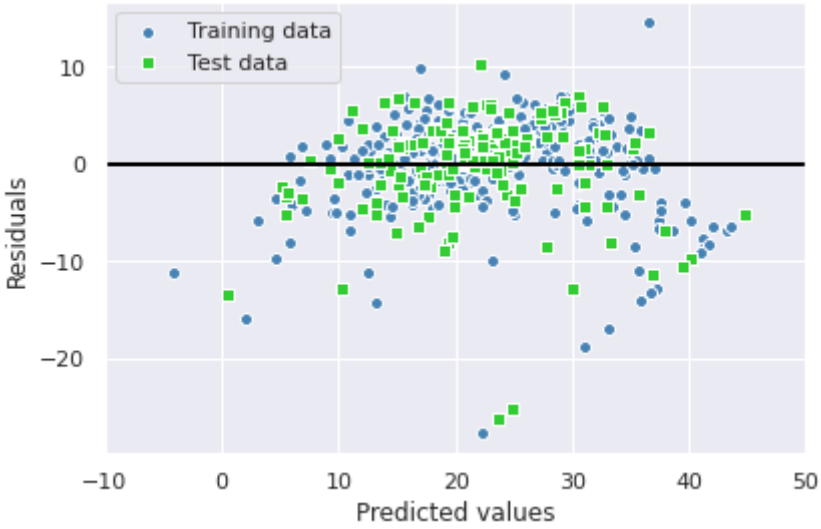
```
Slope: 11.026
Intercept: -46.621
```



重回帰モデルと性能評価

In [65]:

```
from sklearn.model_selection import train_test_split
X = boston_df.iloc[:, :-1].values
y = boston_df['MEDV'].values

X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.3, random_state=

slr = LinearRegression()

slr.fit(X_train, y_train)
y_train_pred = slr.predict(X_train)
y_test_pred = slr.predict(X_test)

plt.scatter(y_train_pred,  y_train_pred - y_train,
            c='steelblue', marker='o', edgecolor='white',
            label='Training data')
plt.scatter(y_test_pred,  y_test_pred - y_test,
            c='limegreen', marker='s', edgecolor='white',
            label='Test data')
plt.xlabel('Predicted values')
plt.ylabel('Residuals')
plt.legend(loc='upper left')
plt.hlines(y=0, xmin=-10, xmax=50, color='black', lw=2)
plt.xlim([-10, 50])
plt.tight_layout()

plt.show()

from sklearn.metrics import mean_squared_error
from sklearn.metrics import r2_score
print('MSE train: %.3f, test: %.3f' % (
        mean_squared_error(y_train, y_train_pred),
        mean_squared_error(y_test, y_test_pred)))
print('R^2 train: %.3f, test: %.3f' % (
        r2_score(y_train, y_train_pred),
        r2_score(y_test, y_test_pred)))
```




```
MSE train: 19.958, test: 27.196
R^2 train: 0.765, test: 0.673
```

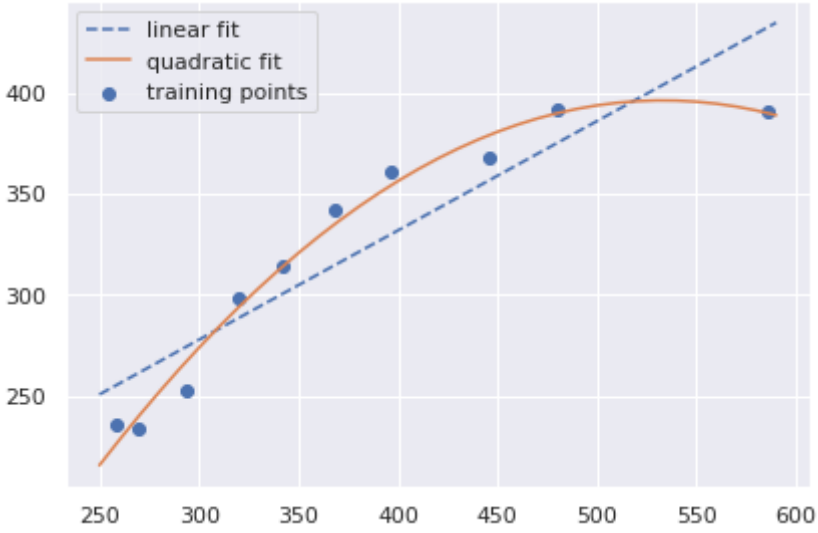
非線形回帰

In [68]:

```
from sklearn.preprocessing import PolynomialFeatures
X = np.array([258.0, 270.0, 294.0,
              320.0, 342.0, 368.0,
              396.0, 446.0, 480.0, 586.0])¥
             [:, np.newaxis]

y = np.array([236.4, 234.4, 252.8,
              298.6, 314.2, 342.2,
              360.8, 368.0, 391.2,
              390.8])

lr = LinearRegression()
pr = LinearRegression()
quadratic = PolynomialFeatures(degree=2)
X_quad = quadratic.fit_transform(X)
print(X_quad)

# fit linear features
lr.fit(X, y)
X_fit = np.arange(250, 600, 10)[:, np.newaxis]
y_lin_fit = lr.predict(X_fit)

# fit quadratic features
pr.fit(X_quad, y)
y_quad_fit = pr.predict(quadratic.fit_transform(X_fit))

# plot results
plt.scatter(X, y, label='training points')
plt.plot(X_fit, y_lin_fit, label='linear fit', linestyle='--')
plt.plot(X_fit, y_quad_fit, label='quadratic fit')
plt.legend(loc='upper left')

plt.tight_layout()
#plt.savefig('images/10_10.png', dpi=300)
plt.show()

y_lin_pred = lr.predict(X)
y_quad_pred = pr.predict(X_quad)

print('Training MSE linear: %.3f, quadratic: %.3f' % (
        mean_squared_error(y, y_lin_pred),
```

```
        mean_squared_error(y, y_quad_pred)))
print('Training R^2 linear: %.3f, quadratic: %.3f' % (
        r2_score(y, y_lin_pred),
        r2_score(y, y_quad_pred)))
```

```
[[1.00000e+00 2.58000e+02 6.65640e+04]
 [1.00000e+00 2.70000e+02 7.29000e+04]
 [1.00000e+00 2.94000e+02 8.64360e+04]
 [1.00000e+00 3.20000e+02 1.02400e+05]
 [1.00000e+00 3.42000e+02 1.16964e+05]
 [1.00000e+00 3.68000e+02 1.35424e+05]
 [1.00000e+00 3.96000e+02 1.56816e+05]
 [1.00000e+00 4.46000e+02 1.98916e+05]
 [1.00000e+00 4.80000e+02 2.30400e+05]
 [1.00000e+00 5.86000e+02 3.43396e+05]]
```

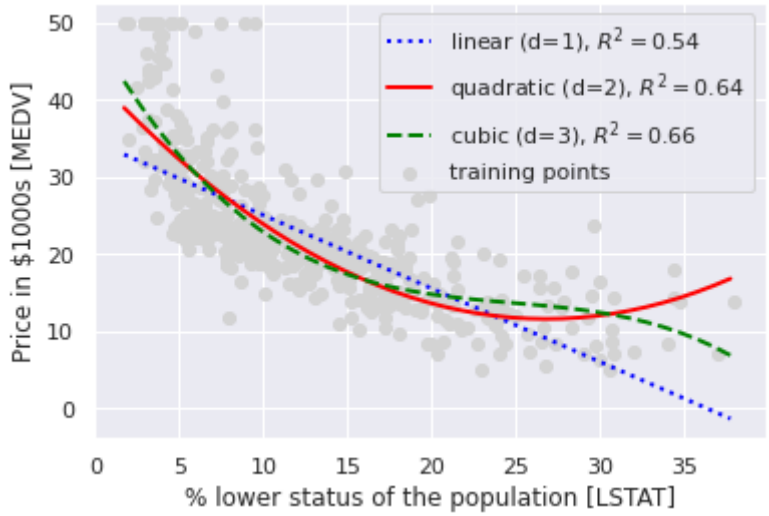



```
Training MSE linear: 569.780, quadratic: 61.330
Training R^2 linear: 0.832, quadratic: 0.982
```

In [69]:

```
X = boston_df[['LSTAT']].values
y = boston_df['MEDV'].values

regr = LinearRegression()

# create quadratic features
quadratic = PolynomialFeatures(degree=2)
cubic = PolynomialFeatures(degree=3)
X_quad = quadratic.fit_transform(X)
X_cubic = cubic.fit_transform(X)

# fit features
X_fit = np.arange(X.min(), X.max(), 1)[:, np.newaxis]

regr = regr.fit(X, y)
y_lin_fit = regr.predict(X_fit)
linear_r2 = r2_score(y, regr.predict(X))

regr = regr.fit(X_quad, y)
y_quad_fit = regr.predict(quadratic.fit_transform(X_fit))
quadratic_r2 = r2_score(y, regr.predict(X_quad))

regr = regr.fit(X_cubic, y)
y_cubic_fit = regr.predict(cubic.fit_transform(X_fit))
cubic_r2 = r2_score(y, regr.predict(X_cubic))


# plot results
plt.scatter(X, y, label='training points', color='lightgray')

plt.plot(X_fit, y_lin_fit,
```

```
          label='linear (d=1), $R^2=%.2f$' % linear_r2,
          color='blue',
          lw=2,
          linestyle=':')

plt.plot(X_fit, y_quad_fit,
          label='quadratic (d=2), $R^2=%.2f$' % quadratic_r2,
          color='red',
          lw=2,
          linestyle='-')

plt.plot(X_fit, y_cubic_fit,
          label='cubic (d=3), $R^2=%.2f$' % cubic_r2,
          color='green',
          lw=2,
          linestyle='--')

plt.xlabel('% lower status of the population [LSTAT]')
plt.ylabel('Price in $1000s [MEDV]')
plt.legend(loc='upper right')

plt.show()
```



　ボストンデータを線形回帰した。ADALINEを用いた勾配降下法を実装し、SSEにてエポックごとの学習を評価した。scikit-learnを用いた回帰も行った。
データ入力に上限がある影響もあるのか、外れ値に引きずられて回帰の結果が変わる傾向があり、RANSACアルゴリズムによって外れ値を避けながらの学習も試みた。
また、複数の特徴量を用いて重回帰も試み、その場合のモデルの性能評価も行った。非線形回帰も様々なパターンでこころみた。
分かりやすいデータであるせいもあるかとは思うが、どのような回帰をしてもある程度それらしい結果になる。適切なモデルを選択するためには、ある程度の情報の背景を勉強したうえで、エンジニアとしての判断も必要なのかなと感じた。