線形回帰モデル（NumPy）

In [1]:

```
import numpy as np
import matplotlib.pyplot as plt
%matplotlib inline
from sklearn.linear_model import LinearRegression
```

In [2]:

```
n_samples = 100  # サンプル数
var = .2
```

In [3]:

```
def linear_func(x: np.ndarray) -> np.ndarray:
    """ 2x + 5 の結果を返す """
    y = 2 * x + 5
    return y
```

In [4]:

```
def add_noise(y_true: np.ndarray, var: float) -> np.ndarray:
    """ 正規分布に従うノイズを追加する """
    noise = np.random.normal(scale=var, size=y_true.shape)  # scale : 標準偏差
    y_noise = y_true + noise
    return y_noise
```

In [5]:

```
xs = np.linspace(0, 1, n_samples)
ys_true = linear_func(xs)  # 実際のy (直線)
ys = add_noise(ys_true, var)  # 実際のyにランダムなノイズを追加する
```

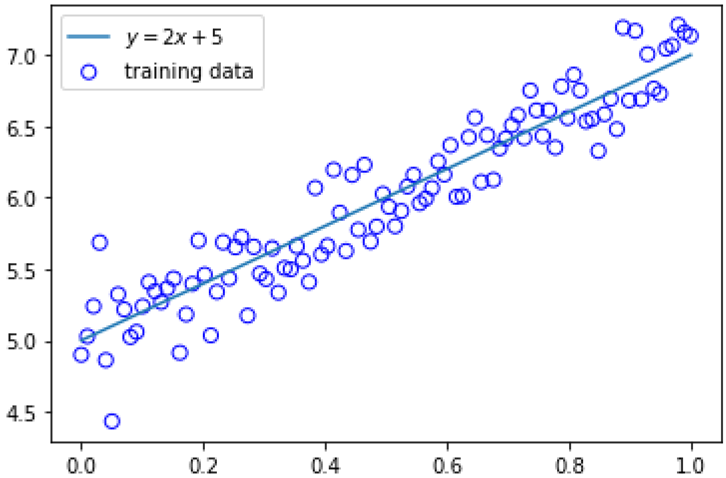
In [6]:

```
print(xs.shape)
print(ys.shape)
print(ys_true[:5])
print(ys[:5])
```

```
(100,)
(100,)
[5.         5.02020202 5.04040404 5.06060606 5.08080808]
[4.89917008 5.02872236 5.23997766 5.68572049 4.86151879]
```

In [7]:

```
# 結果をプロットする
# ノイズを追加したデータは散布図で描画
plt.scatter(xs, ys, facecolor="none", edgecolor="b", s=50,  label="training data")
# 実際の結果は直線で描画
plt.plot(xs, ys_true, label="$y = 2x + 5$")
plt.legend()
plt.show()
```



考察：
上のグラフから、$y = 2x + 5$ にランダムな値を加算して学習データが作成されていることがわかる。

学習

目的： y = ax + b の a, b を求める。
具体的には
(1) $\hat{a} = \mathrm{Cov}[x, y]/\mathrm{Var}[x]$
(2) $\hat{b} = \mu_y - \hat{a}\mu_x$ で求める。
μ は、平均を意味する。

In [8]:

```
def train(xs: np.ndarray, ys: np.ndarray):
    """ 回帰の実装 """
    cov = np.cov(xs, ys, ddof=0)  # ddof=0なら、平均を返す。
    cov_xy = cov[0, 1]
    var_x = cov[0, 0]  # 共分散行列の(0, 0)は、xの分散

    a = cov_xy / var_x  # (1) aを求める
    b = np.mean(ys) - a * np.mean(xs)  # (2) bを求める
    return cov, a, b

cov, a, b = train(xs, ys)
print(f'共分散:\n{cov}')
print(f'係数a: {a}')
print(f'切片b: {b}')
```

```
共分散:
[[0.08501684 0.17516249]
 [0.17516249 0.40390536]]
係数a: 2.0603271612606227
切片b: 4.957649247978992
```

In [9]:

```
# 比較として、scikit-learnの出力を表示する
model_lr = LinearRegression()
# sklearnのfit使用時は、ndarrayを reshape(-1, 1) する (n行1列にする)
history_lr = model_lr.fit(xs.reshape(-1, 1), ys.reshape(-1, 1))

print(f'係数a: {history_lr.coef_}')
print(f'切片b: {history_lr.intercept_}')
```

```
係数a: [[2.06032716]]
切片b: [4.95764925]
```

考察：
出力された係数と切片を見ると、$y = 2x + 5$ に非常に近い予測が出来ていることがわかる。

予測

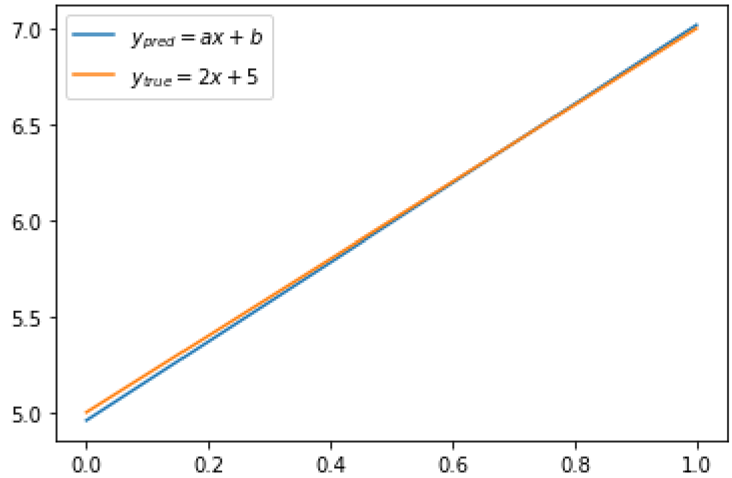
上で学習した a, b から y_pred = ax + b の直線を描画し、元の y_true と比較する。

In [11]:

```
ys_pred = a * xs + b
```

In [12]:

```
# 比較結果をプロットする
# 実際の結果は直線で描画
plt.plot(xs, ys_pred, label="$y_{pred} = ax + b$")
plt.plot(xs, ys_true, label="$y_{true} = 2x + 5$")
plt.legend()
plt.show()
```



考察：
先ほど出力された係数と切片からy_predの直線を描画し、元の $y = 2x + 5$ と比較すると、ほぼ重なっているように見える。

重回帰分析

訓練データ生成（三次元入力）

In [13]:

```
np.random.random((10, 3))  # 10行3列で0以上1未満の乱数を作成
```

Out[13]:

```
array([[0.89601987, 0.48827978, 0.58282269],
       [0.72751961, 0.27124584, 0.49551241],
       [0.24907989, 0.22434333, 0.97440657],
       [0.04677017, 0.56660098, 0.0277153 ],
       [0.67087055, 0.03291724, 0.71484886],
       [0.80550533, 0.16851497, 0.00651588],
       [0.10966505, 0.11739847, 0.07252016],
       [0.55251786, 0.74530996, 0.24719961],
       [0.25234939, 0.39189595, 0.19284386],
       [0.32240546, 0.97832625, 0.51198802]])
```

In [28]:

```
n_sample = 100
var = .2
x_dim = 3
```

下記グラフは $y = 1.0 + 0.5x_0 + 2x_1 + x_2$ を表す。

In [29]:

```
def mul_linear_func(x: np.ndarray) -> np.ndarray:
    """ y = 1.0 + 0.5*x_0 + 2*x_1 + x_2 """
    ww = [1., 0.5, 2., 1.]
    return ww[0] + ww[1] * x[:, 0] + ww[2] * x[:, 1] + ww[3] * x[:, 2]
```

In [30]:

```
def add_noise(y_true: np.ndarray, var: float) -> np.ndarray:
    """元のデータにランダムなノイズを追加する"""
    noise = np.random.normal(scale=var, size=y_true.shape)
    result = y_true + noise
    return result
```

In [31]:

```
X = np.random.random((n_sample, x_dim))
ys_true = mul_linear_func(X)  # 元データ
ys = add_noise(ys_true, var)  # ノイズを追加
```

In [32]:

```
print(X.shape)
print(ys.shape)
```

```
(100, 3)
(100,)
```

In [33]:

```
X[:3]
```

Out[33]:

```
array([[0.63445253, 0.9490093 , 0.25176967],
       [0.12092448, 0.0817593 , 0.0380423 ],
       [0.05260663, 0.06635887, 0.27975522]])
```

学習

求める回帰係数 w は以下のように書ける。

$$\hat{w} = \left(X^{\mathrm{T}} X\right)^{-1} X^{\mathrm{T}} y$$

In [34]:

```
def add_one(X: np.ndarray) -> np.ndarray:
    """ 行列Xの1列目に全て1の列を追加する """
    ones = np.ones(len(X))[:, None]
    added_X = np.concatenate([ones, X], axis=1)
    return added_X
```

In [35]:

```
X_train = add_one(X)  # Xに1の列を追加する
```

In [36]:

```
X_train[:3]  # 1の列が追加されていることを確認
```

Out[36]:

```
array([[1.        , 0.63445253, 0.9490093 , 0.25176967],
       [1.        , 0.12092448, 0.0817593 , 0.0380423 ],
       [1.        , 0.05260663, 0.06635887, 0.27975522]])
```

In [37]:

```
# 公式にならって w を求める
tmp = np.dot(X_train.T, X_train)
tmp = np.linalg.inv(tmp)
tmp = np.dot(tmp, X_train.T)
w = np.dot(tmp, ys)
```

In [38]:

```
print(w)
```

```
[0.88688741 0.50534656 2.11413418 1.0743534 ]
```

予測

入力に対する値を $y(x) = \hat{w}^{\mathrm{T}} x \ (y = X\hat{w})$ で予測する

In [39]:

```
w_true = [1., 0.5, 2., 1.]  # 正解のwの値
```

In [40]:

```
for i in range(len(w)):
    print("w{0}_true: {1:>5.2}   w{0}_estimated: {2:>5.2}".format(i, w_true[i], w[i]))
```

```
w0_true:   1.0   w0_estimated:  0.89
w1_true:   0.5   w1_estimated:  0.51
w2_true:   2.0   w2_estimated:   2.1
w3_true:   1.0   w3_estimated:   1.1
```

どの w_i についても、元の値と予測値が非常に近いことが確認できた。

In [ ]:

```

```