「手書き数字」認識課題

課題：手書き数字を8×8(各要素は0～16の整数値）のビットマップ画像の64次元ベクトル x として入力すると、その数字 y を予測する。

y = f(p, x)

データセットの読み込み

```
from sklearn.datasets import load_digits
digits = load_digits()  # データセットの読み込み
```

(1) データセットの確認

```
digits.keys()
```

```
dict_keys(['data', 'target', 'frame', 'feature_names', 'target_names', 'images', 'DESCR'])
```

```
print(digits.DESCR)  # データ数：1797、特徴量：64（8×8）
```

```
.. _digits_dataset:

Optical recognition of handwritten digits dataset
--------------------------------------------------

**Data Set Characteristics:**

:Number of Instances: 1797
:Number of Attributes: 64
:Attribute Information: 8x8 image of integer pixels in the range 0..16.
:Missing Attribute Values: None
:Creator: E. Alpaydin (alpaydin '@' boun.edu.tr)
:Date: July; 1998

This is a copy of the test set of the UCI ML hand-written digits datasets
https://archive.ics.uci.edu/ml/datasets/Optical+Recognition+of+Handwritten+Digits

The data set contains images of hand-written digits: 10 classes where
each class refers to a digit.

Preprocessing programs made available by NIST were used to extract
normalized bitmaps of handwritten digits from a preprinted form. From a
total of 43 people, 30 contributed to the training set and different 13
to the test set. 32x32 bitmaps are divided into nonoverlapping blocks of
4x4 and the number of on pixels are counted in each block. This generates
an input matrix of 8x8 where each element is an integer in the range
0..16. This reduces dimensionality and gives invariance to small
distortions.

For info on NIST preprocessing routines, see M. D. Garris, J. L. Blue, G.
T. Candela, D. L. Dimmick, J. Geist, P. J. Grother, S. A. Janet, and C.
L. Wilson, NIST Form-Based Handprint Recognition System, NISTIR 5469,
1994.

.. dropdown:: References

  - C. Kaynak (1995) Methods of Combining Multiple Classifiers and Their
    Applications to Handwritten Digit Recognition, MSc Thesis, Institute of
    Graduate Studies in Science and Engineering, Bogazici University.
  - E. Alpaydin, C. Kaynak (1998) Cascading Classifiers, Kybernetika.
  - Ken Tang and Ponnuthurai N. Suganthan and Xi Yao and A. Kai Qin.
    Linear dimensionalityreduction using relevance weighted LDA. School of
    Electrical and Electronic Engineering Nanyang Technological University.
    2005.
  - Claudio Gentile. A New Approximate Maximal Margin Classification
    Algorithm. NIPS. 2000.
```

+ コード + テキスト

```
print(digits.target_names)
```

```
[0 1 2 3 4 5 6 7 8 9]
```

```
print(digits.feature_names)  # 特徴量の名前
```

```
['pixel_0_0', 'pixel_0_1', 'pixel_0_2', 'pixel_0_3', 'pixel_0_4', 'pixel_0_5', 'pixel_0_6', 'pixel_0_7', 'pixel_1_0', 'pixel_1_1', 'pixel_1_2', '
```

```
print(type(digits.data))  # numpy.ndarray
print(digits.data.shape)  # 個数: 1797, 要素数: 64
```

```
<class 'numpy.ndarray'>
(1797, 64)
```

```
print(digits.data[:5])
```

```
[[ 0.  0.  5. 13.  9.  1.  0.  0.  0.  0. 13. 15. 10. 15.  5.  0.  0.  3.
  15.  2.  0. 11.  8.  0.  0.  4. 12.  0.  0.  8.  8.  0.  0.  5.  8.  0.
   0.  9.  8.  0.  0.  4. 11.  0.  1. 12.  7.  0.  0.  2. 14.  5. 10. 12.
   0.  0.  0.  0.  6. 13. 10.  0.  0.  0.]
 [ 0.  0.  0. 12. 13.  5.  0.  0.  0.  0.  0. 11. 16.  9.  0.  0.  0.  0.
   3. 15. 16.  6.  0.  0.  0.  7. 15. 16. 16.  2.  0.  0.  0.  0.  1. 16.
  16.  3.  0.  0.  0.  0.  1. 16. 16.  6.  0.  0.  0.  0.  1. 16. 16.  6.
   0.  0.  0.  0.  0. 11. 16. 10.  0.  0.]
 [ 0.  0.  0.  4. 15. 12.  0.  0.  0.  0.  3. 16. 15. 14.  0.  0.  0.  0.
   8. 13.  8. 16.  0.  0.  0.  0.  1.  6. 15. 11.  0.  0.  0.  1.  8. 13.
  15.  1.  0.  0.  0.  9. 16. 16.  5.  0.  0.  0.  0.  3. 13. 16. 16. 11.
   5.  0.  0.  0.  0.  3. 11. 16.  9.  0.]
 [ 0.  0.  7. 15. 13.  1.  0.  0.  0.  8. 13.  6. 15.  4.  0.  0.  0.  2.
   1. 13. 13.  0.  0.  0.  0.  0.  2. 15. 11.  1.  0.  0.  0.  0.  0.  1.
  12. 12.  1.  0.  0.  0.  0.  0.  1. 10.  8.  0.  0.  0.  8.  4.  5. 14.
   9.  0.  0.  0.  7. 13. 13.  9.  0.  0.]
 [ 0.  0.  0.  1. 11.  0.  0.  0.  0.  0.  0.  7.  8.  0.  0.  0.  0.  0.
   1. 13.  6.  2.  2.  0.  0.  0.  7. 15.  0.  9.  8.  0.  0.  5. 16. 10.
   0. 16.  6.  0.  0.  4. 15. 16. 13. 16.  1.  0.  0.  0.  0.  3. 15. 10.
   0.  0.  0.  0.  0.  2. 16.  4.  0.  0.]]
```

```
print(type(digits.target))  # numpy.ndarray
print(digits.target.shape)  # データ数: 1797, 要素数: 1
```

```
<class 'numpy.ndarray'>
(1797,)
```

```
print(digits.target[:100])
```

```
[0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 9 5 5 6 5 0
 9 8 9 8 4 1 7 7 3 5 1 0 0 2 2 7 8 2 0 1 2 6 3 3 7 3 3 4 6 6 6 4 9 1 5 0 9
 5 2 8 2 0 0 1 7 6 3 2 1 7 4 6 3 1 3 9 1 7 6 8 4 3 1]
```

```
# 各数字のデータ個数
digits_target_list = list(digits.target)
for n in range(10):
  print(f"{n}: {digits_target_list.count(n)}")
```

```
0: 178
1: 182
2: 177
3: 183
4: 181
5: 182
6: 181
7: 179
8: 174
9: 180
```

```
print(type(digits.images))  # numpy.ndarray
print(digits.images.shape)  # imageは8x8の2次元配列
```

```
<class 'numpy.ndarray'>
(1797, 8, 8)
```

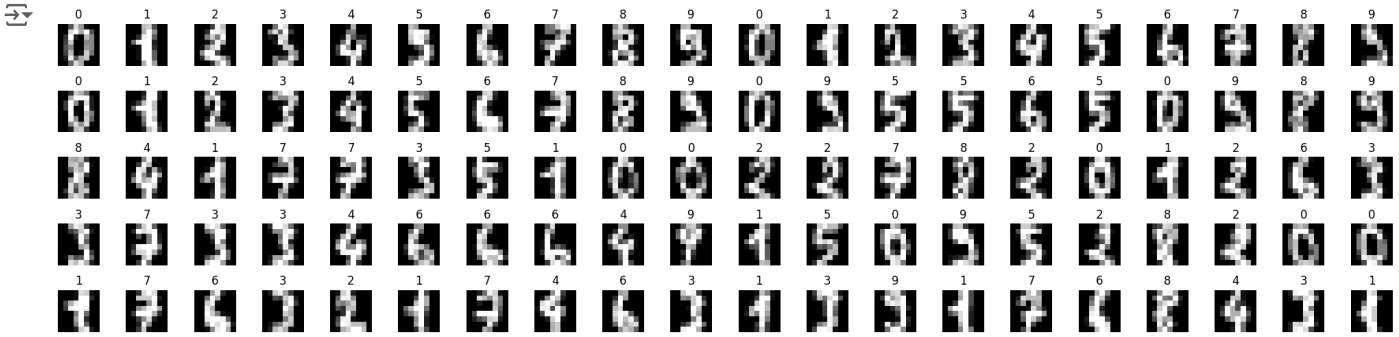
```
print(digits.images[:5])
```

```
[[[ 0.  0.  5. 13.  9.  1.  0.  0.]
  [ 0.  0. 13. 15. 10. 15.  5.  0.]
  [ 0.  3. 15.  2.  0. 11.  8.  0.]
  [ 0.  4. 12.  0.  0.  8.  8.  0.]
  [ 0.  5.  8.  0.  0.  9.  8.  0.]
  [ 0.  4. 11.  0.  1. 12.  7.  0.]
  [ 0.  2. 14.  5. 10. 12.  0.  0.]
  [ 0.  0.  6. 13. 10.  0.  0.  0.]]

 [[ 0.  0.  0. 12. 13.  5.  0.  0.]
  [ 0.  0.  0. 11. 16.  9.  0.  0.]
  [ 0.  0.  3. 15. 16.  6.  0.  0.]
  [ 0.  7. 15. 16. 16.  2.  0.  0.]
  [ 0.  0.  1. 16. 16.  3.  0.  0.]
  [ 0.  0.  1. 16. 16.  6.  0.  0.]
  [ 0.  0.  1. 16. 16.  6.  0.  0.]
```

```
        [ 0.  0.  0. 11. 16. 10.  0.  0.]]

       [[ 0.  0.  0.  4. 15. 12.  0.  0.]
        [ 0.  0.  3. 16. 15. 14.  0.  0.]
        [ 0.  0.  8. 13.  8. 16.  0.  0.]
        [ 0.  0.  1.  6. 15. 11.  0.  0.]
        [ 0.  1.  8. 13. 15.  1.  0.  0.]
        [ 0.  9. 16. 16.  5.  0.  0.  0.]
        [ 0.  3. 13. 16. 16. 11.  5.  0.]
        [ 0.  0.  0.  3. 11. 16.  9.  0.]]

       [[ 0.  0.  7. 15. 13.  1.  0.  0.]
        [ 0.  8. 13.  6. 15.  4.  0.  0.]
        [ 0.  2.  1. 13. 13.  0.  0.  0.]
        [ 0.  0.  2. 15. 11.  1.  0.  0.]
        [ 0.  0.  0.  1. 12. 12.  1.  0.]
        [ 0.  0.  0.  0.  1. 10.  8.  0.]
        [ 0.  0.  8.  4.  5. 14.  9.  0.]
        [ 0.  0.  7. 13. 13.  9.  0.  0.]]

       [[ 0.  0.  0.  1. 11.  0.  0.  0.]
        [ 0.  0.  0.  7.  8.  0.  0.  0.]
        [ 0.  0.  1. 13.  6.  2.  2.  0.]
        [ 0.  0.  7. 15.  0.  9.  8.  0.]
        [ 0.  5. 16. 10.  0. 16.  6.  0.]
        [ 0.  4. 15. 16. 13. 16.  1.  0.]
        [ 0.  0.  0.  3. 15. 10.  0.  0.]
        [ 0.  0.  0.  2. 16.  4.  0.  0.]]]
```

(2) データ画像を確認

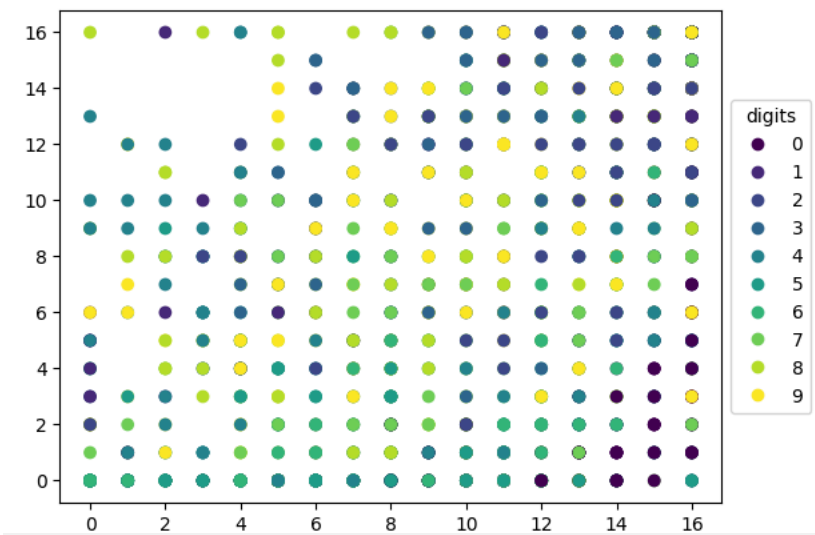
```
# 最初の100個のデータを表示
import matplotlib.pyplot as plt

plt.figure(figsize=(20, 5))
for i in range(100):
  plt.subplot(5, 20, i + 1)  # 5x20のグリッドに配置
  plt.imshow(digits.images[i], cmap='gray')
  plt.title(digits.target[i])
  plt.axis('off')  # 軸を非表示

plt.tight_layout()  # レイアウト調整
plt.show()
```



(3) 特徴量をプロット

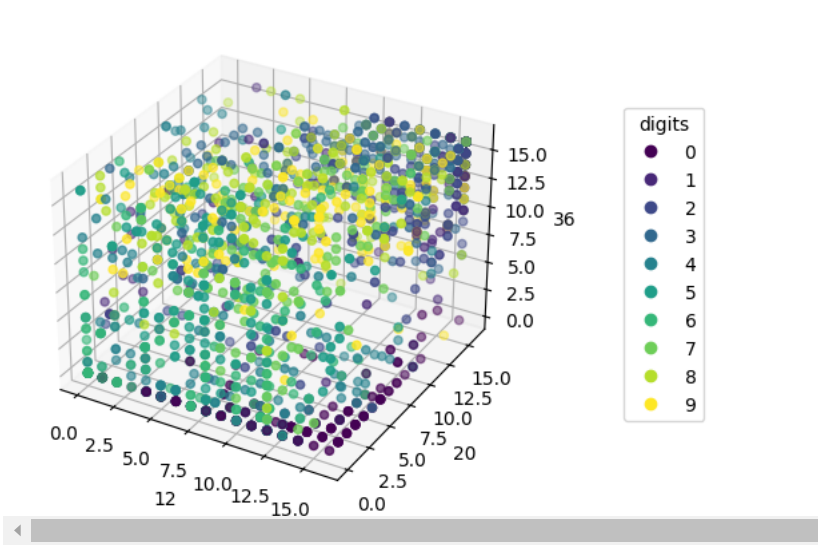
```
import matplotlib.pyplot as plt
import numpy as np

fig, ax = plt.subplots()
x = digits.data[:, 12]
y = digits.data[:, 20]

# 散布図を作成
scatter = ax.scatter(x, y, c=digits.target, cmap='viridis')

# 凡例用のラベルを作成
handles, labels = scatter.legend_elements()

# 凡例を表示（図の外側右横に表示）
ax.legend(handles, labels, loc='center left', bbox_to_anchor=(1, 0.5), title="digits")
```

```
plt.show()
```



データは64次元空間の点、値は0〜16。次元が高いので単純な2次元プロットではクラス分けの可否を判定できない。

```
# 特徴量の3次元プロット
fig, ax = plt.subplots(subplot_kw={"projection": "3d"})
x = digits.data[:, 12]
y = digits.data[:, 20]
z = digits.data[:, 28]

# 散布図を作成
scatter = ax.scatter(x, y, z, c=digits.target, cmap='viridis')
ax.set_xlabel("12")
ax.set_ylabel("20")
ax.set_zlabel("36")

# 凡例を表示（図の外側右横に表示）
ax.legend(handles, labels, loc='center left', bbox_to_anchor=(1.2, 0.5), title="digits")

plt.show()
```



単純に3次元プロットしてもクラス分けの可否は分からない。

主成分分析 (PCA) を用いて次元削減を行い、2次元または3次元で可視化する

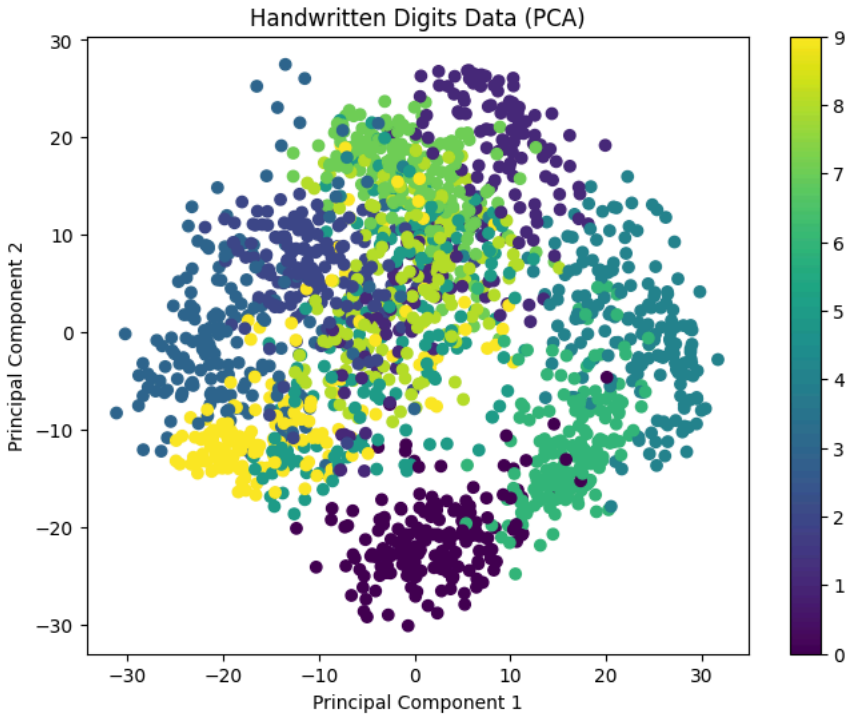
数字画像は 64 次元 (8x8 ピクセル) のデータであるが、PCA(Principal Component Analysis) を用いて次元数を削減することで、主要な特徴を保持したまま 2 次元または 3 次元で可視化できる。これにより、数字のクラスごとにデータがどのように分布しているのか、視覚的に確認できる。 **主成分分析(PCA)は、高次元データをより低次元で表現するための手法である。**

PCAの仕組み:

データの分散を最大化する方向を探す: PCAは、データが最もばらついている方向を探す。この方向が、データの特徴を最もよく表す第1主成分となる。 第1主成分と直交する方向を探す: 次に、第1主成分と直交する方向で、データの分散が最大となる方向を探します。これが第2主成分となります。 主成分を必要な数だけ求める: このように、分散が最大となる方向を順番に求めていくことで、必要な数だけ主成分を求めることができる。

主成分2次元で可視化

```
from sklearn.decomposition import PCA

# 手書き数字データの読み込み
X = digits.data
y = digits.target

# PCA を用いて 2 次元に次元削減
pca = PCA(n_components=2)
X_reduced = pca.fit_transform(X)

# 2 次元で可視化
plt.figure(figsize=(8, 6))
plt.scatter(X_reduced[:, 0], X_reduced[:, 1], c=y, cmap='viridis')
plt.colorbar()
plt.title('Handwritten Digits Data (PCA)')
plt.xlabel('Principal Component 1')
plt.ylabel('Principal Component 2')
plt.show()

# 第1, 2主成分
first_principal_component = pca.components_[0]
second_principal_component = pca.components_[1]

# 有効数字3桁で主成分ベクトル表示
print("First Principal Component:", [np.format_float_scientific(x, precision=3) for x in first_principal_component])
print("Second Principal Component:", [np.format_float_scientific(x, precision=3) for x in second_principal_component])
```

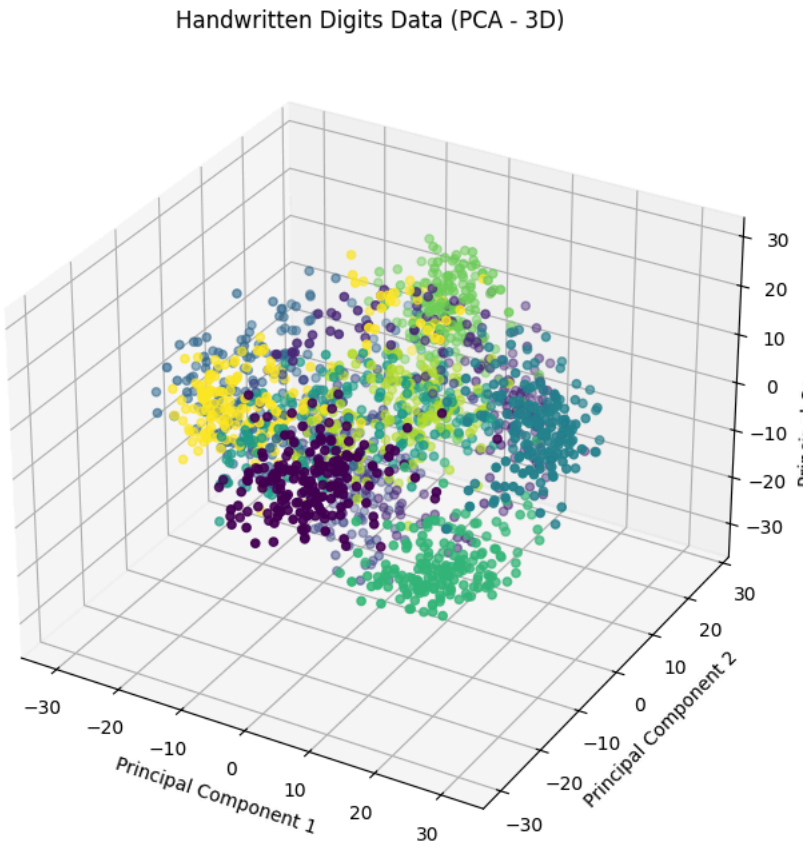



```
First Principal Component: ['0.e+00', '-1.731e-02', '-2.234e-01', '-1.359e-01', '-3.303e-02', '-9.663e-02', '-8.329e-03', '2.269e-03', '-3.205e-(
Second Principal Component: ['0.e+00', '1.011e-02', '4.908e-02', '9.433e-03', '5.360e-02', '1.178e-01', '6.213e-02', '7.936e-03', '1.632e-04', '2
```

主成分3次元で可視化

```
from mpl_toolkits.mplot3d import Axes3D  # 3Dプロットに必要なライブラリ
```

```
# PCAを用いて3次元に次元削減
pca = PCA(n_components=3)  # 次元数を3に設定
X_reduced = pca.fit_transform(X)

# 3次元で可視化
fig = plt.figure(figsize=(10, 8))
ax = fig.add_subplot(111, projection='3d')  # 3Dプロットを作成
ax.scatter(X_reduced[:, 0], X_reduced[:, 1], X_reduced[:, 2], c=y, cmap='viridis')  # y = digits.target
ax.set_xlabel('Principal Component 1')
ax.set_ylabel('Principal Component 2')
ax.set_zlabel('Principal Component 3')
plt.title('Handwritten Digits Data (PCA - 3D)')
plt.show()

first_principal_component = pca.components_[0]
second_principal_component = pca.components_[1]
third_principal_component = pca.components_[2]

# 有効数字3桁で主成分ベクトル表示
print("First Principal Component:", [np.format_float_scientific(x, precision=3) for x in first_principal_component])
print("Second Principal Component:", [np.format_float_scientific(x, precision=3) for x in second_principal_component])
print("Third Principal Component:", [np.format_float_scientific(x, precision=3) for x in third_principal_component])
```



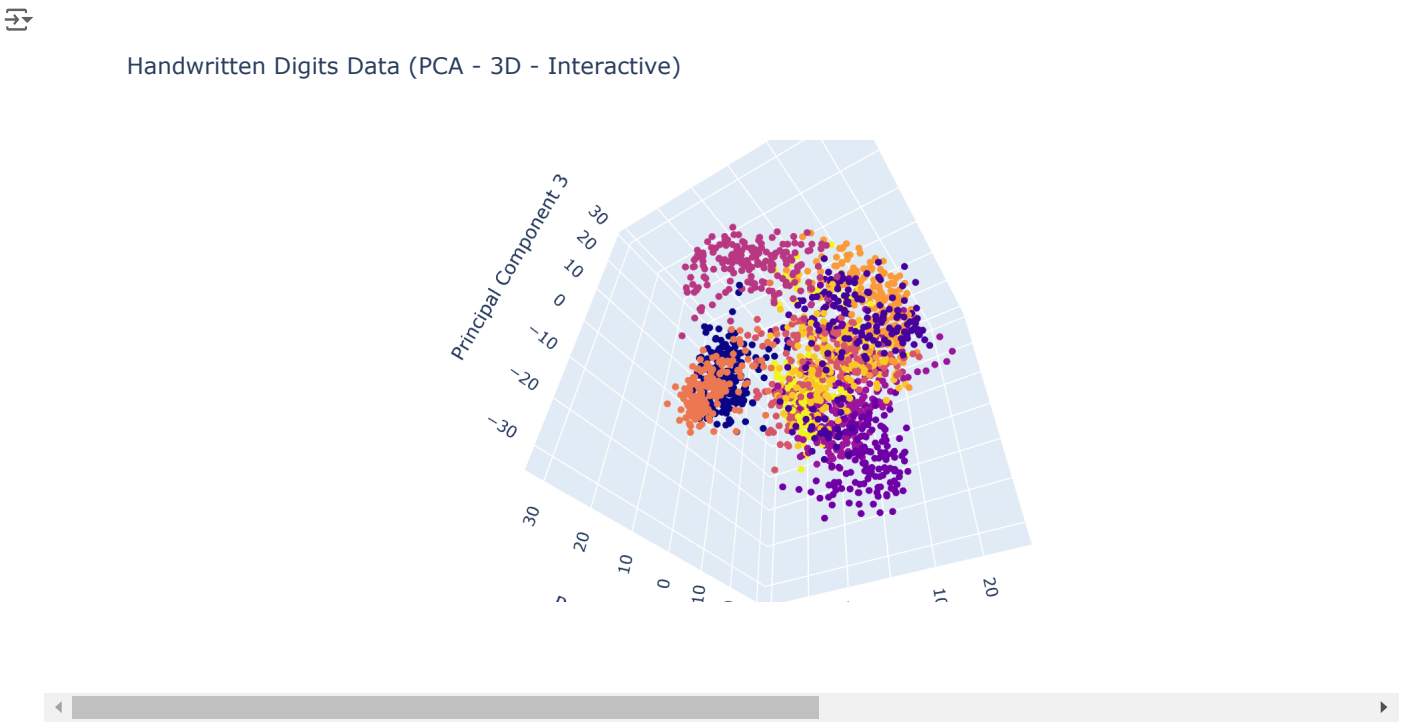

```
First Principal Component: ['0.e+00', '-1.731e-02', '-2.234e-01', '-1.359e-01', '-3.303e-02', '-9.663e-02', '-8.329e-03', '2.269e-03', '-3.205e-(
Second Principal Component: ['0.e+00', '1.011e-02', '4.908e-02', '9.433e-03', '5.360e-02', '1.178e-01', '6.213e-02', '7.936e-03', '1.632e-04', '2
Third Principal Component: ['0.e+00', '-1.834e-02', '-1.265e-01', '-1.322e-01', '1.340e-01', '2.649e-01', '1.166e-01', '1.684e-02', '-3.94e-04',
```

```
!pip install plotly==5.13.1
```

```
Collecting plotly==5.13.1
  Downloading plotly-5.13.1-py2.py3-none-any.whl.metadata (7.0 kB)
Requirement already satisfied: tenacity>=6.2.0 in /usr/local/lib/python3.10/dist-packages (from plotly==5.13.1) (9.0.0)
Downloading plotly-5.13.1-py2.py3-none-any.whl (15.2 MB)
   ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 15.2/15.2 MB 67.0 MB/s eta 0:00:00
Installing collected packages: plotly
  Attempting uninstall: plotly
    Found existing installation: plotly 5.24.1
    Uninstalling plotly-5.24.1:
      Successfully uninstalled plotly-5.24.1
Successfully installed plotly-5.13.1
```

```
import plotly.express as px

# PCAを用いて3次元に次元削減
```

```
pca = PCA(n_components=3)
X_reduced = pca.fit_transform(X)

# plotly.expressで3次元散布図を作成
fig = px.scatter_3d(x=X_reduced[:, 0], y=X_reduced[:, 1], z=X_reduced[:, 2],
                    color=y,
                    labels={'x': 'Principal Component 1', 'y': 'Principal Component 2', 'z': 'Principal Component 3'},
                    title='Handwritten Digits Data (PCA - 3D - Interactive)',
                    )  # 点の最大サイズを指定 # Corrected indentation

# 点のサイズを小さく設定
fig.update_traces(marker=dict(size=2)) # sizeの値を小さくする

fig.show()
```



(4) 機械学習: Support Vector Machine

```
# データとラベルを抽出
from sklearn.model_selection import train_test_split
X = digits.data  # ベクトル（配列）は大文字にするのが習慣
y = digits.target

# データセットを訓練データとテストデータに分割
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=9)

from sklearn.svm import SVC
from sklearn.metrics import accuracy_score

# SVMモデルのインスタンス作成と訓練
model = SVC()
model.fit(X_train, y_train)
```

```
▾ SVC
SVC()
```

(5) テスト

```
# テストデータに対する予測
y_pred = model.predict(X_test)

# 精度の評価
accuracy = accuracy_score(y_test, y_pred)
print(f"予測精度: {accuracy:.5f}")
```

```
n_wrong = (y_pred != y_test).sum()
print(f"間違った予測の数: {n_wrong}")
```

```
予測精度: 0.98889
間違った予測の数: 4
```

```
# 間違った予測の可視化
fig, axes = plt.subplots(1, n_wrong, figsize=(15, 2))
k = 0
for i in range(len(y_test)):
    if y_pred[i] != y_test[i]:
        axes[k].imshow(X_test[i].reshape(8, 8), cmap='gray')
        axes[k].set_title(f"P: {y_pred[i]}, C: {y_test[i]}")
        axes[k].axis('off')
        k += 1
plt.show()
```

機械学習：k-最近傍

あるデータに対する予測値は、学習データの中で最も近いデータのラベル（クラス）とする。なお、kはパラメータで最も近いk個の最近傍データの多数決でクラスを決める。

```
from sklearn.neighbors import KNeighborsClassifier
from sklearn.preprocessing import StandardScaler
scaler = StandardScaler()
# データの規格化
X_train_std = scaler.fit_transform(X_train)
X_test_std = scaler.transform(X_test)

digits_knn = KNeighborsClassifier(n_neighbors=3)
digits_knn.fit(X_train_std, y_train)

digits_knn_score = digits_knn.score(X_test_std, y_test)
print(f'digits_knn_score= {digits_knn_score:.5f}')
```

```
digits_knn_score= 0.97222
```

Colab上で手書き数字を描き、手書き数字認識する

```
# 参考： https://qiita.com/MKen_bu/items/2603700fa8ae65c92a6b

from IPython.display import display, HTML  # IPythonのHTML(), display()を利用
from google.colab import output
from google.colab.output import eval_js  # PythonからJavaScript関数の呼び出し
import base64  # canvasデータ（base64エンコード）をデコード
import re  # 正規表現
import io
from PIL import Image, ImageFilter
import numpy as np


html = HTML('''
<canvas id="canvas" height="140" width="140" style="border-style: solid; border-color: black;"></canvas>
<div>
  <button type="button" id="predictBtn" style="margin-top:20px">予測</button>
  <button type="button" id="clearBtn" style="margin-top:20px">クリア</button>
</div>
<div id="resultDiv"></div>

<!-- Fabric.jsの読み込み：Canvasライブラリ -->
```

```
<script src="https://cdnjs.cloudflare.com/ajax/libs/fabric.js/4.5.0/fabric.min.js"></script>

<script>
  const canvas = new fabric.Canvas("canvas", {isDrawingMode: true, backgroundColor: 'black'});
  canvas.freeDrawingBrush.width = 16;
  canvas.freeDrawingBrush.color = 'white';

  // https://colab.research.google.com/notebooks/snippets/advanced_outputs.ipynb#scrollTo=QS5x4lFfOfJE
  document.getElementById("predictBtn").addEventListener("click", () => {
                                      //The callback name, The arguments, kwargs
    google.colab.kernel.invokeFunction('PredictImage', [canvas.toDataURL()], {})
  });

  document.getElementById("clearBtn").addEventListener("click", () => {
        canvas.clear();
        canvas.backgroundColor = 'black';
  });


  displayResult = (array, n) => {
    document.getElementById("resultDiv").insertAdjacentHTML('afterbegin', '<div style="margin-bottom: 1em">' + array + '<br>予測値:
  };

</script>
''')
display(html)  # htmlを表示

# Python ---------------------------------------------
def predictImage(imageCode):
    # canvasデータ取得
    decodedImage = base64.b64decode(re.sub('', '', imageCode))
    # pilで読み込み
    pil_image = Image.open(io.BytesIO(decodedImage))
    # グレイスケール
    pil_image_gray = pil_image.convert("L")  # 8bitグレイスケール
    # リサイズ
    resized_image = pil_image_gray.resize((8, 8))
    np_image = np.array(resized_image, dtype=int)
    # 正規化
    np_image //= 16
    #1次元化: 64次元ベクトル
    x = np_image.reshape(64)
    prediction = model.predict([x])
    n = prediction[0]
    array = ""
    for i in range(64):
        array += str(x[i]) + '<br>' if (i+1) % 8 == 0 else str(x[i]) + ','

    #print(array)
    #print("予測値", n)
    #id="resuletDiv"の先頭に追加:jsからPython
    eval_js('displayResult("{}","{}")'.format(array, n))

    #print(f'{np_image=}')
    #print('予測結果： ', prediction[0])


output.register_callback('PredictImage', predictImage)
```

予測 クリア

参考：Geminiが提示したコード ----------

```
# prompt: sikit-learnから手書き数字のデータセットを読込、手書き数字認識して下さい

import matplotlib.pyplot as plt
from sklearn.datasets import load_digits
from sklearn.model_selection import train_test_split
from sklearn.linear_model import LogisticRegression
from sklearn.metrics import accuracy_score

# データセットの読み込み
digits = load_digits()

# データとラベルを抽出
X = digits.data
y = digits.target

# データセットを訓練データとテストデータに分割
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)

# ロジスティック回帰モデルのインスタンス作成と訓練
model = LogisticRegression(max_iter=10000) # max_iterを増やすことで収束しない問題を回避
model.fit(X_train, y_train)

# テストデータに対する予測
y_pred = model.predict(X_test)

# 精度の評価
accuracy = accuracy_score(y_test, y_pred)
print(f"予測精度: {accuracy}")

# 予測結果の可視化（最初の5個）
fig, axes = plt.subplots(1, 5, figsize=(10, 2))
for i in range(5):
    axes[i].imshow(X_test[i].reshape(8, 8), cmap='gray')
    axes[i].set_title(f"Predicted: {y_pred[i]}, Actual: {y_test[i]}")
    axes[i].axis('off')
plt.show()
```

予測精度: 0.975