MNIST DATASET을 이용한 머신러닝 모델 최적화 및 분석

전자정보공학부 IT융합전공 신성식 (20192601)
전자정보공학부 IT융합전공 김정현 (20160455)
전자정보공학부 IT융합전공 박준혁 (20170590)
전자정보공학부 IT융합전공 강문준 (20170550)

Check Data

가. MNIST dataset 개요 및 분석

Check Data

```
# MNIST data Load
from sklearn.datasets import fetch_openml
mnist = fetch_openml('mnist_784', version=1)
mnist.keys()
```

```
dict_keys(['data', 'target', 'frame', 'categories', 'feature_names', 'target_nam
es', 'DESCR', 'details', 'url'])
```

- Mnist data 를 fetch_openml에서 가져와 새로운 data instance를 추가하기 위한 data format 확인

```
X, y = mnist["data"], mnist["target"] # X, y: pandas DataFrame
X, y = X.to_numpy(), y.to_numpy() # X, y: numpy array 이용자따라 주석처리할 것
y = y.astype(np.uint8)
print(X.shape, X.dtype)
print(y.shape, y.dtype)
```

```
(70000, 784) float64
(70000,) uint8
```

- 결과로 X는 float64 y는 unit8로 확인.

```
def plot_digits(instances, images_per_row=10, **options):
    size = 28
    images_per_row = min(len(instances), images_per_row)
    n_rows = (len(instances) - 1) // images_per_row + 1

    n_empty = n_rows * images_per_row - len(instances)
    padded_instances = np.concatenate([instances, np.zeros((n_empty, size * size))], axis=0)

    image_grid = padded_instances.reshape((n_rows, images_per_row, size, size))

    big_image = image_grid.transpose(0, 2, 1, 3).reshape(n_rows * size, images_per_row * size)
    plt.imshow(big_image, cmap = mpl.cm.binary, **options)
    plt.axis("off")
```

```
plt.figure(figsize=(9,9))
plot_digits(X[100:200], images_per_row=10)
save_fig("more_digits_plot")
plt.show()
```

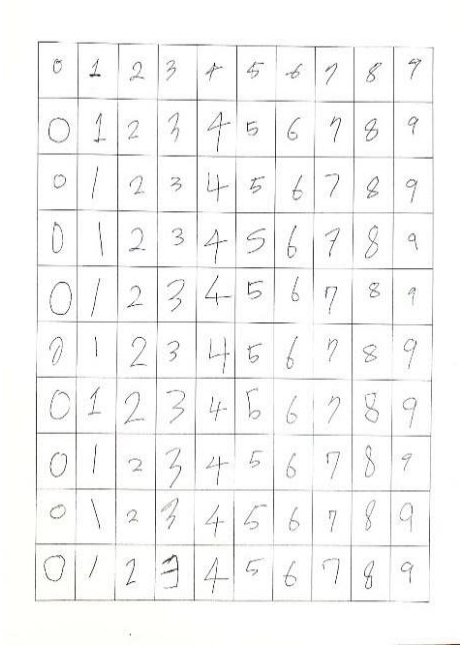
```
Saving figure more_digits_plot
```

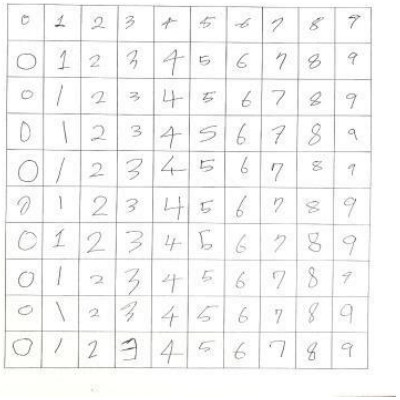
- 데이터가 어떻게 되어있는 지 시각적으로 확인.
- (선들이 없는 것을 확인)

Data Collection and Preprocessing

나. 새로운 data instance 수집 및 MNIST에 추가

New Data Collection





```
import numpy as np
import os
import cv2
```

```
url = './images/'
for i in range(1, 18):
    filename = url + f'MNIST{i}.jpg'
    img = cv2.imread(filename)
    resized_image = cv2.resize(img, (380, 380), interpolation=cv2.INTER_AREA)
    cv2.imwrite(f'./resized_images/MNIST{i}.jpg', resized_image)
```

- 데이터수집
  데이터 수집은 조원 한 명당 3장씩 12장을 기본으로 하며, 프로젝트를 진행하는 동안 지속적으로 추가하였음.
  (총 42장의 데이터를 추가함.)

- 380x380 해상도 변환
  주어진 조건은 이미지 해상도 380x380 임으로 해상도를 변경하기 위한 기초작업이 필요함.

  이미지를 불러오는 경로를 미리 정의하고, resize 함수를 사용하여 380x380 사이즈로 이미지의 크기를 조정해 주었음.

Data Collection and Preprocessing

Data(Image) Load

```python
def load_img(path): # path내의 모든 jpg 파일 불러오기
    files = glob.glob(path + '*.jpg')
    img_list = []
    for file in files:
        img = cv2.imread(file)
        img_list.append(img)
    return img_list

def show_image_list(img_list, figsize=(10, 4)): # 이미지 리스트 출력
    i = 1
    for img in img_list:
        plt.figure(figsize=figsize)
        plt.title(i)
        plt.imshow(img)
        i += 1
    plt.show()

def show(img, figsize=(10, 4), title=None): # 이미지 하나 출력
    plt.figure(figsize=figsize)
    plt.imshow(img)
    if title:
        plt.title(title)
    plt.show()
```

- 이미지 불러오기
  간단한 이미지 불러오기 작업을 수행함. 380x380인 이미지를 리스트에 담아, openCV를 사용하여 처리함.

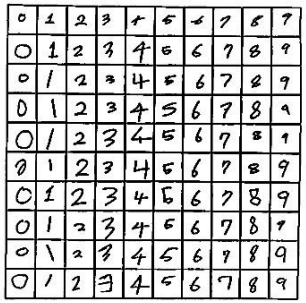
  load_img 함수는 glob를 이용하여 path 내 모든 jpg 파일을 불러온 후 cv2의 imread를 사용하여 이미지 리스트를 생성하는 함수임.

  show_image_list 함수는 show 함수를 약간 변형하여 이미지 리스트 내 모든 이미지를 출력하는 함수임.

Data Collection and Preprocessing

Data(Image) Correction

```
def adjust_gamma(image, gamma=1.0): # 감마 보정 함수
    invGamma = 1.0 / gamma
    table = np.array([((i / 255.0) ** invGamma) * 255
        for i in np.arange(0, 256)]).astype("uint8")
    return cv2.LUT(image, table)

def img_correction(img_list): # 이미지 보정
    new_img_list = []
    for im in img_list:
        img = im.copy()
        img = cv2.cvtColor(img, cv2.COLOR_BGR2GRAY)
        img = cv2.resize(img, (380 * 5, 380 * 5), cv2.INTER_LINEAR)
        img = img[20:-20, 20:-20]
        img = cv2.adaptiveThreshold(img, 255,
                                    cv2.ADAPTIVE_THRESH_GAUSSIAN_C,
                                    cv2.THRESH_BINARY,
                                    201,
                                    2) # 적응적 임계처리를 적용
        img = adjust_gamma(img, 0.3)
        new_img_list.append(img)
    return new_img_list
```



- 이미지 보정

  이미지 보정은 grayscale 변환 -> 1900x1900 resize -> 외곽 crop -> 적응형 임계처리 -> 감마보정 순서로 이루어짐.

  Resizing한 이유는 380x380의 작은 이미지를 사용할 경우 ,이후 진행하는 과정인 이미지 외곽 자르기에서 외곽 탐색을 잘 못 수행하기 때문임.

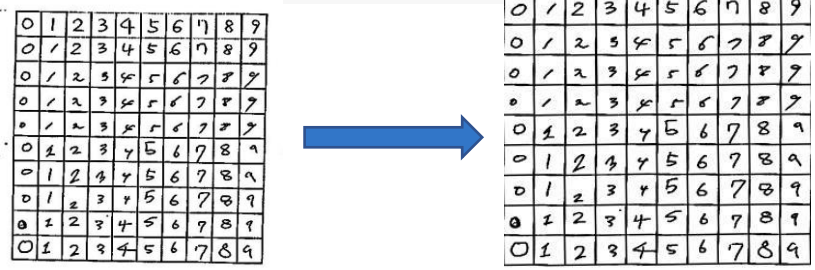
  cv2.adaptiveThreshold 함수를 사용하면, 모든 픽셀에 중심으로부터 거리의 가우시안 가중치를 적용한 threshold를 사용하게 됨.

  마지막으로 adjust_gamma 함수를 사용.

Data Collection and Preprocessing

Data(Image) Crop

```python
def img_crop(img_list): # 이미지 자르기
    new_img_list = []
    for im in img_list:
        img = im.copy()
        img = cv2.bitwise_not(img) # 반전
        contours, _ = cv2.findContours(img, cv2.RETR_EXTERNAL, cv2.CHAIN_APPROX_SIMPLE) # 윤곽선 검출
        for cnt in contours:
            rect = cv2.minAreaRect(cnt) # [0] 중심점, [1] (길이, 높이) [2] 각도
            if rect[1] > (1500, 1500): # 1500x1500 이상의 사각형만
                center = rect[0]
                x, y = rect[1][0], rect[1][1]
                angle = rect[2]
                height, width= img.shape[:2]
                if angle == 0 or angle == 90: # 각 변환
                    pass
                elif angle < 45:
                    matrix = cv2.getRotationMatrix2D((width/2, height/2), angle, 1)
                    img = cv2.warpAffine(img, matrix, (width, height))
                    m = matrix[:, :2]
                    x, y = m @ (x, y)
                else:
                    matrix = cv2.getRotationMatrix2D((width/2, height/2), -(90 - angle), 1)
                    img = cv2.warpAffine(img, matrix, (width, height))
                    m = matrix[:, :2]
                    x, y = m @ (x, y)

                img = img[math.floor(center[0]-x/2):math.floor(center[0]+x/2),
                          math.floor(center[1]-y/2):math.floor(center[1]+y/2)]
                img = cv2.bitwise_not(img) # 반전
                new_img_list.append(img)
    return new_img_list
```



- 이미지 외곽 자르기

  img_crop 함수는 opencv의 외곽선 검출 함수 findcontour를 사용하여 가장 큰 외곽선을 검출하여 그것을 기준으로 crop 하는 함수임.

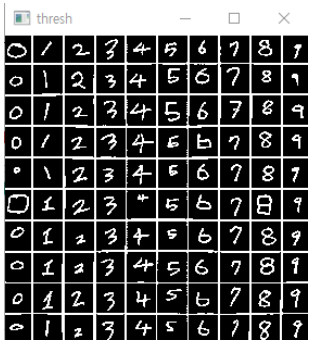
  findcontour 함수는 흰색 선을 찾는 함수이기 때문에 우선 이진화 된 이미지를 반전시켜 찾은 외곽선들을 contours에 저장하고, 각 contour를 minAreaRect에 넣어 픽셀크기가 1500x1500이상인 가장 큰 사각형을 찾음.

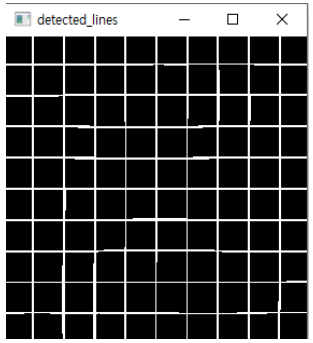
  이후 검출된 사각형크기로 crop 해준 뒤 다시 반전시키면 최외곽이 잘린 이미지를 얻을 수 있다.

Prepare Data

Detect line and remove

```
def img_line_crop(img_list):
    new_img_list = []
    for im in img_list:
        img = im.copy()
        thresh = cv2.threshold(img, 0, 255, cv2.THRESH_BINARY_INV + cv2.THRESH_OTSU)[1]
        horizontal_line = cv2.getStructuringElement(cv2.MORPH_RECT, (18,1))
        vertical_line= cv2.getStructuringElement(cv2.MORPH_RECT, (1, 18))

        detected_lines2 = cv2.morphologyEx(thresh, cv2.MORPH_OPEN, vertical_line, iterations=2)
        detected_lines1 = cv2.morphologyEx(thresh, cv2.MORPH_OPEN, horizontal_line, iterations=2)
        cnts1 = cv2.findContours(detected_lines1, cv2.RETR_EXTERNAL, cv2.CHAIN_APPROX_SIMPLE)
        cnts1 = cnts1[0] if len(cnts1) == 2 else cnts1[1]
        for c in cnts1:
            cv2.drawContours(img, [c], -1, (255,255,255), 2)
        cnts2 = cv2.findContours(detected_lines2, cv2.RETR_EXTERNAL, cv2.CHAIN_APPROX_SIMPLE)
        cnts2 = cnts2[0] if len(cnts2) == 2 else cnts2[1]
        for a in cnts2:
            cv2.drawContours(img, [a], -1, (255,255,255), 2)
        new_img_list.append(img)

    return new_img_list
```



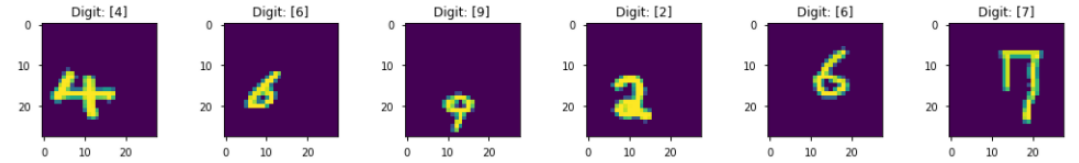


- 첫 번째로 OpenCV와 Thresh_OTSU를 사용하여 이미지의 임계값을 Binary로 나누고 사진의 흑과 백의 경계선을 뚜렷하게 만듦.
- 이후 getStructuringElement를 통해 외곽선을 검출함.
- 마지막으로 Contours를 사용하여 식별된 detected linde을 제거하여 숫자만 보이는 최종 이미지를 얻음.

Prepare Data

Remodification

```
f = plt.figure(figsize=(18,2))
ax = f.subplots(1, 6)
np.random.seed(42)
for i, n in enumerate(np.random.randint(0, len(samples), size=6)):
    m = (np.random.randint(0, len(samples[n])))
    ax[i].imshow(samples[n][m])
    ax[i].set_title(f'Digit: [{m}]')
plt.show()
```

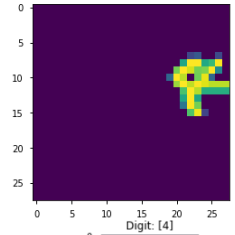



```
def cut_img(img_list):
    SIZE = int(img_list[0].shape[0] / 10)
    samples = [] #array to store cut images
    width, heigth = img_list[0].shape
    for i in range(len(img_list)):
        for x in range(0, width, SIZE):
            cuts = []
            for y in range(0, heigth, SIZE):
                cut = img_list[i][x:x+SIZE, y:y+SIZE]
                cut = cv2.bitwise_not(cut)
                cut = cv2.threshold(cut, 50, 255, cv2.THRESH_TOZERO)[1]
                cuts.append(cut)
            samples.append(cuts)
    print(f'Cut {len(samples)*len(samples[0])} images total.')
    return samples
```

```
samples = cut_img(img_line_croped)
```

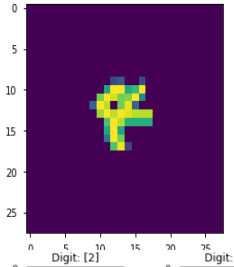
```
Cut 4200 images total.
```

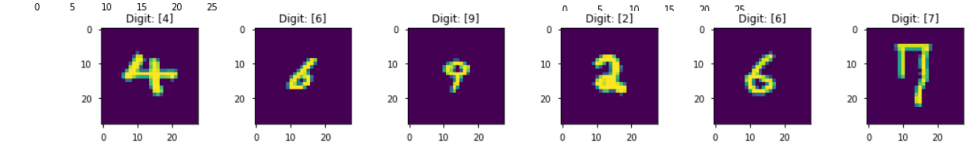
```
show(samples[219][4])
```



```
def resize_and_center(sample, new_size=28):
    background = Image.new(sample.mode, sample.size, sample.getpixel((0, 0)))
    diff = ImageChops.difference(sample, background)
    diff = ImageChops.add(diff, diff, 2.0, -35)
    bbox = diff.getbbox()
    crop = sample.crop(bbox)
    delta_w = new_size - crop.size[0]
    delta_h = new_size - crop.size[1]
    padding = (delta_w//2, delta_h//2, delta_w-(delta_w//2), delta_h-(delta_h//2))
    return ImageOps.expand(crop, padding)

resized_samples = []
for row in samples1:
    resized_samples.append([resize_and_center(sample) for sample in row])
```

```
show(resized_samples[219][4])
```





- 여기서 나온 것을 /10으로 해서 각 칸들을 자름
- 그렇게 된다면 데이터(숫자)의 위치가 모두 다르다는 문제가 생겨 다음의 방법으로 해결함.

1. 자른 사진의 배경과 선을 구별하여 box를 가짐.
2. 그 부분만을 자른 다음에 가운데로 모아 다시 나열함.

- 그 결과 사진에서 보이는 것과 같이 대부분의 데이터가 의도한 대로 가공됨.

Dataset

다. Dataset 분류:

Existing + New Data

```
# 30000개 균등하게 뽑아오기
X_c = []
y_c = []
for i in range(10):
    index = np.where(y == i)
    X_c.append(X[index][:3000])
    y_c.append(y[index][:3000])
X_c = np.reshape(X_c, (-1, 784))
y_c = np.reshape(y_c, -1)

print(X_c.shape, X_c.dtype)
print(y_c.shape, y_c.dtype)
```

```
(30000, 784) float64
(30000,) uint8
```

```
# test 10%
X_new = np.concatenate((X_c, data), axis=0)
y_new = np.concatenate((y_c, target), axis=0)
print(X_new.shape, X_new.dtype)
print(y_new.shape, y_new.dtype)
```

```
(34200, 784) float64
(34200,) uint8
```

```
def count_exemple_per_digit(exemples): # 데이터 분포도 확인 함수
    hist = np.zeros(10)

    for y in exemples:
        hist[y] += 1 # np.where(y == 1)

    colors = []
    for i in range(10):
        colors.append(plt.get_cmap('viridis')(np.random.uniform(0.0,1.0,1)[0]

    bar = plt.bar(np.arange(10), hist, 0.8, color=colors)

    plt.grid()
    plt.show()
```

- MNIST 데이터 30,000개 추출
  기존 데이터에 신규 데이터를 더해서 총 34,200개의 인스턴스를 포함한 데이터셋을 완성시킴.

또한, 데이터셋 별로 레이블이 균등하게 들어가 있는지 확인하기 위해 데이터 분포도 확인 함수를 정의함.

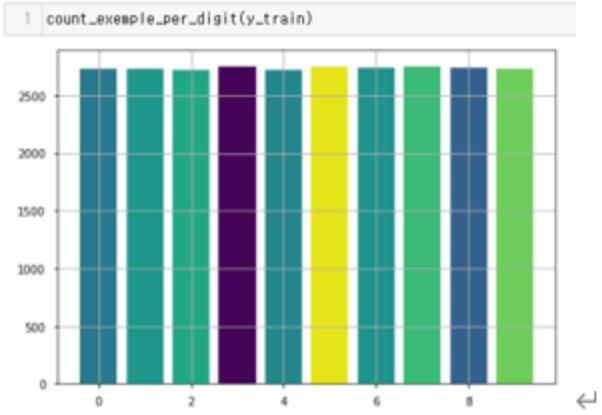
Dataset

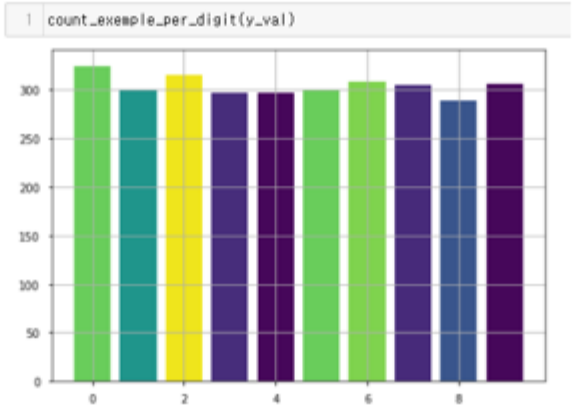
Existing + New Data

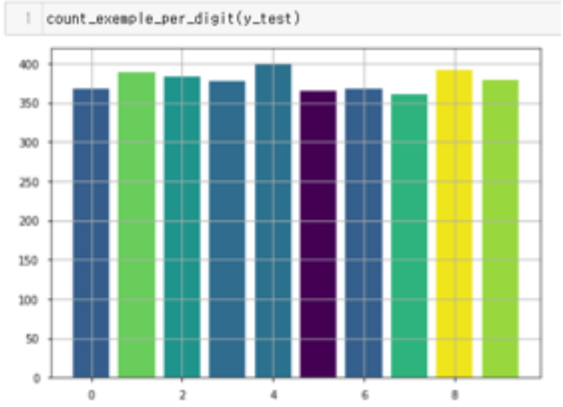
```
# train/val/test 대략 8:1:1
from sklearn.model_selection import train_test_split # random_state 26
X_train_val, X_test, y_train_val, y_test = train_test_split(X_new[:-400], y_new[:-400],
                                                            test_size=0.1, shuffle=True, random_state=42)
X_train, X_val, y_train, y_val = train_test_split(X_train_val, y_train_val,
                                                  test_size=0.1, shuffle=True, random_state=42)

X_test = np.concatenate((X_test, X_new[-400:]), axis=0)
y_test = np.concatenate((y_test, y_new[-400:]), axis=0)
for sets in (X_train, y_train, X_val, y_val, X_test, y_test):
    np.random.seed(42)
    np.random.shuffle(sets)
```

시험 데이터에 신규 데이터 400개를 미리 할당한 후, 훈련 데이터와 검증 데이터 그리고 시험 데이터를 분할함.

그리고 각 데이터셋 별로 셔플을 진행.







Select Model

라. 학습에 사용할 모델 선택과 최적화:

SGD classifier

```
In [35]: from sklearn.linear_model import SGDClassifier

         sgd_clf = SGDClassifier(random_state=42)
         sgd_clf.fit(X_train, y_train)

Out[35]: SGDClassifier(random_state=42)
```

```
In [36]: print(f'{sgd_clf.__class__.__name__}, {sgd_clf.score(X_val, y_val):0.4f}')

         SGDClassifier, 0.7876
```

```
In [37]: from sklearn.metrics import confusion_matrix
         from sklearn.metrics import classification_report
         from sklearn.metrics import accuracy_score
```

```
In [38]: pred_1 = sgd_clf.predict(X_val)
         SGD_cm = confusion_matrix(y_val, pred_1)
         print(SGD_cm)
         print(classification_report(y_val,pred_1))
```

```
[[281   0   2   2  10   2   3   7  12   6]
 [  1 270   4   4   0   4   1   2  12   1]
 [  4   8 222   9   7  10  15  11  28   1]
 [ 14   3   4 215   5  23   3  18  10   2]
 [  4   0   2   1 228   9   8  12   7  26]
 [  8   5   5  12   6 234   6   6  13   6]
 [ 10   3   6   8   3   6 264   2   1   5]
 [  2   0   2   4   7   0   0 276   3  11]
 [  8   5   5   7   7  20  11   9 199  18]
 [  0   2   3   4  20  10   1  52   7 207]]
              precision    recall  f1-score   support

           0       0.85      0.86      0.86       325
           1       0.91      0.90      0.91       299
           2       0.87      0.70      0.78       315
           3       0.81      0.72      0.76       297
           4       0.78      0.77      0.77       297
           5       0.74      0.78      0.76       301
           6       0.85      0.86      0.85       308
           7       0.70      0.90      0.79       305
           8       0.68      0.69      0.69       289
           9       0.73      0.68      0.70       306

    accuracy                           0.79      3042
   macro avg       0.79      0.79      0.79      3042
weighted avg       0.79      0.79      0.79      3042
```

- SGD classifier는 매 스텝에서 한 개의 샘플을 무작위로 선택하고 그 하나의 샘플에 대한 gradient를 계산하는 알고리즘임.

- SGD classifier 학습 결과, Score는 0.7876정도로 다소 낮은 결과가 보여져, SGD classifier 모델은 최종 선별에 사용하지 않음.

Select Model

Softmax regression

```
In [39]: from sklearn.linear_model import LogisticRegression
softmax_reg=LogisticRegression(random_state=42)
softmax_reg.fit(X_train, y_train)
```

```
C:₩Users₩cd081₩anaconda3₩lib₩site-packages₩sklearn₩linear_model₩_logistic.py:763: ConvergenceWarning: lbfgs failed to converge (statu
s=1):
STOP: TOTAL NO. of ITERATIONS REACHED LIMIT.

Increase the number of iterations (max_iter) or scale the data as shown in:
    https://scikit-learn.org/stable/modules/preprocessing.html
Please also refer to the documentation for alternative solver options:
    https://scikit-learn.org/stable/modules/linear_model.html#logistic-regression
  n_iter_i = _check_optimize_result(
```

```
Out[39]: LogisticRegression(random_state=42)
```

```
In [40]: print(f'{softmax_reg.__class__.__name__}, {softmax_reg.score(X_val, y_val):0.4f}')
```

```
LogisticRegression, 0.8429
```

```
In [41]: pred_1 = softmax_reg.predict(X_val)
Softmax_cm = confusion_matrix(y_val, pred_1)
print(Softmax_cm)
print(classification_report(y_val,pred_1))
```

```
[[283   0   8   2  13   4   3   2   5   5]
 [  1 279   3   5   0   3   1   1   5   1]
 [  0   8 254   8   6   6   6   5  21   1]
 [  2   8   7 243   3  14   1   7   9   3]
 [  1   1   1   2 247   6   6   3   8  22]
 [  8   5   5  12   9 236   7   2  15   2]
 [  1   7   5   3   6   6 276   1   3   0]
 [  1   1   4   6   5   1   0 277   1   9]
 [  1   8   8   6   8  26   2   1 217  12]
 [  0   4   0   7  12   6   2  14   9 252]]
              precision    recall  f1-score   support

           0       0.95      0.87      0.91       325
           1       0.87      0.93      0.90       299
           2       0.86      0.81      0.83       315
           3       0.83      0.82      0.82       297
           4       0.80      0.83      0.82       297
           5       0.77      0.78      0.78       301
           6       0.91      0.90      0.90       308
           7       0.88      0.91      0.90       305
           8       0.74      0.75      0.75       289
           9       0.82      0.82      0.82       306

    accuracy                           0.84      3042
   macro avg       0.84      0.84      0.84      3042
weighted avg       0.84      0.84      0.84      3042
```

- Softmax regression은 3개 이상의 선택지 중 1개를 고르는 다중 클래스 분류 문제를 위한 알고리즘.
- Softmax regression 알고리즘은 다중 클래스에 대한 선형 회귀 모델과 유사하므로 이미지 분류에 사용하기에는 다소 단순한 구조라 판단됨.
- 또한 결과적으로 score가 0.8429정도로 낮게 측정되어 적합하지 않은 모델이라 생각되어 최종모델선택 시 고려하지 않음.

Select Model

DecisionTree Classifier

```
In [42]: # DT
         from sklearn.tree import DecisionTreeClassifier
         tree_clf = DecisionTreeClassifier(random_state=42)
         tree_clf.fit(X_train, y_train)
```

```
Out[42]: DecisionTreeClassifier(random_state=42)
```

```
In [43]: print(f'{tree_clf.__class__.__name__}, {tree_clf.score(X_val, y_val):0.4f}')
```

```
DecisionTreeClassifier, 0.7965
```

```
In [44]: pred_1 = tree_clf.predict(X_val)
         dt_cm = confusion_matrix(y_val, pred_1)
         print(dt_cm)
         print(classification_report(y_val,pred_1))
```

```
[[283   1   3   5   5   9  10   1   3   5]
 [  1 278   1   7   2   3   1   1   2   3]
 [  6  10 238   6   9   6   7  13  10  10]
 [  4   5  11 234   3  17   1   7  10   5]
 [  2   6   4   3 229   9   6   3  15  20]
 [  9   4   5  24   6 222  14   1   9   7]
 [  6   5   9   7   6  10 259   0   4   2]
 [  5   0  10   3   5   6   0 262   2  12]
 [  6   5  10   9  15  11   6   4 205  18]
 [  2   5   6   9  32   9   0  17  13 213]]
              precision    recall  f1-score   support

           0       0.87      0.87      0.87       325
           1       0.87      0.93      0.90       299
           2       0.80      0.76      0.78       315
           3       0.76      0.79      0.77       297
           4       0.73      0.77      0.75       297
           5       0.74      0.74      0.74       301
           6       0.85      0.84      0.85       308
           7       0.85      0.86      0.85       305
           8       0.75      0.71      0.73       289
           9       0.72      0.70      0.71       306

    accuracy                           0.80      3042
   macro avg       0.80      0.80      0.80      3042
weighted avg       0.80      0.80      0.80      3042
```

- 결정 트리(DecisionTree Classifier)는 분류와 회귀 작업 그리고 다중출력 작업도 가능한 머신러닝 알고리즘임.

- DecisionTree Classifier를 통해 학습을 진행 한 결과 0.7965 정도의 성능으로 부정적인 결과가 확인되어 최종 결정에서 제외하기로 함.

Select Model

KNeigborsClassifier

```
In [45]: from sklearn.neighbors import KNeighborsClassifier
         knn_clf = KNeighborsClassifier(n_jobs=-1) # 적절한 k값
         knn_clf.fit(X_train,y_train)

Out[45]: KNeighborsClassifier(n_jobs=-1)

In [46]: print(f'{knn_clf.__class__.__name__}, {knn_clf.score(X_val, y_val):0.4f}')

         KNeighborsClassifier, 0.9402
```

```
In [47]: pred_1 = knn_clf.predict(X_val)
         knn_cm = confusion_matrix(y_val, pred_1)
         print(knn_cm)
         print(classification_report(y_val,pred_1))

[[317   0   0   1   1   1   3   0   0   2]
 [  0 297   0   0   0   0   0   1   0   1]
 [  1   5 295   2   0   0   4   7   1   0]
 [  0   4   7 272   2   4   0   2   4   2]
 [  2   6   1   0 277   0   0   1   0  10]
 [  0   0   0   9   0 281   3   1   3   4]
 [  2   3   0   0   0   4 298   0   1   0]
 [  1   3   1   0   2   0   0 290   0   8]
 [  0   2   1  10   1  14   2   0 249  10]
 [  1   1   0   3   6   0   1   9   1 284]]
              precision    recall  f1-score   support

           0       0.98      0.98      0.98       325
           1       0.93      0.99      0.96       299
           2       0.97      0.94      0.95       315
           3       0.92      0.92      0.92       297
           4       0.96      0.93      0.95       297
           5       0.92      0.93      0.93       301
           6       0.96      0.97      0.96       308
           7       0.93      0.95      0.94       305
           8       0.96      0.86      0.91       289
           9       0.88      0.93      0.91       306

    accuracy                           0.94      3042
   macro avg       0.94      0.94      0.94      3042
weighted avg       0.94      0.94      0.94      3042
```

- KNN은 k개의 가장 가까운 이웃을 찾아, 이웃이 갖는 범주로 예측을 수행하는 모델로 추천 시스템과 인공지능 이미지 인식 시스템의 기본이 되는 알고리즘임.

- 테스트 데이터 검증 결과 정확도 0.9402의 높은 성능을 보여 Kneighbor Classifier는 최종 모델 선택에서 사용하기로 함.

Select Model

KNeigborsClassifier

- Model Hyperparameter 선정

In [48]:
```
from sklearn.model_selection import GridSearchCV

knn_clf = KNeighborsClassifier(n_jobs=-1)
params = {'n_neighbors' : list(range(2, 10)),
          'weights' : ["uniform", "distance"],
          'metric' : ['euclidean', 'minkowski']
         }

knn_grid = GridSearchCV(knn_clf, param_grid=params, cv=5, n_jobs=-1)
%time knn_grid.fit(X_train[:2000], y_train[:2000])
```

```
Wall time: 3.41 s
```

Out[48]:
```
GridSearchCV(cv=5, estimator=KNeighborsClassifier(n_jobs=-1), n_jobs=-1,
             param_grid={'metric': ['euclidean', 'minkowski'],
                         'n_neighbors': [2, 3, 4, 5, 6, 7, 8, 9],
                         'weights': ['uniform', 'distance']})
```

In [49]:
```
%time knn_grid.best_estimator_.fit(X_train, y_train) # 학습시간
print(f'{knn_grid.best_params_} {knn_grid.best_estimator_.score(X_val, y_val):0.4f}')
```

```
Wall time: 31 ms
{'metric': 'euclidean', 'n_neighbors': 4, 'weights': 'distance'} 0.9481
```

- GridSearchCV로 최적 하이퍼 파라미터를 찾은 결과, {'metric' : 'euclidean', 'n_neighbors' : 4, 'weights' : 'distance'}의 파라미터 값을 얻음.

- 위 결과를 이용해 training을 진행한 결과, 하이퍼파라미터 조정 전 0.9402이었던 score가 0.9481로 개선됨.

Select Model

MLPClassifier

```
In [53]: from sklearn.neural_network import MLPClassifier
         mlp_clf = MLPClassifier(random_state=42)
         mlp_clf.fit(X_train,y_train)

Out[53]: MLPClassifier(random_state=42)
```

```
In [54]: print(f'{mlp_clf.__class__.__name__}, {mlp_clf.score(X_val, y_val):0.4f}')

         MLPClassifier, 0.9300
```

```
In [55]: pred_1 = mlp_clf.predict(X_val)
         mlp_cm = confusion_matrix(y_val, pred_1)
         print(mlp_cm)
         print(classification_report(y_val,pred_1))
```

```
[[309   0   0   2   1   0   0   1   7   5]
 [  1 290   1   0   0   1   1   1   2   2]
 [  2   2 288   8   1   0   2   5   5   2]
 [  1   3   3 271   2   6   1   2   5   3]
 [  2   0   2   1 271   0   3   3   6   9]
 [  0   2   1   4   0 282   3   1   6   2]
 [  1   0   0   0   1   7 292   1   5   1]
 [  1   0   3   1   2   2   0 288   0   8]
 [  1   1   1   7   0   9   5   0 260   5]
 [  0   1   0   5  12   1   1   4   4 278]]
              precision    recall  f1-score   support

           0       0.97      0.95      0.96       325
           1       0.97      0.97      0.97       299
           2       0.96      0.91      0.94       315
           3       0.91      0.91      0.91       297
           4       0.93      0.91      0.92       297
           5       0.92      0.94      0.93       301
           6       0.95      0.95      0.95       308
           7       0.94      0.94      0.94       305
           8       0.87      0.90      0.88       289
           9       0.88      0.91      0.90       306

    accuracy                           0.93      3042
   macro avg       0.93      0.93      0.93      3042
weighted avg       0.93      0.93      0.93      3042
```

- 다층 퍼셉트론은 퍼셉트론을 여러 층으로 순차적으로 쌓아 이은 구조의 모델임.
- 주요 하이퍼 파라미터로는 적응적 학습률 알고리즘을 결정하는 'solver', 은닉층의 크기를 결정하는 'hidden_layer_size'등이 있음.
- 테스트 데이터 검증 결과 정확도 0.944의 높은 성능을 보여 MLP Classifier 또한 최종 모델 선택에서 사용함.

Select Model

MLPClassifier

- Model Hyperparameter 선정

In [56]:
```
from sklearn.model_selection import GridSearchCV

mlp_clf = MLPClassifier(random_state=42)
params = {'hidden_layer_sizes' : list(range(300, 400, 10))
         }

mlp_grid = GridSearchCV(mlp_clf, param_grid=params, cv=5, n_jobs=-1)
%time mlp_grid.fit(X_train[:2000], y_train[:2000])
```

```
Wall time: 1min 11s
```

Out[56]:
```
GridSearchCV(cv=5, estimator=MLPClassifier(random_state=42), n_jobs=-1,
             param_grid={'hidden_layer_sizes': [300, 310, 320, 330, 340, 350,
                                                360, 370, 380, 390]})
```

In [57]:
```
%time mlp_grid.best_estimator_.fit(X_train, y_train) # 학습시간
print(f'{mlp_grid.best_params_} {mlp_grid.best_estimator_.score(X_val, y_val):0.4f}')
```

```
Wall time: 1min 33s
{'hidden_layer_sizes': 370} 0.9487
```

- GridSearchCV를 통해 최적의 하이퍼파라미터를 도출해낸 결과 {'hidden_layer_sizes' : 370}을 얻음.
- 위 결과를 토대로 MLPClassifier 모델로 다시 학습한 결과 socre가 0.9300에서 0.9487로 개선됨.

Select Model

SVC(kernel = 'poly', degree=2)

```
In [50]: from sklearn.svm import SVC
         svm_clf = SVC(kernel="poly", degree=2, probability=True, random_state=42)
         svm_clf.fit(X_train,y_train)

Out[50]: SVC(degree=2, kernel='poly', probability=True, random_state=42)

In [51]: print(f'{svm_clf.__class__.__name__}, {svm_clf.score(X_val, y_val):0.4f}')

         SVC, 0.9421

In [58]: pred_1 = svm_clf.predict(X_val)
         svc_cm = confusion_matrix(y_val, pred_1)
         print(svc_cm)
         print(classification_report(y_val,pred_1))

[[310   1   3   1   1   0   3   0   3   3]
 [  0 297   1   0   0   0   0   0   0   1]
 [  0   3 295   0   1   0   3   7   6   0]
 [  1   4   6 276   1   1   0   2   4   2]
 [  1   0   1   1 280   2   1   1   1   9]
 [  0   0   0   8   1 286   1   1   4   0]
 [  2   5   2   1   1   4 290   0   3   0]
 [  1   4   2   0   3   0   0 289   1   5]
 [  0   3   2   4   4   6   0   1 265   4]
 [  1   1   0   3  11   1   2   5   4 278]]
              precision    recall  f1-score   support

           0       0.98      0.95      0.97       325
           1       0.93      0.99      0.96       299
           2       0.95      0.94      0.94       315
           3       0.94      0.93      0.93       297
           4       0.92      0.94      0.93       297
           5       0.95      0.95      0.95       301
           6       0.97      0.94      0.95       308
           7       0.94      0.95      0.95       305
           8       0.91      0.92      0.91       289
           9       0.92      0.91      0.91       306

    accuracy                           0.94      3042
   macro avg       0.94      0.94      0.94      3042
weighted avg       0.94      0.94      0.94      3042
```

- SVM은 각 그룹이 최대로 떨어질 수 있는 최대 마진을 찾고, 최적 초평면을 기준으로 그룹을 구분짓는 기법임.
- SVC의 kernel 함수를 사용하여 보다 복잡하고 강력한 다항 SVC 모델을 구축함.
- 테스트 데이터 검증 결과 정확도 0.9421의 높은 성능을 보임.

Select Model

SVC(kernel = 'poly', degree=2)

- Model Hyperparameter 선정

In [59]:
```
from sklearn.preprocessing import StandardScaler
from sklearn.model_selection import RandomizedSearchCV
from scipy.stats import uniform
from sklearn.pipeline import Pipeline

svc_clf = SVC(probability=True, random_state=42)
svc_pipeline = Pipeline([
    ('scaler', StandardScaler()),
    ('svc_clf', svc_clf)
])
params = {'svc_clf__C': uniform(5,15),
          'svc_clf__gamma': uniform(0.0005,0.0015)
         }

svc_rnd = RandomizedSearchCV(svc_pipeline, param_distributions=params, n_iter=200, cv=5, n_jobs=-1, random_state=42)
%time svc_rnd.fit(X_train[:2000], y_train[:2000])
```

```
Wall time: 6min 1s
```

Out[59]:
```
RandomizedSearchCV(cv=5,
                   estimator=Pipeline(steps=[('scaler', StandardScaler()),
                                             ('svc_clf',
                                              SVC(probability=True,
                                                  random_state=42))]),
                   n_iter=200, n_jobs=-1,
                   param_distributions={'svc_clf__C': <scipy.stats._distn_infrastructure.rv_frozen object at 0x00000293D7890040>,
                                        'svc_clf__gamma': <scipy.stats._distn_infrastructure.rv_frozen object at 0x00000293D87F5220
>},
                   random_state=42)
```

- RandomizedSerchCV를 이용하여 'svc_clf__C' 와 'svc_clf__gamma'의 최적 값을 찾아 모델에 적용시켜 학습한 결과, score가 0.9421에서 0.9454로 개선됨

Select Model

Extratrees Classifier

```
In [61]: from sklearn.ensemble import ExtraTreesClassifier
         etc_clf = ExtraTreesClassifier(n_jobs=-1, random_state=42)
         etc_clf.fit(X_train,y_train)

Out[61]: ExtraTreesClassifier(n_jobs=-1, random_state=42)

In [62]: print(f'{etc_clf.__class__.__name__}, {etc_clf.score(X_val, y_val):0.4f}')

         ExtraTreesClassifier, 0.9444

In [63]: pred_1 = etc_clf.predict(X_val)
         etc_cm = confusion_matrix(y_val, pred_1)
         print(etc_cm)
         print(classification_report(y_val,pred_1))

[[313   0   1   0   1   2   4   0   4   0]
 [  1 295   0   0   0   0   1   0   1   1]
 [  0   2 296   1   0   0   4   6   5   1]
 [  1   3   7 271   2   4   0   2   7   0]
 [  1   0   0   0 284   0   1   0   2   9]
 [  0   1   0   6   1 285   3   0   3   2]
 [  2   2   0   0   1   4 296   0   2   1]
 [  0   0   5   1   1   0   0 290   0   8]
 [  0   2   3   4   1   3   2   1 263  10]
 [  1   1   1   7   9   1   1   3   2 280]]
              precision    recall  f1-score   support

           0       0.98      0.96      0.97       325
           1       0.96      0.99      0.98       299
           2       0.95      0.94      0.94       315
           3       0.93      0.91      0.92       297
           4       0.95      0.96      0.95       297
           5       0.95      0.95      0.95       301
           6       0.95      0.96      0.95       308
           7       0.96      0.95      0.96       305
           8       0.91      0.91      0.91       289
           9       0.90      0.92      0.91       306

    accuracy                           0.94      3042
   macro avg       0.94      0.94      0.94      3042
weighted avg       0.94      0.94      0.94      3042
```

- 사이킷런에서는 RandomForestClassifier와 동시에 ExtratreesClassifier모델도 지원함.
- Extratrees Classifier 모델은 트리를 더욱 무작위로 만들기 위해, 일반적인 결정트리모델처럼 최적의 임계값을 찾는 대신 후보 변수를 사용해 무작위로 분할해 그 중 최상의 분할을 선택함.
- Extratrees Classifier를 이용해 테스트 데이터를 검증해본 결과 0.9444의 높은 score를 보임

Select Model

Extratrees Classifier

- Model Hyperparameter 선정

In [64]:
```
from sklearn.model_selection import GridSearchCV

etc_clf = ExtraTreesClassifier(n_jobs=-1, random_state=42)
params = {'max_depth': [None, 12, 13, 14, 15, 16],
          'n_estimators': [100, 500, 700, 1000]
         }
ext_grid = GridSearchCV(etc_clf, param_grid=params, cv=5, n_jobs=-1)
%time ext_grid.fit(X_train[:2000], y_train[:2000])
```

```
Wall time: 49.7 s
```

Out[64]:
```
GridSearchCV(cv=5, estimator=ExtraTreesClassifier(n_jobs=-1, random_state=42),
             n_jobs=-1,
             param_grid={'max_depth': [None, 12, 13, 14, 15, 16],
                         'n_estimators': [100, 500, 700, 1000]})
```

In [65]:
```
%time ext_grid.best_estimator_.fit(X_train, y_train) # 학습시간
print(f'{ext_grid.best_params_} {ext_grid.best_estimator_.score(X_val, y_val):0.4f}')
```

```
Wall time: 23.8 s
{'max_depth': None, 'n_estimators': 1000} 0.9458
```

- GridSearchCV를 이용하여 최적의 하이퍼파라미터 값, {'max_depth' : None, 'n_estimators' : 1000} 을 얻음.

- 이를 ExtratreesClassifier 모델에 적용시켜 본 결과, 이전에 0.9444였던 성능이 0.9458까지 개선된 것을 볼 수 있음

Select Model

RandomForest Classifier

In [71]:
```
# RF
from sklearn.ensemble import RandomForestClassifier

rfc_clf = RandomForestClassifier(n_jobs=-1, random_state=42)
rfc_clf.fit(X_train, y_train)
```

Out[71]:
```
RandomForestClassifier(n_jobs=-1, random_state=42)
```

In [72]:
```
print(f'{rfc_clf.__class__.__name__}, {rfc_clf.score(X_val, y_val):0.4f}')
```

```
RandomForestClassifier, 0.9379
```

In [73]:
```
pred_1 = rfc_clf.predict(X_val)
rfc_cm = confusion_matrix(y_val, pred_1)
print(rfc_cm)
print(classification_report(y_val,pred_1))
```

```
[[313   0   1   0   0   3   5   0   3   0]
 [  1 293   1   1   0   0   1   0   1   1]
 [  2   2 294   2   1   0   4   6   3   1]
 [  1   3   6 271   1   3   1   4   7   0]
 [  2   0   1   1 281   0   1   0   2   9]
 [  0   0   0   9   0 283   4   0   2   3]
 [  2   2   1   0   2   7 290   0   3   1]
 [  0   0   4   1   2   0   0 287   1  10]
 [  0   2   0   6   2   3   3   1 261  11]
 [  1   1   1   6   8   0   2   5   2 280]]
              precision    recall  f1-score   support

           0       0.97      0.96      0.97       325
           1       0.97      0.98      0.97       299
           2       0.95      0.93      0.94       315
           3       0.91      0.91      0.91       297
           4       0.95      0.95      0.95       297
           5       0.95      0.94      0.94       301
           6       0.93      0.94      0.94       308
           7       0.95      0.94      0.94       305
           8       0.92      0.90      0.91       289
           9       0.89      0.92      0.90       306

    accuracy                           0.94      3042
   macro avg       0.94      0.94      0.94      3042
weighted avg       0.94      0.94      0.94      3042
```

- 랜덤 포레스트는 훈련 과정에서 구성한 다수의 결정 트리로부터 레이블 또는 평균 값을 결과값으로 출력하는 모델임.
- 일반적으로 결정 트리를 이용한 방법의 경우 그 결과 또는 성능의 변동 폭이 큼. 이러한 단점을 앙상블 모델을 통해 극복이 가능함.
- RandomForest Clasifier를 이용한 테스트 데이터 검증 결과 정확도 0.9379의 높은 성능을 보여, 최종 모델 선택에서 사용하기로 함.

Select Model

RandomForest Classifier

- Model Hyperparameter 선정

In [80]:
```
from sklearn.model_selection import GridSearchCV

rfc_clf = RandomForestClassifier(n_jobs=-1, random_state=42)
params = {'max_depth': [None, 14, 15, 16, 17, 18],
          'n_estimators': [800, 1000, 1200]
         }
rfc_grid = GridSearchCV(rfc_clf, param_grid=params, cv=5, n_jobs=-1)
%time rfc_grid.fit(X_train[:5000], y_train[:5000])
```

```
Wall time: 2min 54s
```

Out[80]:
```
GridSearchCV(cv=5, estimator=RandomForestClassifier(n_jobs=-1, random_state=42),
             n_jobs=-1,
             param_grid={'max_depth': [None, 14, 15, 16, 17, 18],
                         'n_estimators': [800, 1000, 1200]})
```

In [81]:
```
%time rfc_grid.best_estimator_.fit(X_train, y_train) # 학습시간
print(f'{rfc_grid.best_params_} {rfc_grid.best_estimator_.score(X_val, y_val):0.4f}')
```

```
Wall time: 22.4 s
{'max_depth': None, 'n_estimators': 1200} 0.9454
```

- GridSerchCV를 통해 최적의 파라미터 값 {'max_depth' : None, 'n_estimators': 1200}을 얻음.

- 이를 이용하여RandomForest Classifier 모델을 통해 학습한 결과 score가 0.9379에서 0.9454로 개선됨.

Voting Classifier

```
from sklearn.neighbors import KNeighborsClassifier
from sklearn.neural_network import MLPClassifier
from sklearn.svm import SVC
from sklearn.ensemble import ExtraTreesClassifier
from sklearn.ensemble import RandomForestClassifier
```

```
knn_clf = knn_grid.best_estimator_ # 0.9481
mlp_clf = mlp_grid.best_estimator_ # 0.9487
svc_clf = svc_rnd.best_estimator_ # 0.9454
etc_clf = ext_grid.best_estimator_ # 0.9458
rfc_clf = rfc_grid.best_estimator_ # 0.9454
```

```
named_estimators = [("knn", knn_clf), ("mlp", mlp_clf), ("svc", svm_clf),
                    ("etc", etc_clf), ("rfc", rfc_clf)]
```

학습시간
knn_clf 31ms
mlp_clf 1min 33s
svc_clf 9min 13s
etc_clf 23.8s
rfc_clf 22.4s

```
estimators = [knn_clf, mlp_clf, svm_clf, etc_clf, rfc_clf]
for estimator in estimators:
    print(f'{estimator.get_params}')
```

```
<bound method BaseEstimator.get_params of KNeighborsClassifier(metric='euclidea
n', n_jobs=-1, n_neighbors=4,
                     weights='distance')>
<bound method BaseEstimator.get_params of MLPClassifier(hidden_layer_sizes=370,
random_state=42)>
<bound method BaseEstimator.get_params of SVC(degree=2, kernel='poly', probabili
ty=True, random_state=42)>
<bound method BaseEstimator.get_params of ExtraTreesClassifier(n_estimators=100
0, n_jobs=-1, random_state=42)>
<bound method BaseEstimator.get_params of RandomForestClassifier(n_estimators=12
00, n_jobs=-1, random_state=42)>
```

best parameter
knn_clf metric='euclidean', n_jobs=-1, n_neighbors=4, weights='distance'
mlp_clf hidden_layer_sizes=370, random_state=42
svm_clf degree=2, kernel='poly', probability=True, random_state=42
etc_clf n_estimators=1000, n_jobs=-1, random_state=42
rfc_clf n_estimators=1200, n_jobs=-1, random_state=42

마. 모델 최적화 및 분석:

- 각 독립 모델 별 최적 하이퍼 파라미터

GridSearch , RandomizedSearch 결과 활용.

- 각 독립 모델 별 예측성능 :

```
knn_clf = knn_grid.best_estimator_
# 0.9481
mlp_clf = mlp_grid.best_estimator_
# 0.9487
svc_clf = svc_rnd.best_estimator_
# 0.9454
etc_clf = ext_grid.best_estimator_
# 0.9458
rfc_clf = rfc_grid.best_estimator_
# 0.9454
```

- 각 독립 모델 별 학습시간 :

Knn_clf : 31ms

Mlp_clf : 1min 33s

SVC_clf : 9min 13s

Etc_clf : 23.8s

Rfc_clf : 22.4s

Voting Classifier

```
from sklearn.ensemble import VotingClassifier

vot_clf = VotingClassifier(named_estimators, n_jobs=-1)
%time vot_clf.fit(X_train, y_train)
```

```
Wall time: 7min 49s

VotingClassifier(estimators=[('knn',
                              KNeighborsClassifier(metric='euclidean',
                                                   n_jobs=-1, n_neighbors=4,
                                                   weights='distance')),
                             ('mlp',
                              MLPClassifier(hidden_layer_sizes=370,
                                            random_state=42)),
                             ('svc',
                              SVC(degree=2, kernel='poly', probability=True,
                                  random_state=42)),
                             ('etc',
                              ExtraTreesClassifier(n_estimators=1000, n_jobs=-1,
                                                   random_state=42)),
                             ('rfc',
                              RandomForestClassifier(n_estimators=1200,
                                                     n_jobs=-1,
                                                     random_state=42))],
                 n_jobs=-1)
```

```
estimators = [knn_clf, mlp_clf, svm_clf, etc_clf, rfc_clf]
for estimator in estimators:
    print(f'{estimator.__class__.__name__}, {estimator.score(X_val, y_val):0.4f}'
```

```
KNeighborsClassifier, 0.9481
MLPClassifier, 0.9487
SVC, 0.9421
ExtraTreesClassifier, 0.9458
RandomForestClassifier, 0.9454
```

hard vs soft

```
# hard
print(f'{vot_clf.__class__.__name__},{vot_clf.voting}: {vot_clf.score(X_val, y_va
# sort
vot_clf.voting = "soft"
print(f'{vot_clf.__class__.__name__},{vot_clf.voting}: {vot_clf.score(X_val, y_va
```

```
VotingClassifier,hard: 0.9550
VotingClassifier,soft: 0.9645
```

- Voting Classifier 모델 학습시간 : 7min 49s

- 각 독립 모델 별 정확도:

Knn_clf : 0.9481

Mlp_clf : 0.9487

SVC_clf :0.9421

Etc_clf : 0.9458

Rfc_clf : 0.9454

- Voting Classifier 하드 보팅, 소프트 보팅 결과:

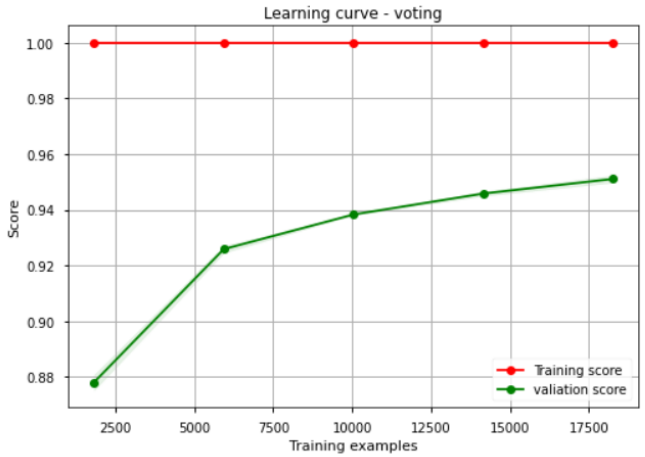
하드 보팅 : 0.9550

소프트 보팅 : 0.9645

Voting Classifier

```
y_pred = vot_clf.predict(X_val)
vot_cm = confusion_matrix(y_val, y_pred)
print(vot_cm)
print(classification_report(y_val, y_pred))
```

```
[[317   0   0   0   1   0   2   0   4   1]
 [  0 296   0   0   0   1   0   1   0   1]
 [  0   2 298   1   0   0   3   7   4   0]
 [  0   3   3 284   1   1   1   1   1   2]
 [  2   0   0   0 284   0   1   2   2   6]
 [  0   0   0   2   0 292   1   0   5   1]
 [  2   1   0   0   0   1 303   0   1   0]
 [  0   0   2   0   1   0   0 296   1   5]
 [  0   0   1   2   0   4   1   1 275   5]
 [  0   0   0   1   6   1   0   7   2 289]]
              precision    recall  f1-score   support

           0       0.99      0.98      0.98       325
           1       0.98      0.99      0.99       299
           2       0.98      0.95      0.96       315
           3       0.98      0.96      0.97       297
           4       0.97      0.96      0.96       297
           5       0.97      0.97      0.97       301
           6       0.97      0.98      0.98       308
           7       0.94      0.97      0.95       305
           8       0.93      0.95      0.94       289
           9       0.93      0.94      0.94       306

    accuracy                           0.96      3042
   macro avg       0.96      0.96      0.96      3042
weighted avg       0.96      0.96      0.96      3042
```



- 주요 성능 지표 결과

Confusion Matrix

Precision

Recall

F1-score

Support

Accuracy

- **Learning curve**

Voter중 soft를 사용하여
Train에서 cross-validation을 3으로 지정하여서
나타낸 learning curve.
Data가 늘수록 validation score가 증가하고 있다.

결론

모델 저장

```
model = vot_clf
```

```
import joblib
joblib.dump(model, "my_model.pkl")
model_loaded = joblib.load("my_model.pkl")
```

Evaluate

```
from sklearn.metrics import accuracy_score

y_test_pred = model_loaded.predict(X_test)
print(f'{accuracy_score(y_test, y_test_pred):0.4f}')
```

```
0.9484
```

모델을 저장하고

이것을 초기에 정한 test data 넣었을 때

accuracy의 값은 0.9484이 나왔다.

Q&A