GR5241 Statistical Machine Learning

Data Analysis Project Milestone 1

Name: Kangshuo Li UNI: kl3259

1 Background

The MNIST database of handwritten digits is one of the most commonly used dataset for training various image processing systems and machine learning algorithms. MNIST has a training set of 60,000 examples, and a test set of 10,000 examples. It is a good database for people who want to try learning techniques and pattern recognition methods on real-world data while spending minimal efforts on preprocessing and formatting.

MNIST is a subset of a larger set available from NIST. The digits have been size-normalized and centered in a fixed-size image. The original black and white (bilevel) images from NIST were size normalized. The resulting images contain grey levels as a result of the anti-aliasing technique used by the normalization algorithm. The images were centered in a 28×28 image by computing the center of mass of the pixels, and translating the image so as to position this point at the center of the 28×28 field.

2 Data Processing

A lot of packages in python has MNIST database. Let's use the one in `torch.utils.data`. Using the following code to download the data from the server.

Setup

In [ ]:
```
import torch
import torchvision
import pandas as pd
import numpy as np
import sklearn
import matplotlib
from matplotlib import pyplot as plt
import time
import os
import re
os.getcwd()
```

Out[ ]:
```
'/Users/kangshuoli/Documents/VScode_workspace/GR5241/Proj_milestone_1'
```

Data processing

In [ ]:
```
# Data download and preprocessing
from torch.utils import data

DOWNLOAD_MNIST = False # If already download , set as False

train_data = torchvision.datasets.MNIST(
```

```
    root ="./mnist/",
    train = True , # this is training data
    # transform = torchvision.transforms.ToTensor(),
    download = DOWNLOAD_MNIST
)
test_data = torchvision.datasets.MNIST(root = "./mnist/", train = False)

# change the features to numpy
X_train = train_data.train_data.numpy()
X_test = test_data.test_data.numpy()

# change the labels to numpy
Y_train = train_data.train_labels.numpy()

Y_test = test_data.test_labels.numpy()
```

```
/Users/kangshuoli/miniforge3/envs/env_torch/lib/python3.9/site-packages/torchvis
ion/datasets/mnist.py:64: UserWarning: train_data has been renamed data
  warnings.warn("train_data has been renamed data")
/Users/kangshuoli/miniforge3/envs/env_torch/lib/python3.9/site-packages/torchvis
ion/datasets/mnist.py:69: UserWarning: test_data has been renamed data
  warnings.warn("test_data has been renamed data")
/Users/kangshuoli/miniforge3/envs/env_torch/lib/python3.9/site-packages/torchvis
ion/datasets/mnist.py:54: UserWarning: train_labels has been renamed targets
  warnings.warn("train_labels has been renamed targets")
/Users/kangshuoli/miniforge3/envs/env_torch/lib/python3.9/site-packages/torchvis
ion/datasets/mnist.py:59: UserWarning: test_labels has been renamed targets
  warnings.warn("test_labels has been renamed targets")
```

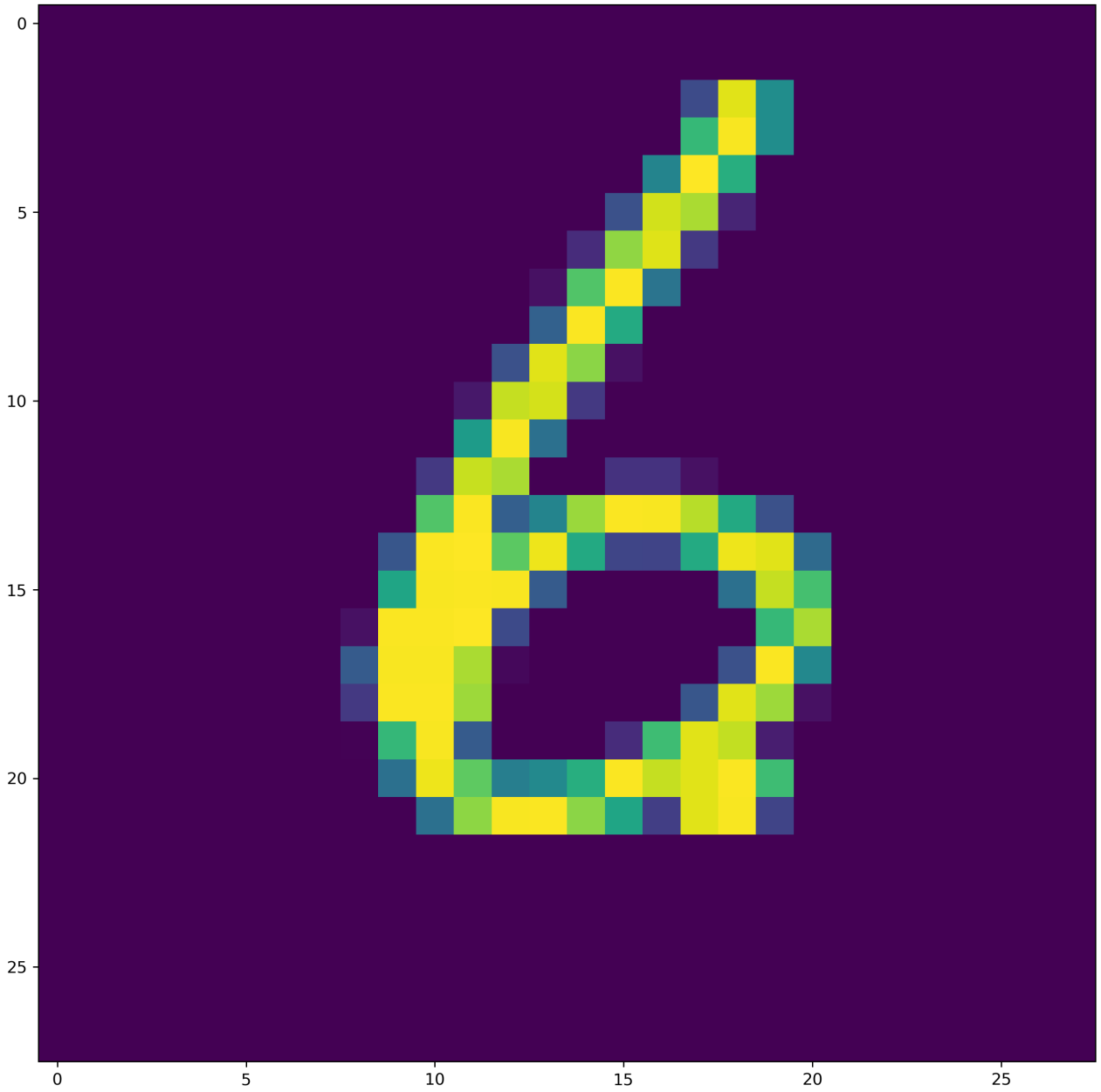
(a)

In [ ]:

```
print(X_train.shape) #(60000, 28, 28)
print(Y_train.shape) # (60000,)

random_index = np.random.randint(0, X_train.shape[0] - 1)
sample = X_train[random_index,:,:]
fig, ax = plt.subplots(1, 1, figsize = (12,12), dpi = 270)
ax.imshow(sample)
plt.show()
print(f'The corresponding label is: {Y_train[random_index]}')
```

```
(60000, 28, 28)
(60000,)
```




```
The corresponding label is: 6
```

According to the image and the label we sampled, the number shown from the image matches the label from Y_train.

(b)

In [ ]:

```
print(f'The dimension of X_train: {X_train.shape}')
print(f'The dimension of X_test: {X_test.shape}')

# print(np.max(X_train)) # 255
# print(np.min(X_train)) # 0
# print(np.max(X_test)) # 255
# print(np.min(X_test)) # 0
X_train = (X_train - np.min(X_train)) / (np.max(X_train) - np.min(X_train))
X_test = (X_test - np.min(X_test)) / (np.max(X_test) - np.min(X_test))
assert np.max(X_train) == 1
assert np.min(X_train) == 0
assert np.max(X_test) == 1
assert np.min(X_test) == 0
```

```
The dimension of X_train: (60000, 28, 28)
The dimension of X_test: (10000, 28, 28)
```

The dimension of X_train is 60000 *28* 28, and the dimension of X_test is 10000 *28* 28.

(c)

In [ ]:

```
# Convert them into dataframe
Y_train_df = pd.DataFrame(Y_train).astype(object)
Y_test_df = pd.DataFrame(Y_test).astype(object)

# Use the get_dummies() method in pandas
Y_train_df = pd.get_dummies(Y_train_df)
Y_train_onehot = Y_train_df.values
print(f'Y_train before onehot encoding: \n {Y_train}')
print(f'After onehot encoding, the shape of Y_train is: {Y_train_df.values.shape
print(Y_train_df.head())
Y_test_df = pd.get_dummies(Y_test_df)
Y_test_onehot = Y_test_df.values
print(f'Y_test before onehot encoding: \n {Y_test}')
print(f'After onehot encoding, the shape of Y_test is: {Y_test_df.values.shape}'
print(Y_test_df.head())
```

```
Y_train before onehot encoding:
 [5 0 4 ... 5 6 8]
After onehot encoding, the shape of Y_train is: (60000, 10)
   0_0  0_1  0_2  0_3  0_4  0_5  0_6  0_7  0_8  0_9
0    0    0    0    0    0    1    0    0    0    0
1    1    0    0    0    0    0    0    0    0    0
2    0    0    0    0    1    0    0    0    0    0
3    0    1    0    0    0    0    0    0    0    0
4    0    0    0    0    0    0    0    0    0    1
Y_test before onehot encoding:
 [7 2 1 ... 4 5 6]
After onehot encoding, the shape of Y_test is: (10000, 10)
   0_0  0_1  0_2  0_3  0_4  0_5  0_6  0_7  0_8  0_9
0    0    0    0    0    0    0    0    1    0    0
1    0    0    1    0    0    0    0    0    0    0
2    0    1    0    0    0    0    0    0    0    0
3    1    0    0    0    0    0    0    0    0    0
4    0    0    0    0    1    0    0    0    0    0
```

The label in this question is 0-9, which is the meaning of the image. While integer encoding may have the natrual order like 1 + 2 = 3, one-hot encoding can aviod this unexpected order of value, which is beneficial for the classification. Besides that, one-hot encoding can enlarge the label dimension with dummy variables, this can provide same distances among each classes(e.g. (0,0,1), (0,1,0), (1,0,0) are 3 classes with same distance $\sqrt{2}$ with each other) that eliminate the ordinal effect.

3 Before Deep Learning

Historically, many methods have been tested with MNIST training and test sets. Yann LeCun, Corinna Cortes, and Christopher J.C. Burges maintained the progress on the accuracy of the classification task until 2012, where you can find the details here: http://yann.lecun.com/exdb/mnist/. Among them, there are 4 reports on the following methods:

Methods	Feature Engineering	Test Error
KNN	None	5%
AdaBoost.M1 / C4.5	None	4.05%
SVM with Gaussian Kernel	None	1.4%

2. (12 points) A key property for a good machine learning algorithm is the *reproducibility*, meaning that one can repeatedly run the algorithm on certain datasets and obtain the same (or similar) results on a particular project.

   (a) (8 points) Try to implement and train the above mentioned classifier on the training dataset, and report the test errors of them using the test dataset. Can you reproduce the results? If not, please justify your reason.

   (b) (4 points) Pick your favorite classifier (not limited to the above mentioned algorithms) and try to implement it on the training set and report the test error using the test dataset. Turn the hyperparameters until it out perform all three of the classifier you implemented in part 2(a).

(a)

In [ ]:
```
from sklearn.neighbors import KNeighborsClassifier
from sklearn.ensemble import AdaBoostClassifier
from sklearn.tree import DecisionTreeClassifier
from sklearn.svm import SVC
from sklearn.model_selection import KFold, GridSearchCV
from sklearn.metrics import accuracy_score, f1_score

# flatten the image
X_train_list = []
for i in np.arange(X_train.shape[0]):
    X_train_list.append(X_train[i,:,:].flatten(order = 'C'))
X_test_list = []
for i in np.arange(X_test.shape[0]):
    X_test_list.append(X_test[i,:,:].flatten(order = 'C'))
X_train_flattened = np.stack(X_train_list)
X_test_flattened = np.stack(X_test_list)
```

In [ ]:
```
params_knn = {
    'n_neighbors': [2, 5, 10],
    'weights': ["uniform", "distance"],
    'p': [2],
    'n_jobs': [-1]
}

gs_knn = GridSearchCV(
    estimator = KNeighborsClassifier(),
```

```
    param_grid = params_knn,
    cv = 5,
    scoring = 'accuracy',
    refit = True,
    n_jobs = -1,
    verbose = 0
)
gs_knn.fit(X = X_train_flattened, y = Y_train_onehot)
print(f'The best training score of KNN is: {gs_knn.best_score_:0.4f}, the parame
print(f'The prediction score of KNN is: {accuracy_score(Y_test_onehot, gs_knn.be
```

```
The best training score of KNN is: 0.9694, the parameters used are: {'n_jobs': -
1, 'n_neighbors': 2, 'p': 2, 'weights': 'distance'}
The prediction score of KNN is: 0.9691
```

The best training result of KNN is :

> The best training score of KNN is: 0.9694, the parameters used are: {'n_jobs': -1, 'n_neighbors': 2, 'p': 2, 'weights': 'distance'}
>
> The prediction score of KNN is: 0.9691

Thus, KNN can reproduce the results.

> C4.5 is the successor to ID3 and removed the restriction that features must be categorical by dynamically defining a discrete attribute (based on numerical variables) that partitions the continuous attribute value into a discrete set of intervals. C4.5 converts the trained trees (i.e. the output of the ID3 algorithm) into sets of if-then rules. These accuracy of each rule is then evaluated to determine the order in which they should be applied. Pruning is done by removing a rule's precondition if the accuracy of the rule improves without it.
>
> CART (Classification and Regression Trees) is very similar to C4.5, but it differs in that it supports numerical target variables (regression) and does not compute rule sets. CART constructs binary trees using the feature and threshold that yield the largest information gain at each node.
>
> scikit-learn uses an optimised version of the CART algorithm.

If we use CART as an approximation of C4.5 by using sklearn, then the code is:

In [ ]:

```
# already gridsearched, use the best parameter to reduce the training time cost
# params_ada = {
#     'base_estimator': [DecisionTreeClassifier(max_depth = 8), DecisionTreeClas
#     'n_estimators': [100, 200, 500, 1000],
#     'learning_rate': [0.01, 0.1, 0.2],
#     'algorithm': ['SAMME.R'],
#     'random_state': [42]
# }

params_ada = {
    'base_estimator': [DecisionTreeClassifier(max_depth = 8)],
    'n_estimators': [1000],
    'learning_rate': [0.2],
```

```
    'algorithm': ['SAMME.R'],
    'random_state': [42]
}

gs_ada = GridSearchCV(
    estimator = AdaBoostClassifier(),
    param_grid = params_ada,
    cv = 5,
    scoring = 'accuracy',
    refit = True,
    n_jobs = -1,
    verbose = 1
)


gs_ada.fit(X = X_train_flattened, y = Y_train)
print(f'The best training score of AdaBoost is: {gs_ada.best_score_:0.4f}, the p
print(f'The prediction score of AdaBoost is: {accuracy_score(Y_test, gs_ada.best
```

```
Fitting 5 folds for each of 1 candidates, totalling 5 fits
The best training score of AdaBoost is: 0.9563, the parameters used are: {'algor
ithm': 'SAMME.R', 'base_estimator': DecisionTreeClassifier(max_depth=8), 'learni
ng_rate': 0.2, 'n_estimators': 1000, 'random_state': 42}
The prediction score of AdaBoost is: 0.9600
```

The best training result of AdaBoost is :

> The best training score of AdaBoost is: 0.9563, the parameters used are:
> {'algorithm': 'SAMME.R', 'base_estimator': DecisionTreeClassifier(max_depth=8),
> 'learning_rate': 0.2, 'n_estimators': 1000, 'random_state': 42}
>
> The prediction score of AdaBoost is: 0.9600

AdaBoost succeed to reproduce the results.

In [ ]:

```
X_train_flattened_df = pd.DataFrame(data = X_train_flattened)
sample_index = X_train_flattened_df.sample(
    n = 6000,
    replace = False,
    random_state = 42
).index
X_train_subsampled = X_train_flattened_df.iloc[sample_index,:]
Y_train_subsampled = Y_train[sample_index]
assert X_train_subsampled.shape[0] == Y_train_subsampled.shape[0]
X_train_subsampled.index = np.arange(X_train_subsampled.shape[0])
```

In [ ]:

```
# params_svc = {
#     'C': [0.001, 0.005, 0.01, 0.05, 0.1, 0.2, 0.5, 1],
#     'kernel': ['rbf'],
#     'degree': [3, 4, 5, 6],
#     'gamma': ['scale', 'auto'],
#     'shrinking': [True],
#     'tol': [0.0001, 0.00001],
#     'verbose': [False],
#     'random_state': [42]
# }
# gs_svc = GridSearchCV(
```

```
#     estimator = SVC(),
#     param_grid = params_svc,
#     cv = 5,
#     scoring = 'accuracy',
#     refit = True,
#     n_jobs = -1,
#     verbose = 2
# )
# gs_svc.fit(X = X_train_subsampled, y = Y_train_subsampled)
# print(f'The best training score of SVM classifier is: {gs_svc.best_score_:0.4f

'''
Only use 10% training data for gridsearch to speed up the hyperparameter tunning

The best training score of SVM classifier is: 0.9532, the parameters used are:
{'C': 1, 'degree': 3, 'gamma': 'scale', 'kernel': 'rbf', 'random_state': 42,
'shrinking': True, 'tol': 0.0001, 'verbose': False}
The prediction score of SVM classifier is: 0.9583
'''

params_svc = {
    'C': [1],
    'kernel': ['rbf'],
    'degree': [3],
    'gamma': ['scale'],
    'shrinking': [True],
    'tol': [0.0001],
    'verbose': [False],
    'random_state': [42]
}

gs_svc = GridSearchCV(
    estimator = SVC(),
    param_grid = params_svc,
    cv = 5,
    scoring = 'accuracy',
    refit = True,
    n_jobs = -1,
    verbose = 2
)

gs_svc.fit(X = X_train_flattened, y = Y_train)
print(f'The best training score of SVM classifier is: {gs_svc.best_score_:0.4f},
print(f'The prediction score of SVM classifier is: {accuracy_score(Y_test, gs_sv
```

```
Fitting 5 folds for each of 1 candidates, totalling 5 fits
[CV] END C=1, degree=3, gamma=scale, kernel=rbf, random_state=42, shrinking=Tru
e, tol=0.0001, verbose=False; total time= 4.0min
[CV] END C=1, degree=3, gamma=scale, kernel=rbf, random_state=42, shrinking=Tru
e, tol=0.0001, verbose=False; total time= 4.1min
[CV] END C=1, degree=3, gamma=scale, kernel=rbf, random_state=42, shrinking=Tru
e, tol=0.0001, verbose=False; total time= 4.1min
[CV] END C=1, degree=3, gamma=scale, kernel=rbf, random_state=42, shrinking=Tru
e, tol=0.0001, verbose=False; total time= 4.1min
[CV] END C=1, degree=3, gamma=scale, kernel=rbf, random_state=42, shrinking=Tru
e, tol=0.0001, verbose=False; total time= 4.1min
The best training score of SVM classifier is: 0.9769, the parameters used are:
{'C': 1, 'degree': 3, 'gamma': 'scale', 'kernel': 'rbf', 'random_state': 42, 'sh
rinking': True, 'tol': 0.0001, 'verbose': False}
The prediction score of SVM classifier is: 0.9792
```

The best result of SVM classifier is :

> The best training score of SVM classifier is: 0.9769, the parameters used are: {'C': 1, 'degree': 3, 'gamma': 'scale', 'kernel': 'rbf', 'random_state': 42, 'shrinking': True, 'tol': 0.0001, 'verbose': False}
>
> The prediction score of SVM classifier is: 0.9792

So the SVM classifier with Gaussian rbf can reproduce similar performance. Potential reasons: We only search the best combination of hyperparameters in a subsampled dataset, which might not be the best combination of hyperparameters of the overall dataset. Besides that, the grid Search cross validation is limited with the hyperparameter range defined by us, which might avoid us finding the global optimal hypermeters.

(b)

Use stacking classifier

Classifier in first layer: LogisticRegression, RandomForest, KNN, AdaBoost

Final classifier: lightGBM

Use the best hyperparameter combinations in previous gridsearch. Use 10-fold cross validation for training.

In [ ]:
```
from sklearn.ensemble import StackingClassifier
from sklearn.ensemble import RandomForestClassifier
from sklearn.linear_model import LogisticRegression
import lightgbm as lgb
import xgboost as xgb
%xmode plain
```

```
Exception reporting mode: Plain
```

In [ ]:
```
estimator_list = [
        # ('xgb', xgb.XGBClassifier(
        #     n_estimators = 1000,
        #     max_depth = 8,
        #     grow_policy = 1,
        #     verbosity = 0,
        #     booster = "gbtree",
        #     n_jobs = -1,
        #     reg_alpha = 0.6,
        #     reg_lambda = 0.4,
        #     random_state = 42,
        #     use_label_encoder = False
        # )),
        ('lr', LogisticRegression(
            penalty = 'l2',
            C = 1.0,
            class_weight = 'balanced',
            random_state = 42,
            max_iter = 1000,
            multi_class = 'ovr',
```

```
            n_jobs = -1
        )),
        ('rf', RandomForestClassifier(
            n_estimators = 1000,
            max_depth = 8,
            max_features = "sqrt",
            bootstrap = True,
            oob_score = True,
            verbose = 0,
            random_state = 42
        )),
        ('knn', KNeighborsClassifier(
            n_neighbors = 2,
            p = 2,
            n_jobs =  -1,
            weights = "distance"
        )),
        ('ada', AdaBoostClassifier(
            algorithm =  'SAMME.R',
            base_estimator = DecisionTreeClassifier(max_depth=8),
            learning_rate = 0.2,
            n_estimators = 1000,
            random_state = 42
        ))
    ]


stacking_clf = StackingClassifier(
    estimators = estimator_list,
    final_estimator = lgb.LGBMClassifier(
        boosting_type = 'gbdt',
        max_depth = 10,
        learning_rate = 0.1,
        n_estimators = 1000,
        objective = 'multiclass',
        class_weight = 'balanced',
        n_jobs = -1,
        random_state = 42,
        importance_type = 'gain',
        reg_alpha = 0.2
    ),
    cv = 10,
    stack_method = 'predict_proba',
    n_jobs = -1,
    passthrough = True,
    verbose = 1
)

stacking_clf.fit(X_train_flattened, Y_train)
print(f'The best training score of Stacking classifier is: {accuracy_score(Y_tra
print(f'The prediction score of Stacking classifier is: {accuracy_score(Y_test,
```

```
[Parallel(n_jobs=-1)]: Using backend ThreadingBackend with 10 concurrent worker
s.
[Parallel(n_jobs=-1)]: Using backend ThreadingBackend with 10 concurrent worker
s.
[Parallel(n_jobs=-1)]: Using backend ThreadingBackend with 10 concurrent worker
s.
[Parallel(n_jobs=-1)]: Using backend ThreadingBackend with 10 concurrent worker
```

```
s.
[Parallel(n_jobs=-1)]: Done   2 out of  10 | elapsed:  1.4min remaining:  5.5min
[Parallel(n_jobs=-1)]: Done  10 out of  10 | elapsed:  1.4min finished
[Parallel(n_jobs=-1)]: Done   2 out of  10 | elapsed:  8.5min remaining: 34.0min
[Parallel(n_jobs=-1)]: Done  10 out of  10 | elapsed:  8.6min finished
[Parallel(n_jobs=-1)]: Done   2 out of  10 | elapsed:  9.6min remaining: 38.3min
[Parallel(n_jobs=-1)]: Done  10 out of  10 | elapsed:  9.6min finished
[Parallel(n_jobs=-1)]: Done   2 out of  10 | elapsed: 116.6min remaining: 466.5m
in
[Parallel(n_jobs=-1)]: Done  10 out of  10 | elapsed: 117.0min finished
The best training score of Stacking classifier is: 1.0000
The prediction score of Stacking classifier is: 0.9820
```

The stacking classifier outperforms all the previous models with a test accuracy of 0.9820.

In [ ]: