WINTER SEMESTER 2022 - 23

MACHINE LEARNING

CSE4020

LAB ASSIGNMENT 3

Name: Gaurav Navada
Registration no.: 20BKT0128
Date of submission: 2/19/2023
Slot no.: L43 + L44
Faculty Name: Prof. SUDHA.S

Question 1

Write a program to implement k-Nearest Neighbour algorithm to classify the iris data set. Print both correct and wrong predictions.

In [12]:
```
from sklearn.datasets import load_iris
from sklearn.model_selection import train_test_split
from sklearn.neighbors import KNeighborsClassifier

# Load the iris dataset
iris = load_iris()

# Split the dataset into training and testing sets
X_train, X_test, y_train, y_test = train_test_split(iris.data, iris.target, test_size=0.3, random_state=42)

# Create a k-Nearest Neighbors classifier with k=3
knn = KNeighborsClassifier(n_neighbors=3)

# Train the classifier on the training set
knn.fit(X_train, y_train)

# Predict the classes of the test set
y_pred = knn.predict(X_test)

# Suppress the warning
import warnings
warnings.filterwarnings("ignore", category=FutureWarning)

# Print the correct and wrong predictions
correct = 0
wrong = 0
for i in range(len(y_test)):
    if y_test[i] == y_pred[i]:
        print("Correct prediction: Actual class = {}, Predicted class = {}".format(y_test[i], y_pred[i]))
        correct += 1
    else:
        print("Wrong prediction: Actual class = {}, Predicted class = {}".format(y_test[i], y_pred[i]))
        wrong += 1

# Print the accuracy of the classifier
accuracy = correct / len(y_test)
print("Accuracy: {:.2f}%".format(accuracy * 100))
```

```
Correct prediction: Actual class = 1, Predicted class = 1
Correct prediction: Actual class = 0, Predicted class = 0
Correct prediction: Actual class = 2, Predicted class = 2
Correct prediction: Actual class = 1, Predicted class = 1
Correct prediction: Actual class = 1, Predicted class = 1
Correct prediction: Actual class = 0, Predicted class = 0
Correct prediction: Actual class = 1, Predicted class = 1
Correct prediction: Actual class = 2, Predicted class = 2
Correct prediction: Actual class = 1, Predicted class = 1
Correct prediction: Actual class = 1, Predicted class = 1
Correct prediction: Actual class = 2, Predicted class = 2
Correct prediction: Actual class = 0, Predicted class = 0
Correct prediction: Actual class = 0, Predicted class = 0
Correct prediction: Actual class = 0, Predicted class = 0
Correct prediction: Actual class = 0, Predicted class = 0
Correct prediction: Actual class = 1, Predicted class = 1
Correct prediction: Actual class = 2, Predicted class = 2
Correct prediction: Actual class = 1, Predicted class = 1
Correct prediction: Actual class = 1, Predicted class = 1
Correct prediction: Actual class = 2, Predicted class = 2
Correct prediction: Actual class = 0, Predicted class = 0
Correct prediction: Actual class = 2, Predicted class = 2
Correct prediction: Actual class = 0, Predicted class = 0
Correct prediction: Actual class = 2, Predicted class = 2
Correct prediction: Actual class = 2, Predicted class = 2
Correct prediction: Actual class = 2, Predicted class = 2
Correct prediction: Actual class = 2, Predicted class = 2
Correct prediction: Actual class = 2, Predicted class = 2
Correct prediction: Actual class = 0, Predicted class = 0
Correct prediction: Actual class = 0, Predicted class = 0
Correct prediction: Actual class = 0, Predicted class = 0
Correct prediction: Actual class = 0, Predicted class = 0
Correct prediction: Actual class = 1, Predicted class = 1
Correct prediction: Actual class = 0, Predicted class = 0
Correct prediction: Actual class = 0, Predicted class = 0
Correct prediction: Actual class = 2, Predicted class = 2
Correct prediction: Actual class = 1, Predicted class = 1
Correct prediction: Actual class = 0, Predicted class = 0
Correct prediction: Actual class = 0, Predicted class = 0
Correct prediction: Actual class = 0, Predicted class = 0
Correct prediction: Actual class = 2, Predicted class = 2
Correct prediction: Actual class = 1, Predicted class = 1
Correct prediction: Actual class = 1, Predicted class = 1
Correct prediction: Actual class = 0, Predicted class = 0
Correct prediction: Actual class = 0, Predicted class = 0
Accuracy: 100.00%
```

Question 2

Train SVM classifier using sklearn digits dataset(i.e from sklearn datasets import load_digits) and then:

a. Measure accuracy of your model using different kernels such as rbf and linear.

b. Tune your model further using regularization and gamma parameters and try tocome up highest accuracy score.

c. Use 80% of samples as training data size.

Code Explanation

a. We then train an SVM classifier with two different kernels, 'linear' and 'rbf'. For each kernel, we create an SVC object and fit it to the training data using the fit method. We then use the trained classifier to predict the labels of the testing data using the predict method. We calculate the accuracy score of the classifier using the accuracy_score function from sklearn.metrics.

b. In this code, we define a parameter grid with different values of C and gamma. We then create an SVC object with 'rbf' kernel and use GridSearchCV to perform a grid search over the parameter grid. We fit the grid search object to the training data using the fit method. Note that we use the GridSearchCV class from sklearn to perform the grid search. This class performs cross-validation over the parameter grid and selects the best parameters based on the cross-validation score.

c. In this code, we first load the digits dataset using load_digits. We then split the dataset into training and testing sets using train_test_split. We use 80% of the samples as the training data size, as specified in the problem statement.

In [13]:
```
from sklearn.datasets import load_digits
from sklearn.model_selection import train_test_split, GridSearchCV
from sklearn.svm import SVC
from sklearn.metrics import accuracy_score

# Load the digits dataset
digits = load_digits()

# Split the dataset into training and testing sets
X_train, X_test, y_train, y_test = train_test_split(digits.data, digits.target, test_size=0.2, random_state=42)

# Train an SVM classifier with different kernels
kernels = ['linear', 'rbf']
for kernel in kernels:
    clf = SVC(kernel=kernel)
    clf.fit(X_train, y_train)
    y_pred = clf.predict(X_test)
    acc = accuracy_score(y_test, y_pred)
    print(f"Accuracy score using {kernel} kernel: {acc:.3f}")

# Tune the SVM classifier using grid search
param_grid = {'C': [0.1, 1, 10, 100], 'gamma': [0.001, 0.01, 0.1, 1]}
clf = SVC(kernel='rbf')
grid_search = GridSearchCV(clf, param_grid=param_grid)
grid_search.fit(X_train, y_train)
print(f"Best parameters: {grid_search.best_params_}")
print(f"Accuracy score: {grid_search.best_score_:.3f}")
```

```
Accuracy score using linear kernel: 0.978
Accuracy score using rbf kernel: 0.986
Best parameters: {'C': 10, 'gamma': 0.001}
Accuracy score: 0.990
```

Question 3

Build an Artificial Neural Network by implementing the Back propagation algorithm and test the same using appropriate data sets.

In [14]:
```
import numpy as np

class NeuralNetwork:
    def __init__(self, input_size, hidden_size, output_size):
        self.input_size = input_size
        self.hidden_size = hidden_size
        self.output_size = output_size

        self.weights1 = np.random.randn(self.input_size, self.hidden_size)
        self.bias1 = np.zeros((1, self.hidden_size))
        self.weights2 = np.random.randn(self.hidden_size, self.output_size)
        self.bias2 = np.zeros((1, self.output_size))

    def sigmoid(self, z):
        return 1 / (1 + np.exp(-z))

    def sigmoid_derivative(self, z):
        return self.sigmoid(z) * (1 - self.sigmoid(z))

    def forward(self, X):
        self.z1 = np.dot(X, self.weights1) + self.bias1
        self.a1 = self.sigmoid(self.z1)
        self.z2 = np.dot(self.a1, self.weights2) + self.bias2
        self.a2 = self.sigmoid(self.z2)
        return self.a2

    def backward(self, X, y, output):
        self.error = output - y
        self.delta2 = np.multiply(self.error, self.sigmoid_derivative(self.z2))
        self.d_weights2 = np.dot(self.a1.T, self.delta2)
        self.d_bias2 = np.sum(self.delta2, axis=0)

        self.delta1 = np.dot(self.delta2, self.weights2.T) * self.sigmoid_derivative(self.z1)
        self.d_weights1 = np.dot(X.T, self.delta1)
        self.d_bias1 = np.sum(self.delta1, axis=0)

    def update_weights(self, learning_rate):
        self.weights1 -= learning_rate * self.d_weights1
        self.bias1 -= learning_rate * self.d_bias1
        self.weights2 -= learning_rate * self.d_weights2
        self.bias2 -= learning_rate * self.d_bias2

    def train(self, X, y, learning_rate, epochs):
        for i in range(epochs):
            output = self.forward(X)
            self.backward(X, y, output)
            self.update_weights(learning_rate)

    def predict(self, X):
        return self.forward(X)
```

Example of how to use this class to train a neural network on the XOR function

In [15]:
```
X = np.array([[0, 0], [0, 1], [1, 0], [1, 1]])
y = np.array([[0], [1], [1], [0]])

nn = NeuralNetwork(2, 3, 1)
nn.train(X, y, 0.1, 10000)

print("Prediction:")
print(nn.predict(X))
```

```
Prediction:
[[0.04323574]
 [0.95885508]
 [0.95763163]
 [0.03653322]]
```

Question 4

Bagging Ensembles including Bagged Decision Trees, Random Forest and Extra Trees.

Code Explanation

In this example, we first load the Iris dataset and split it into training and testing sets. We then create three different bagging ensembles: a Bagged Decision Trees ensemble, a Random Forest ensemble, and an Extra Trees ensemble. We fit each ensemble to the training data and evaluate the accuracy of each ensemble on the test data. Finally, we print the accuracy score for each ensemble.

In [16]:
```
from sklearn.datasets import load_iris
from sklearn.model_selection import train_test_split
from sklearn.ensemble import BaggingClassifier, RandomForestClassifier, ExtraTreesClassifier
from sklearn.tree import DecisionTreeClassifier

# Load the Iris dataset
iris = load_iris()

# Split the dataset into training and testing sets
X_train, X_test, y_train, y_test = train_test_split(iris.data, iris.target, test_size=0.2)

# Create a Bagged Decision Trees ensemble
bagging = BaggingClassifier(base_estimator=DecisionTreeClassifier(), n_estimators=10)
bagging.fit(X_train, y_train)
print("Bagging Decision Trees accuracy:", bagging.score(X_test, y_test))

# Create a Random Forest ensemble
rf = RandomForestClassifier(n_estimators=10)
rf.fit(X_train, y_train)
print("Random Forest accuracy:", rf.score(X_test, y_test))

# Create an Extra Trees ensemble
et = ExtraTreesClassifier(n_estimators=10)
et.fit(X_train, y_train)
print("Extra Trees accuracy:", et.score(X_test, y_test))
```

```
Bagging Decision Trees accuracy: 0.9666666666666667
Random Forest accuracy: 0.9666666666666667
Extra Trees accuracy: 0.9666666666666667
```

Question 5

Boosting Ensembles including AdaBoost and Stochastic Gradient Boosting.

Code Explanation

In this example, the Iris dataset is initially loaded and then divided into training and testing sets. Then, two distinct boosting ensembles—an AdaBoost ensemble and a Stochastic Gradient Boosting ensemble—are created. We analyse the accuracy of each ensemble using the test data after fitting each ensemble to the training set of data. The accuracy score for each ensemble is then printed.

We employ a decision tree with a maximum depth of one as the base estimator in the AdaBoost ensemble. This is so because AdaBoost performs best with simple and weak models, and a decision tree with a maximum depth of 1 is both. We employ decision trees with a maximum depth of 2 as the base estimators in the Gradient Boosting ensemble. This is due to the fact that Gradient Boosting functions best with complex models.

In [17]:
```
from sklearn.datasets import load_iris
from sklearn.model_selection import train_test_split
from sklearn.ensemble import AdaBoostClassifier, GradientBoostingClassifier
from sklearn.tree import DecisionTreeClassifier

# Load the Iris dataset
iris = load_iris()

# Split the dataset into training and testing sets
X_train, X_test, y_train, y_test = train_test_split(iris.data, iris.target, test_size=0.2)

# Create an AdaBoost ensemble
ada = AdaBoostClassifier(base_estimator=DecisionTreeClassifier(max_depth=1), n_estimators=10)
ada.fit(X_train, y_train)
print("AdaBoost accuracy:", ada.score(X_test, y_test))

# Create a Gradient Boosting ensemble
gb = GradientBoostingClassifier(max_depth=2, n_estimators=10)
gb.fit(X_train, y_train)
print("Gradient Boosting accuracy:", gb.score(X_test, y_test))
```

```
AdaBoost accuracy: 1.0
Gradient Boosting accuracy: 1.0
```

Question 6

Voting Ensembles for averaging the predictions for any arbitrary models.

Code Explanation

In this example, we first load the Iris dataset and split it into training and testing sets. We then create three arbitrary models: a logistic regression model, a decision tree model, and a random forest model. We then create a voting ensemble of these three models using the 'VotingClassifier' class in scikit-learn. We set the 'voting' parameter to 'hard', which means the ensemble will make predictions by majority voting. We fit the ensemble to the training data and evaluate the accuracy of the ensemble on the test data. Finally, we print the accuracy score for the ensemble.

In [18]:
```
from sklearn.datasets import load_iris
from sklearn.model_selection import train_test_split
from sklearn.linear_model import LogisticRegression
from sklearn.tree import DecisionTreeClassifier
from sklearn.ensemble import RandomForestClassifier, VotingClassifier

# Load the Iris dataset
iris = load_iris()

# Split the dataset into training and testing sets
X_train, X_test, y_train, y_test = train_test_split(iris.data, iris.target, test_size=0.2)

# Create some arbitrary models
lr = LogisticRegression()
dt = DecisionTreeClassifier()
rf = RandomForestClassifier()

# Create a voting ensemble of the arbitrary models
ensemble = VotingClassifier(estimators=[('lr', lr), ('dt', dt), ('rf', rf)], voting='hard')
ensemble.fit(X_train, y_train)
print("Ensemble accuracy:", ensemble.score(X_test, y_test))
```

```
Ensemble accuracy: 0.9333333333333333
```