Abhishek Murthy
21BDS0064
Fall Sem 2024-2025
DA - 3
Machine Learning Lab
11-09-2024

KNN Algorithm

Code:

```
import numpy as np

class KNN:
    def __init__(self, k=3):
        self.k = k

    def fit(self, X, y):
        self.X_train = X
        self.y_train = y

    def euclidean_distance(self, x1, x2):
        return np.sqrt(np.sum((x1 - x2) ** 2))

    def predict(self, X):
        y_pred = [self._predict(x) for x in X]
        return np.array(y_pred)

    def _predict(self, x):
        distances = [self.euclidean_distance(x, x_train) for x_train in self.X_train]
        k_indices = np.argsort(distances)[:self.k]
        k_nearest_labels = [self.y_train[i] for i in k_indices]
        most_common = np.bincount(k_nearest_labels).argmax()
        return most_common


from sklearn.datasets import make_classification
from sklearn.model_selection import train_test_split
X, y = make_classification(n_samples = 50, n_features = 2, n_informative = 2, n_redundant = 0,
n_classes = 2, weights = [0.51, .49])

X_train, X_test,y_train, y_test = train_test_split(X,y , random_state=104, test_size=0.25)

knn = KNN(k=2)
knn.fit(X_train, y_train)
X_new = np.array([[5.5, 3.5]])
prediction = knn.predict(X_test)

print("Predicted class:", prediction)
print("Actual class:", y_test)
```

Output:

```
Predicted class: [0 0 1 0 0 0 0 1 1 0 1 0 1]
Actual class: [0 0 1 0 0 0 0 1 1 0 1 0 1]
```

Logistic Regression

Code:

```
import numpy as np
from sklearn.model_selection import train_test_split
from sklearn import datasets
from sklearn.metrics import confusion_matrix, accuracy_score, precision_score,
recall_score, f1_score

class LogisticRegression:
    def __init__(self, learning_rate=0.001, n_iters=1000):
        self.lr = learning_rate
        self.n_iters = n_iters
        self.weights = None
        self.bias = None
        self.losses = []

    def _sigmoid(self, x):
        return 1 / (1 + np.exp(-x))

    def compute_loss(self, y_true, y_pred):
        epsilon = 1e-9
        y1 = y_true * np.log(y_pred + epsilon)
        y2 = (1-y_true) * np.log(1 - y_pred + epsilon)
        return -np.mean(y1 + y2)

    def feed_forward(self, X):
        z = np.dot(X, self.weights) + self.bias
        A = self._sigmoid(z)
        return A

    def fit(self, X, y):
        n_samples, n_features = X.shape
        self.weights = np.zeros(n_features)
        self.bias = 0

        for _ in range(self.n_iters):
            A = self.feed_forward(X)
            self.losses.append(self.compute_loss(y, A))
            dz = A - y
            dw = (1 / n_samples) * np.dot(X.T, dz)
            db = (1 / n_samples) * np.sum(dz)
            self.weights -= self.lr * dw
            self.bias -= self.lr * db
```

```
    def predict(self, X):
        threshold = .5
        y_hat = np.dot(X, self.weights) + self.bias
        y_predicted = self._sigmoid(y_hat)
        y_predicted_cls = [1 if i > threshold else 0 for i in y_predicted]
        return np.array(y_predicted_cls)

dataset = datasets.load_breast_cancer()
X, y = dataset.data, dataset.target
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=1234)
regressor = LogisticRegression(learning_rate=0.0001, n_iters=1000)
regressor.fit(X_train, y_train)
predictions = regressor.predict(X_test)
cm = confusion_matrix(y_test, predictions)
accuracy = accuracy_score(y_test, predictions)
precision = precision_score(y_test, predictions)
sensitivity = recall_score(y_test, predictions)
f_score = f1_score(y_test, predictions)

print(f"Test accuracy: {accuracy:.3f}")
print(f"Confusion Matrix:\n{cm}")
print(f"Precision: {precision:.3f}")
print(f"Sensitivity (Recall): {sensitivity:.3f}")
print(f"F1 Score: {f_score:.3f}")
```

Output

```
Test accuracy: 0.930
Confusion Matrix:
[[39  6]
 [ 2 67]]
Precision: 0.918
Sensitivity (Recall): 0.971
F1 Score: 0.944
```

Multi-layer Perceptron

Code:

```
import numpy as np
from sklearn import datasets
iris = datasets.load_iris()
X = iris["data"][:, (2, 3)]
y = (iris["target"] == 2).astype(int)  # 1 if Iris-Virginica, else 0
y = y.reshape([150,1])

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

def sigmoid_derivative(z):
    s = sigmoid(z)
    return s * (1 - s)

class MLP:
    def __init__(self, input_size, hidden_size, output_size, learning_rate=0.01):
        self.input_size = input_size
        self.hidden_size = hidden_size
        self.output_size = output_size
        self.learning_rate = learning_rate

        self.weights1 = np.random.randn(self.input_size, self.hidden_size)
        self.weights2 = np.random.randn(self.hidden_size, self.output_size)

        self.bias1 = np.zeros((1, self.hidden_size))
        self.bias2 = np.zeros((1, self.output_size))

    def fit(self, X, y, epochs=1000):
        for epoch in range(epochs):
            layer1 = X.dot(self.weights1) + self.bias1
            activation1 = sigmoid(layer1)
            layer2 = activation1.dot(self.weights2) + self.bias2
            activation2 = sigmoid(layer2)

            error = activation2 - y
            d_weights2 = activation1.T.dot(error * sigmoid_derivative(layer2))
            d_bias2 = np.sum(error * sigmoid_derivative(layer2), axis=0, keepdims=True)
            error_hidden = error.dot(self.weights2.T) * sigmoid_derivative(layer1)
            d_weights1 = X.T.dot(error_hidden)
            d_bias1 = np.sum(error_hidden, axis=0, keepdims=True)

            self.weights2 -= self.learning_rate * d_weights2
            self.bias2 -= self.learning_rate * d_bias2
            self.weights1 -= self.learning_rate * d_weights1
            self.bias1 -= self.learning_rate * d_bias1

    def predict(self, X):
        layer1 = X.dot(self.weights1) + self.bias1
```

```
        activation1 = sigmoid(layer1)
        layer2 = activation1.dot(self.weights2) + self.bias2
        activation2 = sigmoid(layer2)
        return (activation2 > 0.5).astype(int)


mlp = MLP(input_size=2, hidden_size=4, output_size=1)
mlp.fit(X, y)
y_pred = mlp.predict(X)
accuracy = np.mean(y_pred == y)
print(f"Accuracy: {accuracy:.2f}")
```

Output:

```
mlp = MLP(input_size=2, hidden_size=4, output_size=1)
mlp.fit(X, y)
y_pred = mlp.predict(X)
accuracy = np.mean(y_pred == y)
print(f"Accuracy: {accuracy:.2f}")
```

✓ 0.2s

```
Accuracy: 0.96
```