LINEAR_REGRESSION_CLASSIFIER

```
import numpy as np
from sklearn.datasets import load_iris
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler
from sklearn.metrics import accuracy_score

class LinearRegressionClassifier:
    def __init__(self, learning_rate=0.01, n_iterations=1000):
        self.learning_rate = learning_rate
        self.n_iterations = n_iterations
        self.weights = None
        self.bias = None

    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_iterations):
            y_predicted = np.dot(X, self.weights) + self.bias

            # Gradient descent
            dw = (1 / n_samples) * np.dot(X.T, (y_predicted - y))
            db = (1 / n_samples) * np.sum(y_predicted - y)

            self.weights -= self.learning_rate * dw
            self.bias -= self.learning_rate * db
```

```
    def predict(self, X):
        y_predicted = np.dot(X, self.weights) + self.bias
        return np.where(y_predicted >= 0.5, 1, 0)

iris = load_iris()
X = iris.data
y = iris.target

X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)

scaler = StandardScaler()
X_train = scaler.fit_transform(X_train)
X_test = scaler.transform(X_test)

classifiers = []
for class_label in np.unique(y):
    y_binary = (y_train == class_label) * 1
    classifier = LinearRegressionClassifier()
    classifier.fit(X_train, y_binary)
    classifiers.append(classifier)

y_pred_train = np.array([classifier.predict(X_train) for classifier in classifiers]).T
y_pred_test = np.array([classifier.predict(X_test) for classifier in classifiers]).T

y_pred_train = np.argmax(y_pred_train, axis=1)
y_pred_test = np.argmax(y_pred_test, axis=1)

accuracy_train = accuracy_score(y_train, y_pred_train)
accuracy_test = accuracy_score(y_test, y_pred_test)

print("Training Accuracy:", accuracy_train)
```

```
print("Testing Accuracy:", accuracy_test)
```

OUTPUT

```
Training Accuracy: 0.7333333333333333
Testing Accuracy: 0.8
```

EXPLANATION

>>Objective:

The goal of linear regression is to find the linear relationship between input features and output labels.

>>Implementation:
The **Linear Regression Classifier** class is implemented with methods for fitting the model (**fit**) and making predictions (**predict**).

>>Gradient descent is used for parameter estimation, where the weights and bias are updated iteratively to minimize the mean squared error between predicted and actual values.

>>Usage:
Initialize the classifier, fit it to training data using **fit**, and then make predictions on test data using **predict**.

NAVIE_BAYES

```
import numpy as np
from sklearn.datasets import load_iris
from sklearn.model_selection import train_test_split
from sklearn.metrics import accuracy_score

class NaiveBayesClassifier:
    def __init__(self):
        self.class_probs = None
        self.feature_probs = None

    def fit(self, X, y):
        n_samples, n_features = X.shape
        self.classes = np.unique(y)
        n_classes = len(self.classes)

        self.class_probs = np.zeros(n_classes)
        for i, c in enumerate(self.classes):
            self.class_probs[i] = np.sum(y == c) / n_samples

        self.feature_probs = []
        for c in self.classes:
            class_indices = np.where(y == c)[0]
            class_features = X[class_indices]

            feature_probs_c = []
            for feature_idx in range(n_features):
                feature_values = class_features[:, feature_idx]

                if isinstance(feature_values[0], (int, float)):
                    mean = np.mean(feature_values)
                    std = np.std(feature_values)
                    feature_probs_c.append((mean, std))
                else:
                    unique_values, counts = np.unique(feature_values, return_counts=True)
                    probs = counts / len(class_indices)
                    feature_probs_c.append(dict(zip(unique_values, probs)))

            self.feature_probs.append(feature_probs_c)

    def predict(self, X):
        predictions = []
        for sample in X:
```

```
            probs = []
            for i, c in enumerate(self.classes):
                class_prob = self.class_probs[i]
                feature_probs_c = self.feature_probs[i]
                likelihood = 1

                for feature_idx, value in enumerate(sample):
                    if isinstance(value, (int, float)):
                        mean, std = feature_probs_c[feature_idx]
                        likelihood *= self.gaussian_probability(value, mean, std)
                    else:
                        if value in feature_probs_c[feature_idx]:
                            likelihood *= feature_probs_c[feature_idx][value]
                        else:
                            likelihood *= 0

                probs.append(class_prob * likelihood)

            predictions.append(self.classes[np.argmax(probs)])

        return predictions

    def gaussian_probability(self, x, mean, std):
        exponent = np.exp(-((x - mean) ** 2) / (2 * std ** 2))
        return (1 / (np.sqrt(2 * np.pi) * std)) * exponent
iris = load_iris()
X = iris.data
y = iris.target

X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)

classifier = NaiveBayesClassifier()
classifier.fit(X_train, y_train)

y_pred_train = classifier.predict(X_train)
y_pred_test = classifier.predict(X_test)

accuracy_train = accuracy_score(y_train, y_pred_train)
accuracy_test = accuracy_score(y_test, y_pred_test)

print("Training Accuracy:", accuracy_train)
print("Testing Accuracy:", accuracy_test)
```

OUTPUT

Training Accuracy: 0.95
Testing Accuracy: 1.0

EXPLANATION

>>Objective:
Naive Bayes is a probabilistic classifier based on Bayes' theorem with the "naive" assumption of independence between features.
>>Implementation:
The **Naive Bayes Classifier** class is implemented with methods for fitting the model (**fit**) and making predictions (**predict**).

>>Class probabilities and feature probabilities are calculated from the training data, assuming that features are conditionally independent given the class.
Predictions are made by multiplying class and feature probabilities and selecting the class with the highest probability.

>>Usage:
Initialize the classifier, fit it to training data using **fit**, and then make predictions on test data using **predict**.