```python
import numpy as np
class GradientDescentMethods:
    class GDRegression:
        def __init__(self,learning_rate=0.01,epoch=100):
            self.learning_rate = learning_rate
            self.epoch = epoch
            self.coef_m = None
            self.intercept_b = None

        def fit(self,X_train,y_train):
            self.coef_m = np.ones(X_train.shape[1])
            self.intercept_b = 0

            for i in range(self.epoch):

                y_hat = np.dot(X_train,self.coef_m) + self.intercept_b

                der_intercept = -2 * np.mean(y_train - y_hat)
                self.intercept_b -= (self.learning_rate * der_intercept)

                der_coef = -2 * np.dot(y_train - y_hat,X_train)/X_train.shape[0]
                self.coef_m -= (self.learning_rate * der_coef)

        def predict(self,X_test):
            return np.dot(X_test,self.coef_m) +self.intercept_b


    class SGDRegression:
        def __init__(self,learning_rate=0.01,epoch=100):
            self.learning_rate = learning_rate
            self.epoch = epoch
            self.coef_m = None
            self.intercept_b = None

        def fit(self,X_train,y_train):
            self.intercept_b = 0
```

```
        self.coef_m = np.ones(X_train.shape[1])

        for i in range(self.epoch):
            for j in range(X_train.shape[0]):
                id = np.random.randint(0,X_train.shape[0])
                y_hat = np.dot(X_train[id] , self.coef_m) + self.intercept_b

                der_intercept = -2 * (y_train[id]- y_hat)
                self.intercept_b -= (self.learning_rate * der_intercept)

                der_coef = -2 * np.dot((y_train[id] - y_hat),X_train[id])
                self.coef_m -= (self.learning_rate * der_coef)

    def predict(self,X_test):
        return np.dot(X_test,self.coef_m) + self.intercept_b


class MBGDRegression:

    def __init__(self,learning_rate=0.01,epoch=100,batch_size=20):
        self.learning_rate = learning_rate
        self.epoch = epoch
        self.batch_size = batch_size
        self.coef_m = None
        self.intercept_b = None

    def fit(self,X_train,y_train):

        y_train = np.ravel(y_train)
        self.intercept_b = 0
        self.coef_m = np.ones(X_train.shape[1])

        for i in range(self.epoch):
            for j in range(0,X_train.shape[0],self.batch_size):
                x_batch = X_train[j:j+self.batch_size]
                y_batch = y_train[j:j+self.batch_size]
```

```
                    y_hat = np.dot(x_batch,self.coef_m) + self.intercept_b

                    der_intercept = -2 * np.mean(y_batch - y_hat)
                    self.intercept_b -= (self.learning_rate * der_intercept)

                    der_coef = -2 * np.dot((y_batch - y_hat),x_batch)/x_batch.shape[0]
                    self.coef_m -= (self.learning_rate * der_coef)

        def predict(self,X_test):
            return np.dot(X_test,self.coef_m) + self.intercept_b


    class OLSRegression:
        def __init__(self):
            self.coef_m = None
            self.intercept_b = None


        def fit(self,X_train,y_train):

            X_train = np.insert(X_train,0,1,axis=1)

            # calculating coeffs
            betas = np.linalg.inv(np.dot(X_train.T,X_train)).dot(X_train.T).dot(y_train)
            self.intercept_b = betas[0]
            self.coef_m = betas[1:]

        def predict(self,X_test):
            return np.dot(X_test,self.coef_m) + self.intercept_b




import numpy as np
from sklearn.datasets import load_diabetes
```

```python
from sklearn.model_selection import train_test_split
from sklearn.metrics import r2_score, mean_squared_error


# Load the Diabetes dataset
data = load_diabetes()
X = data.data
y = data.target

# Ensure there are no NaN values
if np.any(np.isnan(X)) or np.any(np.isnan(y)):
    print("Dataset contains NaN values. Cleaning the data.")
    X = np.nan_to_num(X)
    y = np.nan_to_num(y)

# Split the data into training and testing sets
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)

# Function to evaluate models
def evaluate_model(model, X_train, y_train, X_test, y_test):
    model.fit(X_train, y_train)
    y_pred = model.predict(X_test)

    # Metrics calculation
    r2 = r2_score(y_test, y_pred)
    mse = mean_squared_error(y_test, y_pred)
    n, p = X_test.shape
    adjusted_r2 = 1 - (1 - r2) * (n - 1) / (n - p - 1)

    return mse, r2, adjusted_r2


# Create and evaluate models
gd_model = GradientDescentMethods.GDRegression(learning_rate=0.3, epoch=1000)
sgd_model = GradientDescentMethods.SGDRegression(learning_rate=0.2, epoch=100)
mbgd_model = GradientDescentMethods.MBGDRegression(learning_rate=0.1, epoch=1000, batch_size=20)
```

```python
ols_model = GradientDescentMethods.OLSRegression()

# GD Regression
gd_mse, gd_r2, gd_adj_r2 = evaluate_model(gd_model, X_train, y_train, X_test, y_test)

# SGD Regression
sgd_mse, sgd_r2, sgd_adj_r2 = evaluate_model(sgd_model, X_train, y_train, X_test, y_test)

# MBGD Regression
mbgd_mse, mbgd_r2, mbgd_adj_r2 = evaluate_model(mbgd_model, X_train, y_train, X_test, y_test)

# OLS Regression
ols_mse, ols_r2, ols_adj_r2 = evaluate_model(ols_model, X_train, y_train, X_test, y_test)

# Print results
print("Gradient Descent Regression:")
print(f"MSE: {gd_mse:.4f}, R²: {gd_r2:.4f}, Adjusted R²: {gd_adj_r2:.4f}\n")

print("Stochastic Gradient Descent Regression:")
print(f"MSE: {sgd_mse:.4f}, R²: {sgd_r2:.4f}, Adjusted R²: {sgd_adj_r2:.4f}\n")

print("Mini-Batch Gradient Descent Regression:")
print(f"MSE: {mbgd_mse:.4f}, R²: {mbgd_r2:.4f}, Adjusted R²: {mbgd_adj_r2:.4f}")
print()
print("ols Gradient Descent Regression:")
print(f"MSE: {ols_mse:.4f}, R²: {ols_r2:.4f}, Adjusted R²: {ols_adj_r2:.4f}")
```

```
Gradient Descent Regression:
MSE: 2870.4437, R²: 0.4582, Adjusted R²: 0.3888

Stochastic Gradient Descent Regression:
MSE: 2852.6578, R²: 0.4616, Adjusted R²: 0.3925

Mini-Batch Gradient Descent Regression:
MSE: 2877.5819, R²: 0.4569, Adjusted R²: 0.3872
```

```
ols Gradient Descent Regression:
MSE: 2900.1936, R²: 0.4526, Adjusted R²: 0.3824
```