In [ ]:
```
#This code implements a Logistic Regression model using Stochastic Gradient Descent
import numpy as np

class LogisticRegressionSGD:
    def __init__(self, lr=0.01, epochs=1000, batch_size=32, tol=1e-4):
        self.lr = lr  # Learning rate
        self.epochs = epochs  # Number of epochs
        self.batch_size = batch_size  # Batch size
        self.tol = tol  # Tolerance for convergence
        self.weights = None
        self.bias = None

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

    def predict_proba(self, X):
        z = np.dot(X, self.weights) + self.bias
        return self.sigmoid(z)

    def predict(self, X):
        probabilities = self.predict_proba(X)
        return (probabilities >= 0.5).astype(int)

    def log_loss(self, y_true, y_pred_proba):
        return -np.mean(y_true * np.log(y_pred_proba + 1e-15) + (1 - y_true) * np.l

    def gradient(self, X_batch, y_batch):
        y_pred_proba = self.predict_proba(X_batch)
        error = y_pred_proba - y_batch
        grad_w = np.dot(X_batch.T, error) / X_batch.shape[0]
        grad_b = np.mean(error)
        return grad_w, grad_b

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

        for epoch in range(self.epochs):
            indices = np.random.permutation(n_samples)
            X_shuffled = X[indices]
            y_shuffled = y[indices]

            for i in range(0, n_samples, self.batch_size):
                X_batch = X_shuffled[i:i+self.batch_size]
                y_batch = y_shuffled[i:i+self.batch_size]

                grad_w, grad_b = self.gradient(X_batch, y_batch)
                self.weights -= self.lr * grad_w
                self.bias -= self.lr * grad_b

            if epoch % 100 == 0:
                y_pred_proba = self.predict_proba(X)
                loss = self.log_loss(y, y_pred_proba)
```

```
                print(f"Epoch {epoch}: Loss {loss}")

            if np.linalg.norm(grad_w) < self.tol:
                print("Convergence reached.")
                break

        return self.weights, self.bias
```

In [ ]:
```
import numpy as np

# Define the Logistic Regression SGD class
class LogisticRegressionSGD:
    def __init__(self, lr=0.01, epochs=1000, batch_size=32, tol=1e-3):
        self.lr = lr  # Learning rate
        self.epochs = epochs  # Number of epochs
        self.batch_size = batch_size  # Batch size
        self.tol = tol  # Tolerance for convergence
        self.weights = None
        self.bias = None

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

    def predict_proba(self, X):
        z = np.dot(X, self.weights) + self.bias
        return self.sigmoid(z)

    def predict(self, X):
        probabilities = self.predict_proba(X)
        return (probabilities >= 0.5).astype(int)

    def log_loss(self, y_true, y_pred_proba):
        return -np.mean(y_true * np.log(y_pred_proba + 1e-15) + (1 - y_true) * np.l

    def gradient(self, X_batch, y_batch):
        y_pred_proba = self.predict_proba(X_batch)
        error = y_pred_proba - y_batch
        grad_w = np.dot(X_batch.T, error) / X_batch.shape[0]
        grad_b = np.mean(error)
        return grad_w, grad_b

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

        for epoch in range(self.epochs):
            indices = np.random.permutation(n_samples)
            X_shuffled = X[indices]
            y_shuffled = y[indices]

            for i in range(0, n_samples, self.batch_size):
                X_batch = X_shuffled[i:i + self.batch_size]
                y_batch = y_shuffled[i:i + self.batch_size]

                grad_w, grad_b = self.gradient(X_batch, y_batch)
```

```python
                self.weights -= self.lr * grad_w
                self.bias -= self.lr * grad_b

            if epoch % 100 == 0:
                y_pred_proba = self.predict_proba(X)
                loss = self.log_loss(y, y_pred_proba)
                print(f"Epoch {epoch}: Loss {loss}")

            if np.linalg.norm(grad_w) < self.tol:
                print("Convergence reached.")
                break

        return self.weights, self.bias


# Generate synthetic dataset for logistic regression
np.random.seed(42)
X = np.random.randn(100, 5)
true_weights = np.array([1, -2, 3, -1, 2])
true_bias = -0.5
linear_combination = np.dot(X, true_weights) + true_bias
y = (linear_combination > 0).astype(int)  # Convert to binary labels (0 or 1)

# Create an instance of the Logistic Regression SGD class
model = LogisticRegressionSGD(lr=0.01, epochs=1000, batch_size=32, tol=1e-3)

# Fit the model
weights, bias = model.fit(X, y)

# Predict using the predict method from the model
y_pred = model.predict(X)

# Print results
print("Learned Weights:", weights)
print("Learned Bias:", bias)
print("Predicted Labels:", y_pred)
```

```
Epoch 0: Loss 1.7954775897116972
Epoch 100: Loss 0.6983430514202746
Epoch 200: Loss 0.4011096161255388
Epoch 300: Loss 0.30419907403400676
Epoch 400: Loss 0.2563243041069259
Epoch 500: Loss 0.22783314077525027
Epoch 600: Loss 0.2081548602924787
Epoch 700: Loss 0.19283362361236306
Epoch 800: Loss 0.1808208923658152
Epoch 900: Loss 0.1709733971167161
Learned Weights: [ 1.03186409 -1.60923512  2.14758115 -0.37395479  1.70440633]
Learned Bias: -0.9762304535696413
Predicted Labels: [1 0 0 0 1 1 0 0 0 0 1 1 0 1 0 0 1 0 0 0 0 0 0 0 1 1 0 0 0 0 0 0 1
1 0 1 0
 0 0 0 0 1 0 1 1 0 1 1 0 0 0 0 0 0 1 0 1 0 0 0 1 0 1 0 0 0 1 0 0 0 0 0 0 0
 1 0 0 1 0 1 0 1 0 0 0 1 1 0 0 0 0 0 0 0 1 0 0 0 0 0]
```