Logistic_Regression_usingNP

October 7, 2024

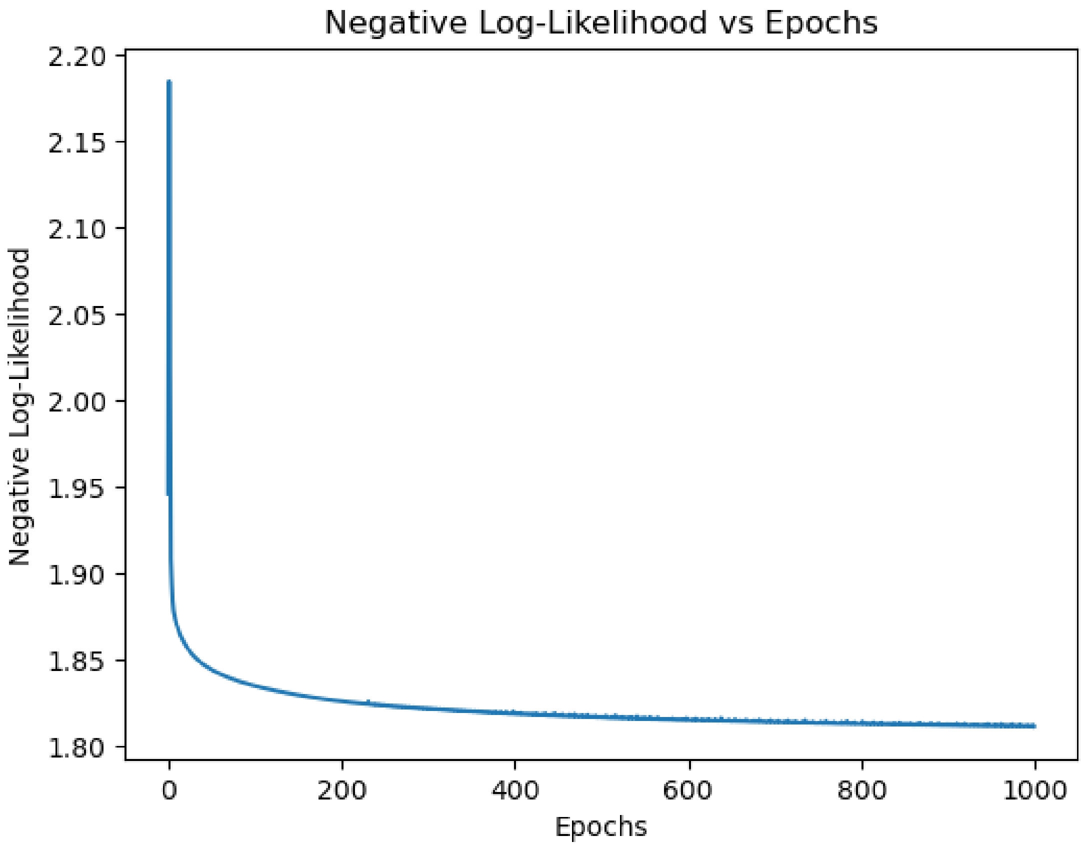
[1]:
```python
import numpy as np

# Step 1: Load the labels
Y_labels = np.genfromtxt('/home/darksst/Desktop/Fall24/
↪StatisticalDecisionTheory/Data/Image/segmentation.data',
                          delimiter=',', dtype=str, encoding=None, usecols=0,␣
↪skip_header=5)

# Load the feature columns (usecols 5, 6, 7, 8, 9 for vedge-mean, vedge-sd,␣
↪hedge-mean, hedge-sd, intensity-mean)
X = np.genfromtxt('/home/darksst/Desktop/Fall24/StatisticalDecisionTheory/Data/
↪Image/segmentation.data',
                   delimiter=',', dtype=float, encoding=None, usecols=(5, 6, 7,␣
↪8, 9), skip_header=5)

# Step 2: One-hot encode the class labels
unique_classes = np.unique(Y_labels)  # Get the unique class names
num_classes = len(unique_classes)

# Create a one-hot encoded matrix for the labels
Y = np.zeros((Y_labels.shape[0], num_classes))
for i, label in enumerate(Y_labels):
    Y[i, np.where(unique_classes == label)[0][0]] = 1

# Initialize the parameter matrix B with zeros
B = np.zeros((X.shape[1], Y.shape[1]))

# Print shapes to verify everything is correct
print(f"Feature matrix (X) shape: {X.shape}")
print(f"One-hot encoded labels (Y) shape: {Y.shape}")
print(f"Parameter matrix (B) shape: {B.shape}")
```

```
Feature matrix (X) shape: (210, 5)
One-hot encoded labels (Y) shape: (210, 7)
Parameter matrix (B) shape: (5, 7)
```

[2]:
```python
# Softmax function for converting logits to probabilities
def softmax(logits):
    exp_logits = np.exp(logits - np.max(logits, axis=1, keepdims=True))
    return exp_logits / np.sum(exp_logits, axis=1, keepdims=True)

# Set hyperparameters
learning_rate = 1e-4
epochs = 1000

# Initialize the parameter matrix B with zeros
B = np.zeros((X.shape[1], Y.shape[1]))

# Initialize array to store the negative log-likelihood at each epoch
neg_log_likelihood = np.zeros(epochs)

# Perform gradient descent
for epoch in range(epochs):
    # Step 1: Compute(Z = X @ B)
    logits = X @ B

    # Step 2: Apply softmax to compute the predicted probabilities
    P = softmax(logits)

    # Step 3: Compute the gradient (X.T @ (Y - P))
    gradient = X.T @ (Y - P)

    # Step 4: Update the parameters (B += learning_rate * gradient)
    B += learning_rate * gradient

    # Step 5: Compute the negative log-likelihood (cross-entropy loss)
    neg_log_likelihood[epoch] = -np.sum(Y * np.log(P + 1e-9)) / Y.shape[0]  #␣
 ↪Adding 1e-9 to avoid log(0)

# Print final parameters and final negative log-likelihood after the last epoch
print("Final parameter matrix (B) after gradient descent:\n", B)
print("Final negative log-likelihood after gradient descent:",␣
 ↪neg_log_likelihood[-1])

# Plot the negative log-likelihood over epochs
import matplotlib.pyplot as plt

plt.plot(range(epochs), neg_log_likelihood)
plt.xlabel('Epochs')
plt.ylabel('Negative Log-Likelihood')
plt.title('Negative Log-Likelihood vs Epochs')
plt.show()
```

```
Final parameter matrix (B) after gradient descent:
```

```
[[-1.11565457e-02  2.80777778e-04  3.64403993e-03 -1.82172075e-02
   3.90212343e-02 -8.07297171e-03 -5.49932713e-03]
 [-1.75492748e-01  2.64499256e-01  5.06411703e-02 -2.29041753e-01
   9.71567001e-02 -8.13986127e-02  7.36359873e-02]
 [ 1.17262116e-01 -5.63569749e-02  1.02660958e-01  1.77358074e-01
  -1.22998160e-01 -2.11380870e-01 -6.54514362e-03]
 [ 2.06150606e-01 -1.26913273e-01 -2.66915155e-02  1.54266945e-01
   6.80851435e-02  1.37868218e-01 -4.12766125e-01]
 [-2.48712794e-01  8.75024745e-02  7.32487509e-02 -3.59891457e-02
   8.51001649e-02 -8.75500746e-02  1.26400624e-01]]
Final negative log-likelihood after gradient descent: 1.8115049646864227
```



Logistic_Regression_UsingPytorch

October 7, 2024

[1]:
```python
import numpy as np

# Step 1: Load the labels (first column) and features
Y_labels = np.genfromtxt('/home/darksst/Desktop/Fall24/
↪StatisticalDecisionTheory/Data/Image/segmentation.data',
                         delimiter=',', dtype=str, encoding=None, usecols=0,␣
↪skip_header=5)

# Load the feature columns (usecols 5, 6, 7, 8, 9 for vedge-mean, vedge-sd,␣
↪hedge-mean, hedge-sd, intensity-mean)
X = np.genfromtxt('/home/darksst/Desktop/Fall24/StatisticalDecisionTheory/Data/
↪Image/segmentation.data',
                  delimiter=',', dtype=float, encoding=None, usecols=(5, 6, 7,␣
↪8, 9), skip_header=5)

# Step 2: One-hot encode the class labels
unique_classes = np.unique(Y_labels)  # Get the unique class names
num_classes = len(unique_classes)

# Create a one-hot encoded matrix for the labels
Y = np.zeros((Y_labels.shape[0], num_classes))
for i, label in enumerate(Y_labels):
    Y[i, np.where(unique_classes == label)[0][0]] = 1

# Initialize the parameter matrix B with zeros
B = np.zeros((X.shape[1], Y.shape[1]))

# Print shapes to verify everything is correct
print(f"Feature matrix (X) shape: {X.shape}")
print(f"One-hot encoded labels (Y) shape: {Y.shape}")
print(f"Parameter matrix (B) shape: {B.shape}")
```

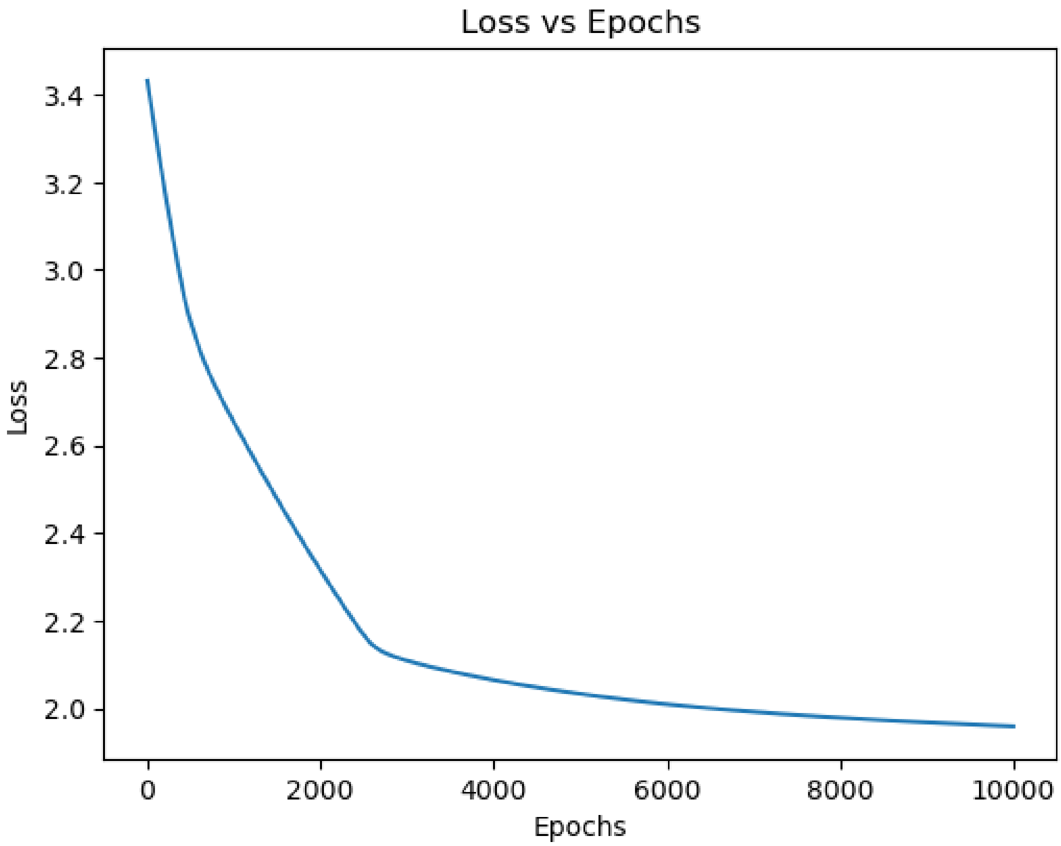
```
Feature matrix (X) shape: (210, 5)
One-hot encoded labels (Y) shape: (210, 7)
Parameter matrix (B) shape: (5, 7)
```

[2]:

```python
import torch
import torch.nn as nn
import torch.optim as optim
import matplotlib.pyplot as plt

# Convert the NumPy arrays to PyTorch tensors
X_tensor = torch.tensor(X, dtype=torch.float32)  # Feature matrix
Y_tensor = torch.tensor(Y, dtype=torch.float32)  # One-hot encoded labels

# Define the Logistic Regression model using PyTorch
class LogisticRegression(nn.Module):
    def __init__(self, dimension_input, dimension_output):
        super(LogisticRegression, self).__init__()
        self.linear = nn.Linear(dimension_input, dimension_output)

    def forward(self, x):
        # Forward pass (logits)
        return self.linear(x)

# Set the input and output dimensions
dimension_input = X_tensor.shape[1]   # Number of features
dimension_output = Y_tensor.shape[1]  # Number of classes (one-hot encoding)

# Initialize the model
model = LogisticRegression(dimension_input, dimension_output)

# Define the loss function (CrossEntropyLoss handles softmax + loss internally)
criterion = nn.CrossEntropyLoss()

# Define the optimizer
optimizer = optim.SGD(model.parameters(), lr=0.0001)

# Number of epochs
epochs = 10000

# Initialize a list to store the loss values for plotting
loss_values = []

# Training loop
for epoch in range(epochs):
    # Forward pass: compute logits
    logits = model(X_tensor)

    # Compute the loss (CrossEntropyLoss expects raw logits, no need for
↪softmax)
    loss = criterion(logits, torch.max(Y_tensor, 1)[1])  # Convert Y_tensor
↪from one-hot to class labels
```

```python
    # Zero the gradients from the previous step
    optimizer.zero_grad()

    # Backward pass: compute gradients
    loss.backward()

    # Update the model parameters
    optimizer.step()

    # Store the loss value for plotting
    loss_values.append(loss.item())

# Print the final model parameters
print("Final parameters after training:", model.linear.weight, model.linear.
 ↪bias)
print("Final Loss:", loss_values[-1])

# Plot the loss over epochs using Matplotlib
plt.plot(range(epochs), loss_values)
plt.xlabel('Epochs')
plt.ylabel('Loss')
plt.title('Loss vs Epochs')
plt.show()
```

```
Final parameters after training: Parameter containing:
tensor([[-0.2686,  0.0679, -0.0367,  0.3721, -0.1195],
        [ 0.3704,  0.3619, -0.2582,  0.1548,  0.1294],
        [-0.1458,  0.1476,  0.0107,  0.0808,  0.1370],
        [-0.2795,  0.1655, -0.0229,  0.1832,  0.0687],
        [ 0.3183, -0.0171, -0.1804,  0.2289,  0.1424],
        [ 0.3996, -0.1315,  0.2426, -0.2876, -0.1482],
        [-0.3680,  0.1303, -0.0515, -0.0277,  0.1631]], requires_grad=True)
Parameter containing:
tensor([-0.1911, -0.0436,  0.3000, -0.0807,  0.2985, -0.1704, -0.2639],
       requires_grad=True)
Final Loss: 1.958688735961914
```


Loss vs Epochs
Loss
3.4
3.2
3.0
2.8
2.6
2.4
2.2
2.0
0
2000
4000
6000
8000
10000
Epochs

Regression_Class

October 7, 2024

[1]:

```
import numpy as np
import matplotlib.pyplot as plt

class RegressionModel:
    def __init__(self, X, Y):
        # Add a constant 1 column for the intercept term
        self.X = np.hstack((np.ones((X.shape[0], 1)), X))

        # Store the response matrix
        self.Y = Y

        # Initialize parameter matrix with zeros
        self.B = np.zeros((self.X.shape[1], self.Y.shape[1] if len(self.Y.
 ↪shape) > 1 else 1))

        # Initialize the loss attribute
        self.loss = np.array([])

    # Function to plot the loss
    def plot_loss(self):
        plt.plot(self.loss)
        plt.xlabel('Iterations')
        plt.ylabel('Loss')
        plt.title('Loss over iterations')
        plt.show()
```

[2]:

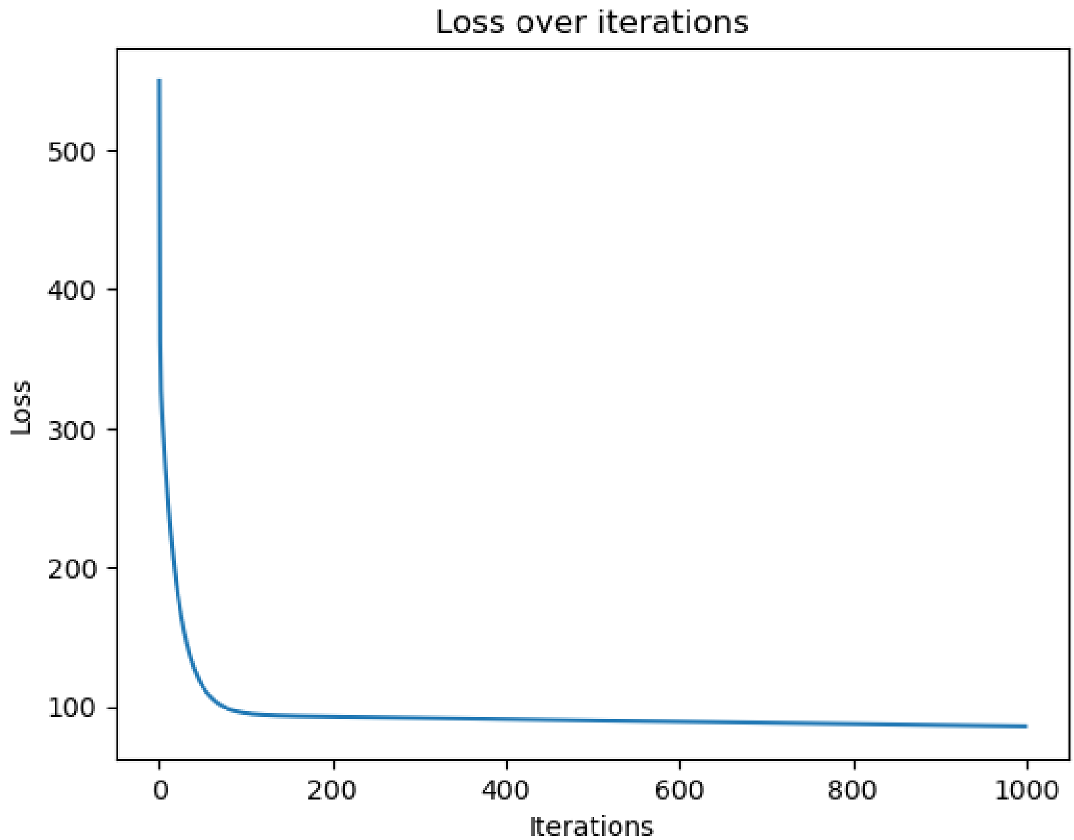
```
class LinearRegressionModel(RegressionModel):
    def __init__(self, X, Y):
        # Call the parent class initializer to inherit its properties
        super().__init__(X, Y)

    # Function to perform gradient descent
    def gradient_descent(self, alpha, n_iterations):
        m = self.Y.shape[0]  # Number of samples
        MSE = np.zeros(n_iterations)  # Array to store MSE at each epoch

        for epoch in range(n_iterations):
```

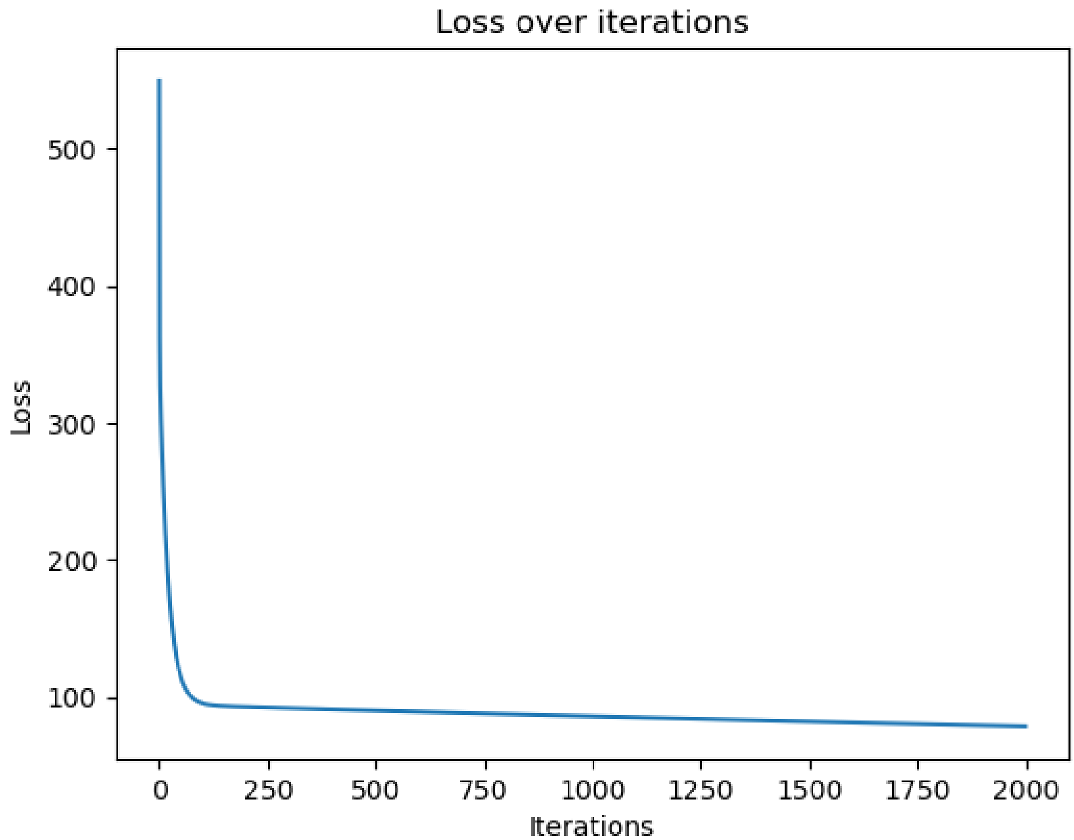
```
            # Compute the gradient and update the parameters
            self.B += alpha * self.X.T @ (self.Y - self.X @ self.B)

            # Compute the MSE for the current iteration
            MSE[epoch] = (1 / (self.Y.shape[1] * m)) * np.sum((self.Y - (self.X␣
↪@ self.B))**2)

        # Append the MSE values to the loss attribute in the parent class
        self.loss = np.append(self.loss, MSE)
```

[3]:
```
# Logistic Regression Model (inherits from RegressionModel)
class LogisticRegressionModel(RegressionModel):
    def __init__(self, X, Y):
        # Call the parent class initializer to inherit its properties
        super().__init__(X, Y)

    # Softmax function to convert logits to probabilities
    def softmax(self, logits):
        exp_logits = np.exp(logits - np.max(logits, axis=1, keepdims=True))  #␣
↪Numerical stability
        return exp_logits / np.sum(exp_logits, axis=1, keepdims=True)

    # Function to perform gradient descent for logistic regression
    def gradient_descent(self, alpha, n_iterations):
        neg_log_likelihood = np.zeros(n_iterations)  # Array to store NLL at␣
↪each epoch

        for epoch in range(n_iterations):
            # Step 1: Compute logits (Z = X @ B)
            logits = self.X @ self.B

            # Step 2: Apply softmax to compute the predicted probabilities
            P = self.softmax(logits)

            # Step 3: Compute the gradient (X.T @ (Y - P))
            gradient = self.X.T @ (self.Y - P)

            # Step 4: Update the parameters (B += learning_rate * gradient)
            self.B += alpha * gradient

            # Step 5: Compute the negative log-likelihood (cross-entropy loss)
            neg_log_likelihood[epoch] = -np.sum(self.Y * np.log(P + 1e-9)) /␣
↪self.Y.shape[0]  # Adding epsilon for numerical stability

        # Store the NLL loss after all iterations
        self.loss = np.append(self.loss, neg_log_likelihood)
```

[4]:
```
##########################HOMEWORK 1#########################

# Load the data for Homework 1 Problem 3
data = np.genfromtxt('/home/darksst/Desktop/Fall24/StatisticalDecisionTheory/
 ↪Data/gt_data/gt_2015.csv', skip_header=1, delimiter=',', usecols=(0, 3, 8,␣
 ↪9, 10))

# Clean the data (remove rows with missing values)
data_clean = data

# Extract the response variables (Y): CO (column 3) and NOx (column 4)
Y = data_clean[:, [3, 4]]

# Extract the predictor variables (X) and add a column of ones for the intercept
X = data_clean[:, [0, 1, 2]]

# Print shapes to verify
print("Shape of X:", X.shape)
print("Shape of Y:", Y.shape)

# Test the LinearRegressionModel class

# Initialize the LinearRegressionModel
lin_model = LinearRegressionModel(X, Y)

# Perform 1,000 epochs of gradient descent with learning rate 1e-4
lin_model.gradient_descent(alpha=17e-8, n_iterations=1000)

# Plot the loss after 1,000 epochs
lin_model.plot_loss()

# Perform an additional 1,000 epochs of gradient descent
lin_model.gradient_descent(alpha=17e-8, n_iterations=1000)

# Plot the loss after 2,000 total epochs
lin_model.plot_loss()
```

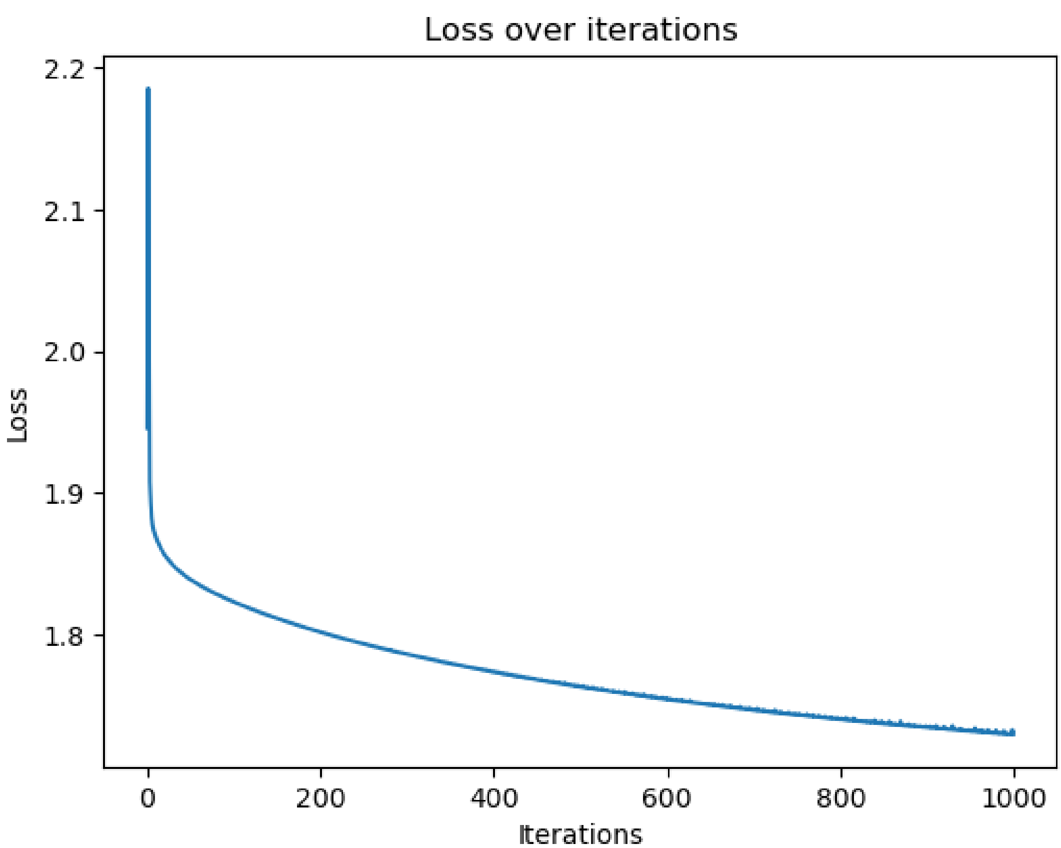
```
Shape of X: (7384, 3)
Shape of Y: (7384, 2)
```

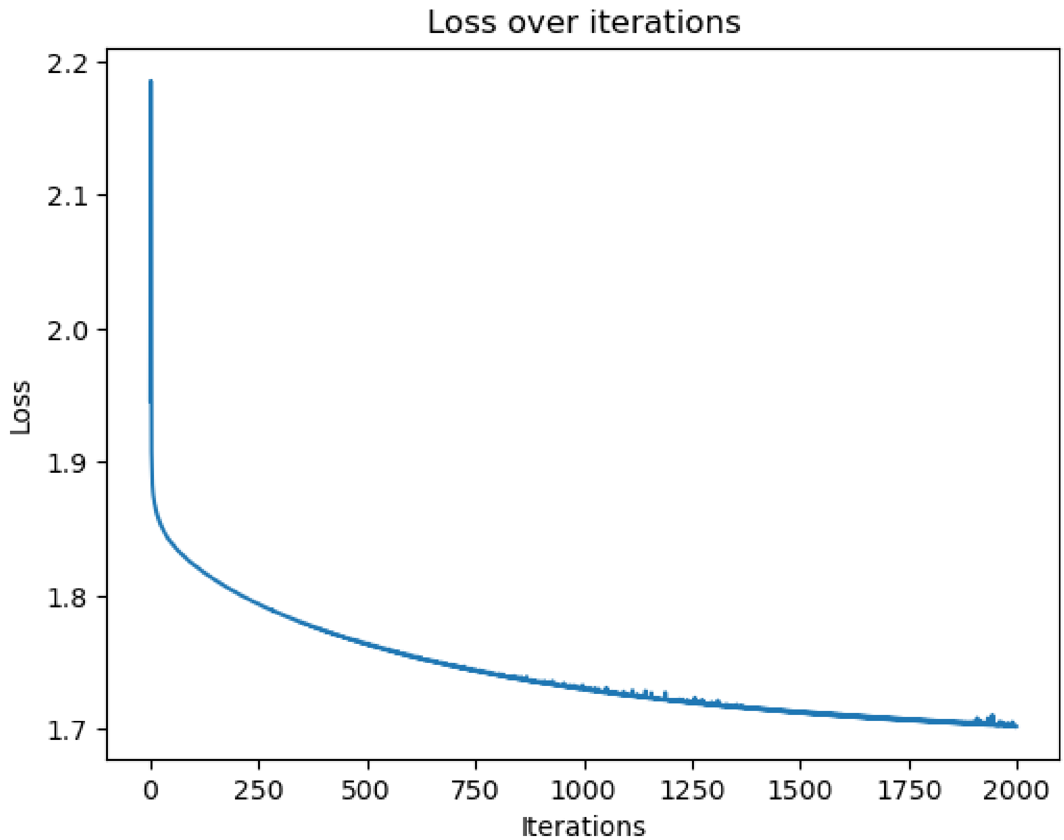

Loss over iterations
Loss
500
400
300
200
100
0
200
400
600
800
1000
Iterations



[5]:
```
#########################HOMEWORK 3#########################
# Load the segmentation dataset
labels = np.genfromtxt('/home/darksst/Desktop/Fall24/StatisticalDecisionTheory/
↪Data/Image/segmentation.data',
                       delimiter=',', dtype=str, encoding=None, usecols=0,␣
↪skip_header=6)

# Load the feature columns
features = np.genfromtxt('/home/darksst/Desktop/Fall24/
↪StatisticalDecisionTheory/Data/Image/segmentation.data',
                         delimiter=',', dtype=float, encoding=None, usecols=(5,␣
↪6, 7, 8, 9), skip_header=6)

# One-hot encode the class labels
unique_classes = np.unique(labels)  # Get the unique class names
num_classes = len(unique_classes)

# Create a one-hot encoded matrix for the labels
Y = np.zeros((labels.shape[0], num_classes))
```

```
for i, label in enumerate(labels):
    Y[i, np.where(unique_classes == label)[0][0]] = 1

# Set X to be the feature matrix (already loaded above as 'features')
X = features

# Test the LogisticRegressionModel class

# Initialize the LogisticRegressionModel
log_model = LogisticRegressionModel(X, Y)

# Perform 1,000 epochs of gradient descent with learning rate 1e-4
log_model.gradient_descent(alpha=1e-4, n_iterations=1000)

# Plot the loss after 1,000 epochs
log_model.plot_loss()

# Perform an additional 1,000 epochs of gradient descent
log_model.gradient_descent(alpha=1e-4, n_iterations=1000)

# Plot the loss after 2,000 total epochs
log_model.plot_loss()
```




Loss over iterations
2.2
2.1
2.0
1.9
1.8
1.7
Loss
0
250
500
750
1000
1250
1500
1750
2000
Iterations