SSY316 Assignment 3

Edoardo Mangia
edoardom@student.chalmers.se

Nuree Kim
nuree@student.chalmers.se

November 2024

Contents

1 Exercise 1 **1**

2 Exercise 2 **2**

3 Exercise 3 **4**

4 Exercise 4 **13**

5 Exercise 5 **25**

6 Extra Credit **28**

1 Exercise 1

Suppose we collect data from a group of students in a Machine learning class with variables x_1 = hours studied, x_2 = grade point average, and y = a binary output if that student received grade 5 ($y = 1$) or not ($y = 0$). We learn a logistic regression model:

$$p(y = 1|x) = \frac{e^{\beta_0+\beta_1 x_1+\beta_2 x_2}}{1 + e^{\beta_0+\beta_1 x_1+\beta_2 x_2}},$$

with parameters $\hat{\beta}_0 = -6$, $\hat{\beta}_1 = 0.05$, and $\hat{\beta}_2 = 1$.

Q1

Estimate the probability according to the logistic regression model that a student who studies for 40 h and has the grade point average of 3.5 gets a 5 in the Machine learning class.

Solution

The logistic regression model is:

$$P(y=1|x) = \frac{\exp(\beta_0 + \beta_1 x_1 + \beta_2 x_2)}{1 + \exp(\beta_0 + \beta_1 x_1 + \beta_2 x_2)}.$$

We are given:

- $\hat{\beta}_0 = -6$,
- $\hat{\beta}_1 = 0.05$,
- $\hat{\beta}_2 = 1$,
- $x_1 = 40$ (hours studied),
- $x_2 = 3.5$ (grade point average).

Compute the linear combination z:

$$z = \beta_0 + \beta_1 x_1 + \beta_2 x_2$$

$$z = -6 + 0.05 \cdot 40 + 1 \cdot 3.5$$

$$z = -6 + 2 + 3.5 = -0.5$$

Calculate the probability:

$$P(y=1|x) = \frac{\exp(z)}{1 + \exp(z)} = \frac{\exp(-0.5)}{1 + \exp(-0.5)}$$

Simplify the exponentials: The value of $\exp(-0.5)$ is approximately 0.6065:

$$P(y=1|x) = \frac{0.6065}{1 + 0.6065} = \frac{0.6065}{1.6065} \approx 0.377$$

The estimated probability that the student receives grade 5 is therefore approximately 37.7%

2 Exercise 2

Q1

Let $\sigma(a) = \frac{1}{1+e^{-a}}$ be the sigmoid function. Show that

$$\frac{d\sigma(a)}{da} = \sigma(a)(1 - \sigma(a)).$$

Solution

Differentiating $\sigma(a)$ with respect to a:

$$\frac{d\sigma(a)}{da} = \frac{d}{da}\left(\frac{1}{1+e^{-a}}\right) = \frac{-\frac{d}{da}(1+e^{-a})}{(1+e^{-a})^2}.$$

The derivative of e^{-a} is $-e^{-a}$, so:

$$\frac{d\sigma(a)}{da} = \frac{-(-e^{-a})}{(1+e^{-a})^2} = \frac{e^{-a}}{(1+e^{-a})^2}$$

Rewrite in terms of $\sigma(a)$: Using $\sigma(a) = \frac{1}{1+e^{-a}}$, we find:

$$\frac{d\sigma(a)}{da} = \sigma(a)(1-\sigma(a))$$

Q2

Using the previous result and the chain rule of calculus, derive an expression for the gradient of the log likelihood for logistic regression.

Solution

The logistic regression model is:

$$P(y=1|x) = \sigma(a), \quad \text{where } a = \beta^\top x.$$

Log-Likelihood:

$$\ln L(\beta) = \sum_{i=1}^{N} \left[y_i \ln \sigma(a_i) + (1-y_i)\ln(1-\sigma(a_i))\right].$$

Gradient of the Log-Likelihood:

$$\frac{\partial}{\partial\beta} \ln L(\beta) = \sum_{i=1}^{N} \frac{\partial}{\partial\beta}\left[y_i \ln \sigma(a_i) + (1-y_i)\ln(1-\sigma(a_i))\right].$$

Simplifying each term:

1. For $y_i \ln \sigma(a_i)$:

$$\frac{\partial}{\partial\beta} \ln \sigma(a_i) = (1-\sigma(a_i))x_i.$$

2. For $(1-y_i)\ln(1-\sigma(a_i))$:

$$\frac{\partial}{\partial\beta} \ln(1-\sigma(a_i)) = -\sigma(a_i)x_i.$$

Combining:

$$\frac{\partial}{\partial \beta} \ln L(\beta) = \sum_{i=1}^{N} (y_i - \sigma(a_i))\, x_i.$$

3 Exercise 3

Q1 - Maximum Likelihood for Class Probabilities

Show that the maximum likelihood estimate for the class probabilities π is given by:

$$\pi_c = \frac{N_c}{N},$$

where N_c is the number of data points assigned to class c, and N is the total number of data points.

Solution

Assume we have N data points and each data point y_i belongs to one of C classes. The likelihood of observing the data given the probabilities $\pi_1, \pi_2, \ldots, \pi_C$ is:

$$L(\pi) = \prod_{i=1}^{N} \pi_{y_i},$$

where π_{y_i} is the probability of class y_i.

Taking the natural logarithm of the likelihood to simplify the computation:

$$\ln L(\pi) = \sum_{i=1}^{N} \ln \pi_{y_i}.$$

Group terms in the summation by class. Let N_c denote the number of points belonging to class c:

$$\ln L(\pi) = \sum_{c=1}^{C} N_c \ln \pi_c.$$

Since the class probabilities must sum to 1:

$$\sum_{c=1}^{C} \pi_c = 1.$$

This is a constraint on the optimization problem.

To maximize $\ln L(\pi)$ under the constraint $\sum_{c=1}^{C} \pi_c = 1$, we use the method of Lagrange multipliers. Define the Lagrangian:

$$\mathcal{L}(\pi, \lambda) = \sum_{c=1}^{C} N_c \ln \pi_c + \lambda \left(1 - \sum_{c=1}^{C} \pi_c \right).$$

Take the derivative of $\mathcal{L}$ with respect to π_c and set it to zero:

$$\frac{\partial \mathcal{L}}{\partial \pi_c} = \frac{N_c}{\pi_c} - \lambda = 0.$$

Rearranging gives:

$$\pi_c = \frac{N_c}{\lambda}.$$

Using the constraint $\sum_{c=1}^{C} \pi_c = 1$, we find:

$$\sum_{c=1}^{C} \frac{N_c}{\lambda} = 1,$$

which implies:

$$\lambda = \sum_{c=1}^{C} N_c = N.$$

Substituting $\lambda = N$ into $\pi_c = \frac{N_c}{\lambda}$:

$$\pi_c = \frac{N_c}{N}.$$

Thus, the maximum likelihood estimate for π_c is $\pi_c = \frac{N_c}{N}$.

Q2 - Class-Conditional Densities with Shared Covariance Matrix (LDA)

Assuming the class-conditional densities follow a Gaussian distribution with a shared covariance matrix:

$$p(x|y = c, \theta) = \mathcal{N}(x|\mu_c, \Sigma),$$

derive the following maximum likelihood estimates:

a) The mean for each class c:

$$\mu_c = \frac{1}{N_c} \sum_{n=1}^{N} \nVdash_{y(n)=c} x^{(n)}.$$

b) The shared covariance matrix Σ:

$$\Sigma = \frac{1}{N} \sum_{c=1}^{C} N_c S_c,$$

where

$$S_c = \frac{1}{N_c} \sum_{n=1}^{N} \nVdash_{y(n)=c} (x^{(n)} - \mu_c)(x^{(n)} - \mu_c)^T.$$

Solution

Assuming the class-conditional densities follow a Gaussian distribution with a shared covariance matrix:

$$p(x|y = c, \theta) = \mathcal{N}(x|\mu_c, \Sigma),$$

we derive the following maximum likelihood estimates.

a) The mean for each class c.

The class-conditional density for class c is:

$$p(x|y = c, \theta) = \mathcal{N}(x|\mu_c, \Sigma),$$

where μ_c is the mean vector for class c, and Σ is the shared covariance matrix.

The likelihood of the data assigned to class c is:

$$L_c(\mu_c) = \prod_{n=1}^{N} \mathcal{N}(x^{(n)}|\mu_c, \Sigma)^{y^{(n)}=c}.$$

Taking the logarithm:

$$\ln L_c(\mu_c) = \sum_{n=1}^{N} (y^{(n)} = c) \ln \mathcal{N}(x^{(n)}|\mu_c, \Sigma).$$

The Gaussian density is:

$$\mathcal{N}(x|\mu_c, \Sigma) = \frac{1}{(2\pi)^{d/2}|\Sigma|^{1/2}} \exp\left(-\frac{1}{2}(x - \mu_c)^\top \Sigma^{-1}(x - \mu_c)\right).$$

Ignoring terms that are constant with respect to μ_c, the log-likelihood becomes:

$$\ln L_c(\mu_c) \propto -\frac{1}{2} \sum_{n=1}^{N} (y^{(n)} = c)(x^{(n)} - \mu_c)^\top \Sigma^{-1}(x^{(n)} - \mu_c).$$

To maximize $\ln L_c(\mu_c)$, take its derivative with respect to μ_c and set it to zero:

$$\frac{\partial}{\partial \mu_c} \ln L_c(\mu_c) \propto \sum_{n=1}^{N} (y^{(n)} = c)\Sigma^{-1}(x^{(n)} - \mu_c) = 0.$$

Simplifying gives:

$$\sum_{n=1}^{N} (y^{(n)} = c)x^{(n)} = \sum_{n=1}^{N} (y^{(n)} = c)\mu_c.$$

Let $N_c = \sum_{n=1}^{N}(y^{(n)} = c)$, the number of points in class c. Then:

$$\mu_c = \frac{1}{N_c}\sum_{n=1}^{N}(y^{(n)} = c)x^{(n)}.$$

This gives:

$$\mu_c = \frac{1}{N_c}\sum_{n=1}^{N}x^{(n)}, \quad \text{for } y^{(n)} = c.$$

b) The shared covariance matrix Σ.

The joint likelihood for all classes is:

$$L(\mu_c, \Sigma) = \prod_{c=1}^{C}\prod_{n=1}^{N}\mathcal{N}(x^{(n)}|\mu_c, \Sigma)^{y^{(n)}=c}.$$

Taking the logarithm and simplifying (ignoring constant terms):

$$\ln L(\mu_c, \Sigma) \propto -\frac{1}{2}\sum_{c=1}^{C}\sum_{n=1}^{N}(y^{(n)} = c)\left(\ln|\Sigma| + (x^{(n)} - \mu_c)^\top\Sigma^{-1}(x^{(n)} - \mu_c)\right).$$

To maximize the log-likelihood with respect to Σ, take its derivative and set it to zero:

$$\frac{\partial}{\partial\Sigma}\ln L(\mu_c, \Sigma) \propto -\frac{1}{2}\sum_{c=1}^{C}\sum_{n=1}^{N}(y^{(n)} = c)\left[\Sigma^{-1} - \Sigma^{-1}(x^{(n)} - \mu_c)(x^{(n)} - \mu_c)^\top\Sigma^{-1}\right] = 0.$$

Simplifying gives:

$$\Sigma = \frac{1}{N}\sum_{c=1}^{C}\sum_{n=1}^{N}(y^{(n)} = c)(x^{(n)} - \mu_c)(x^{(n)} - \mu_c)^\top.$$

Let S_c denote the within-class scatter matrix for class c:

$$S_c = \frac{1}{N_c}\sum_{n=1}^{N}(y^{(n)} = c)(x^{(n)} - \mu_c)(x^{(n)} - \mu_c)^\top.$$

Then the shared covariance matrix Σ becomes:

$$\Sigma = \sum_{c=1}^{C}\frac{N_c}{N}S_c.$$

Q3 - Class-Conditional Densities with Separate Covariance Matrices (QDA)

Assuming the class-conditional densities follow Gaussian distributions with separate covariance matrices:

$$p(x|y = c, \theta) = \mathcal{N}(x|\mu_c, \Sigma_c),$$

derive the maximum likelihood estimate for the class-specific covariance matrix:

$$\Sigma_c = \frac{1}{N_c} \sum_{n=1}^{N} \nVdash_{y(n)=c} (x^{(n)} - \mu_c)(x^{(n)} - \mu_c)^T.$$

Solution

We assume that the class-conditional densities follow Gaussian distributions with separate covariance matrices:

$$p(x|y = c, \theta) = \mathcal{N}(x|\mu_c, \Sigma_c),$$

where μ_c is the mean of class c, and Σ_c is the covariance matrix for class c.

The likelihood of the data points assigned to class c is given by:

$$L(\Sigma_c) = \prod_{n=1}^{N} \mathcal{N}(x^{(n)}|\mu_c, \Sigma_c)^{y^{(n)}=c}.$$

The Gaussian density function is:

$$\mathcal{N}(x|\mu_c, \Sigma_c) = \frac{1}{(2\pi)^{d/2}|\Sigma_c|^{1/2}} \exp\left(-\frac{1}{2}(x - \mu_c)^\top \Sigma_c^{-1}(x - \mu_c)\right).$$

Taking the natural logarithm of the likelihood function:

$$\ln L(\Sigma_c) = \sum_{n=1}^{N} (y^{(n)} = c) \ln \mathcal{N}(x^{(n)}|\mu_c, \Sigma_c).$$

Substituting the expression for $\ln \mathcal{N}(x|\mu_c, \Sigma_c)$:

$$\ln L(\Sigma_c) = \sum_{n=1}^{N} (y^{(n)} = c) \left[-\frac{d}{2}\ln(2\pi) - \frac{1}{2}\ln|\Sigma_c| - \frac{1}{2}(x^{(n)} - \mu_c)^\top \Sigma_c^{-1}(x^{(n)} - \mu_c)\right].$$

Simplifying, we get:

$$\ln L(\Sigma_c) = -\frac{N_c d}{2}\ln(2\pi) - \frac{N_c}{2}\ln|\Sigma_c| - \frac{1}{2}\sum_{n=1}^{N}(y^{(n)} = c)(x^{(n)} - \mu_c)^\top \Sigma_c^{-1}(x^{(n)} - \mu_c).$$

To maximize $\ln L(\Sigma_c)$, take the derivative with respect to Σ_c^{-1} (noting that $\partial \ln |\Sigma_c| / \partial \Sigma_c^{-1} = \Sigma_c$):

$$\frac{\partial}{\partial \Sigma_c^{-1}} \ln L(\Sigma_c) = \frac{N_c}{2} \Sigma_c - \frac{1}{2} \sum_{n=1}^{N} (y^{(n)} = c)(x^{(n)} - \mu_c)(x^{(n)} - \mu_c)^\top = 0.$$

Rearranging:

$$\Sigma_c = \frac{1}{N_c} \sum_{n=1}^{N} (y^{(n)} = c)(x^{(n)} - \mu_c)(x^{(n)} - \mu_c)^\top.$$

The maximum likelihood estimate for the class-specific covariance matrix is:

$$\Sigma_c = \frac{1}{N_c} \sum_{n=1}^{N} (y^{(n)} = c)(x^{(n)} - \mu_c)(x^{(n)} - \mu_c)^\top.$$

Q4 - Model Implementation

Implement the LDA and QDA fit function to learn $\mu_0, \mu_1, \Sigma_0, \Sigma_1, \pi, w_0, w_1$.

Solution

This section implements Linear Discriminant Analysis (LDA) and Quadratic Discriminant Analysis (QDA) for binary classification. Both models are designed to classify data points into one of two classes by learning the parameters from the training data.

LDA Implementation: Below is the Python implementation of the LDA fit and predict functions:

```python
class LDA:
    def __init__(self, num_features=2) -> None:
        self.num_features = num_features
        self.w = np.ones((num_features))  # Weight vector
        self.w0 = 0  # Bias term
        self.pi = 0.5  # Prior probability
        self.mean_class0 = np.zeros((num_features))  # Mean of class 0
        self.mean_class1 = np.ones((num_features))  # Mean of class 1
        self.cov = np.eye(num_features)  # Shared covariance matrix

    def fit(self, X, y) -> None:
        n_class0 = sum(y == 0)
        n_class1 = sum(y == 1)
        n_total = len(y)
        self.pi = n_class1 / n_total
        self.mean_class0 = X[y == 0].mean(axis=0)
        self.mean_class1 = X[y == 1].mean(axis=0)
        self.cov = np.cov(X, rowvar=False)
        self.w = np.linalg.inv(self.cov).dot(self.mean_class1 -
            self.mean_class0)
        self.w0 = (
            -0.5 * (self.mean_class1.T @ np.linalg.inv(self.cov) @
                self.mean_class1)
            + 0.5 * (self.mean_class0.T @ np.linalg.inv(self.cov)
                @ self.mean_class0)
            + np.log(self.pi / (1 - self.pi))
        )

    def predict(self, X) -> np.ndarray:
        return (X @ self.w + self.w0 > 0).astype(int)
```

Listing 1: LDA Implementation

QDA Implementation: Below is the Python implementation of the QDA fit and predict functions:

```
class QDA:
    def __init__(self, num_features=2) -> None:
        self.num_features = num_features
        self.pi = 0.5
        self.mean_class0 = np.zeros((num_features))
        self.mean_class1 = np.ones((num_features))
        self.cov_class0 = np.eye(num_features)
        self.cov_class1 = np.eye(num_features)

    def fit(self, X, y) -> None:
        n_class0 = np.sum(y == 0)
        n_class1 = np.sum(y == 1)
        n_total = len(y)
        self.pi = n_class1 / n_total
        self.mean_class0 = X[y == 0].mean(axis=0)
        self.mean_class1 = X[y == 1].mean(axis=0)
        self.cov_class0 = np.cov(X[y == 0], rowvar=False)
        self.cov_class1 = np.cov(X[y == 1], rowvar=False)

    def predict(self, X) -> np.ndarray:
        inv_cov_class0 = np.linalg.inv(self.cov_class0)
        inv_cov_class1 = np.linalg.inv(self.cov_class1)
        det_cov_class0 = np.linalg.det(self.cov_class0)
        det_cov_class1 = np.linalg.det(self.cov_class1)
        def discriminant(x, mean, inv_cov, det_cov, pi):
            diff = x - mean
            return (
                -0.5 * diff.T @ inv_cov @ diff
                - 0.5 * np.log(det_cov)
                + np.log(pi)
            )
        scores = np.array([
            discriminant(x, self.mean_class1, inv_cov_class1,
                det_cov_class1, self.pi)
            - discriminant(x, self.mean_class0, inv_cov_class0,
                det_cov_class0, 1 - self.pi)
            for x in X
        ])
        return (scores > 0).astype(int)
```

Listing 2: QDA Implementation

Performance Evaluation: The accuracy of both models is computed using synthetic data:

```
X, y = generate_data()

lda = LDA()
lda.fit(X, y)
lda_acc = lda.calc_accuracy(X, y)
print(f"LDA accuracy: {lda_acc}")

qda = QDA()
qda.fit(X, y)
qda_acc = qda.calc_accuracy(X, y)
print(f"QDA accuracy: {qda_acc}")
```

Listing 3: Model Evaluation

The decision boundaries and parameters are visualized to understand the learned models.

Q5 - Performance Comparison

Evaluate the performance of LDA and QDA by calculating accuracy on the test set. Discuss the differences between the decision boundaries of LDA and QDA and their behavior under different class distributions.

Solution

In this section, we evaluate the performance of Linear Discriminant Analysis (LDA) and Quadratic Discriminant Analysis (QDA) by calculating their accuracy on a test set. Additionally, we discuss the differences in their decision boundaries and behavior under varying class distributions.

Accuracy Evaluation: The following Python code calculates the accuracy of LDA and QDA on synthetic test data:

```
# Generate synthetic data and split into training and testing sets
from sklearn.model_selection import train_test_split

X, y = generate_data()
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.3, random_state=42)

# Fit LDA model
lda = LDA()
lda.fit(X_train, y_train)
```

```python
lda_acc = lda.calc_accuracy(X_test, y_test)
print(f"LDA Accuracy: {lda_acc}")

# Fit QDA model
qda = QDA()
qda.fit(X_train, y_train)
qda_acc = qda.calc_accuracy(X_test, y_test)
print(f"QDA Accuracy: {qda_acc}")
```

Listing 4: Accuracy Evaluation

Comparison of Decision Boundaries:

- **LDA:**
  - LDA assumes shared covariance between classes, resulting in a linear decision boundary.
  - This assumption simplifies the model and is effective when the true decision boundary is approximately linear.
- **QDA:**
  - QDA models separate covariance matrices for each class, resulting in a quadratic decision boundary.
  - This flexibility allows QDA to handle non-linear boundaries but increases the risk of overfitting, particularly for small datasets.

Behavior Under Different Class Distributions:

- **LDA:**
  - LDA performs well when class distributions have similar covariances.
  - It may fail when the decision boundary is non-linear, as it is restricted to a linear form.
- **QDA:**
  - QDA is effective for non-linear boundaries and when the class distributions have distinct covariances.
  - However, QDA may struggle with limited data due to the need to estimate more parameters.

Visualization of Decision Boundaries: Below is the Python code to visualize the decision boundaries of LDA and QDA:

```python
import matplotlib.pyplot as plt

plt.figure(figsize=(12, 6))

# Plot LDA Decision Boundary
```

```python
plt.subplot(1, 2, 1)
plt.scatter(X_test[:, 0], X_test[:, 1], c=y_test, edgecolors="k",
    cmap=plt.cm.coolwarm)
lda.plot_params()
plot_decision_boundaries(lda)
plt.title("LDA Decision Boundary")
plt.xlabel("Feature 1")
plt.ylabel("Feature 2")

# Plot QDA Decision Boundary
plt.subplot(1, 2, 2)
plt.scatter(X_test[:, 0], X_test[:, 1], c=y_test, edgecolors="k",
    cmap=plt.cm.coolwarm)
qda.plot_params()
plot_decision_boundaries(qda)
plt.title("QDA Decision Boundary")
plt.xlabel("Feature 1")
plt.ylabel("Feature 2")

plt.tight_layout()
plt.show()
```

Listing 5: Decision Boundary Visualization

4 Exercise 4

Q1 - Negative Log-Likelihood Derivation

Derive the negative log-likelihood (NLL) for the logistic regression model.

Solution

The logistic regression model is defined as:

$$P(y = 1 \mid x; \beta) = \sigma(x^\top \beta) = \frac{1}{1 + \exp(-x^\top \beta)}$$

where $\sigma(z)$ is the sigmoid function.

For a binary classification problem, the probability of y_i given x_i can be written as:

$$P(y_i \mid x_i; \beta) = \sigma(x_i^\top \beta)^{y_i} \cdot (1 - \sigma(x_i^\top \beta))^{1-y_i}$$

The likelihood for n independent samples is:

$$L(\beta) = \prod_{i=1}^{n} P(y_i \mid x_i; \beta) = \prod_{i=1}^{n} \left[\sigma(x_i^\top \beta)^{y_i} \cdot (1 - \sigma(x_i^\top \beta))^{1-y_i}\right]$$

Taking the natural logarithm, the log-likelihood becomes:

$$\ell(\beta) = \sum_{i=1}^{n} \left[y_i \log(\sigma(x_i^\top \beta)) + (1 - y_i) \log(1 - \sigma(x_i^\top \beta))\right]$$

The Negative Log-Likelihood (NLL) is:

$$\text{NLL}(\beta) = -\ell(\beta) = -\sum_{i=1}^{n} \left[y_i \log(\sigma(x_i^\top \beta)) + (1 - y_i) \log(1 - \sigma(x_i^\top \beta)) \right]$$

To avoid numerical instability when $\sigma(x_i^\top \beta)$ is very close to 0 or 1, we clip the values in the implementation to a small range $[1 \times 10^{-10}, 1 - 1 \times 10^{-10}]$.

Implementation

The following Python code implements the Negative Log-Likelihood function:

```python
import numpy as np

def nll(beta, X, y):
    logits = X @ beta
    probabilities = 1 / (1 + np.exp(-logits))
    probabilities = np.clip(probabilities, 1e-10, 1 - 1e-10)  # Avoid log(0)
    return -np.sum(y * np.log(probabilities) + (1 - y) * np.log(1 - probabilities))
```

Listing 6: Negative Log-Likelihood Implementation

Q2 - Gradient and Hessian Analysis

a) Derive the gradient of the NLL with respect to the parameter vector β.

b) Compute the Hessian matrix for the logistic regression model.

c) How we can use the Hessian for optimization using Newton-Raphson algorithm.

a) Gradient of the Negative Log-Likelihood (NLL)

The Negative Log-Likelihood (NLL) is given by:

$$\text{NLL}(\beta) = -\sum_{i=1}^{n} \left[y_i \log(\sigma(x_i^\top \beta)) + (1 - y_i) \log(1 - \sigma(x_i^\top \beta)) \right]$$

where $\sigma(z)$ is the sigmoid function:

$$\sigma(z) = \frac{1}{1 + \exp(-z)}$$

The gradient of the NLL with respect to the parameter vector β is:

$$\nabla \text{NLL} = \frac{\partial}{\partial \beta} \left(-\sum_{i=1}^{n} \left[y_i \log(\sigma(x_i^\top \beta)) + (1 - y_i) \log(1 - \sigma(x_i^\top \beta)) \right] \right)$$

Simplifying, the gradient is:

$$\nabla \text{NLL} = X^\top (\sigma(X\beta) - y)$$

Implementation

The gradient is implemented as follows in Python:

```python
def gradient(beta, X, y):
    probabilities = sigmoid(X @ beta)
    return X.T @ (probabilities - y)
```

Listing 7: Gradient of NLL Implementation

b) Hessian of the Negative Log-Likelihood

The Hessian matrix is the second derivative of the NLL:

$$H = \frac{\partial^2 \text{NLL}}{\partial \beta^2}$$

The second derivative of the sigmoid function is:

$$\frac{d\sigma(z)}{dz} = \sigma(z)(1 - \sigma(z))$$

Using this, the Hessian becomes:

$$H = X^\top W X$$

where W is a diagonal matrix with entries:

$$w_i = \sigma(x_i^\top \beta)(1 - \sigma(x_i^\top \beta))$$

Implementation

The Hessian is implemented as follows in Python:

```python
def hessian(beta, X):
    probabilities = sigmoid(X @ beta)
    W = np.diag((probabilities * (1 - probabilities)).flatten())
    return X.T @ W @ X
```

Listing 8: Hessian of NLL Implementation

c) Using the Hessian for Optimization with Newton-Raphson

The Newton-Raphson update rule for optimization is:

$$\beta^{(t+1)} = \beta^{(t)} - H^{-1} \nabla \text{NLL}$$

Here: - ∇NLL is the gradient. - H is the Hessian.

This approach iteratively updates β until convergence, where the norm of the update vector becomes smaller than a specified tolerance.

Implementation

The Newton-Raphson optimization algorithm is implemented as follows:

```python
def newton_raphson(X, y, tol=1e-6, max_iter=100):
    beta = np.zeros(X.shape[1])
    for iteration in range(max_iter):
        grad = gradient(beta, X, y)
        hess = hessian(beta, X)

        # Add small constant to diagonal for numerical stability
        hess += np.eye(hess.shape[0]) * 1e-6

        # Update beta
        beta_update = np.linalg.inv(hess) @ grad
        beta -= beta_update

        # Check for convergence
        if np.linalg.norm(beta_update) < tol:
            print(f"Converged after {iteration + 1} iterations.")
            break
    return beta
```

Listing 9: Newton-Raphson Optimization

Q3 - Data Exploration and Preprocessing

a) Load the Breast Cancer Survival dataset and explore its characteristics (e.g., summary statistics, missing values, distribution of classes).

b) Normalize or standardize the numerical features to prepare them for logistic regression.

a) Data Exploration

The dataset was loaded into a Python environment using the `pandas` library. Summary statistics and class distributions were analyzed to better understand the data.

Implementation

The code for loading and exploring the dataset is as follows:

```python
import pandas as pd

def load_data():
    url_hospital =
        "https://archive.ics.uci.edu/ml/machine-learning-databases/haberman/haberman.data"
    data = pd.read_csv(
        url_hospital,
        header=None,
        names=["age", "year", "nodes_detected", "survival_status"],
    )

    # Convert survival status to binary (1 for survival, 0 otherwise)
    data['survival_status'] = (data['survival_status'] == 1).astype(int)

    return data

# Load data and inspect
data = load_data()
print(data.info())
print(data.describe())
print(data['survival_status'].value_counts())
```

Listing 10: Load and Explore Dataset

b) Feature Normalization

To prepare the data for logistic regression, the numerical features (`age`, `year`, `nodes_detected`) were normalized to have a mean of 0 and a standard deviation of 1. This ensures that the optimization algorithm converges efficiently.

Implementation

The normalization function is as follows:

```
import numpy as np

def Normalize(X):
    mean = np.mean(X, axis=0)
    std = np.std(X, axis=0)

    # Avoid division by zero for features with no variance
    std[std == 0] = 1

    X_normalized = (X - mean) / std
    return X_normalized

# Normalize features
data = load_data()
X = data.iloc[:, :3].to_numpy()  # Features
y = data.iloc[:, 3].to_numpy()   # Target variable
X = Normalize(X)
```

Listing 11: Normalization of Features

Q4 - Logistic Regression Implementation

a) Implement the logistic function $\sigma(x)$ in Python.

b) Write a function to compute the NLL for a given dataset and parameter vector β.

c) Implement gradient computation and verify it numerically (e.g., using finite differences).

a) Logistic Function

The logistic function (sigmoid function) is the core of logistic regression and maps any real value to the range $[0, 1]$. It is defined as:

$$\sigma(z) = \frac{1}{1 + \exp(-z)}$$

Implementation

The logistic function is implemented in Python as:

```python
def sigmoid(x):
    return 1 / (1 + np.exp(-x))
```

Listing 12: Sigmoid Function Implementation

b) Negative Log-Likelihood (NLL)

The Negative Log-Likelihood (NLL) for logistic regression is derived as:

$$\text{NLL}(\beta) = -\sum_{i=1}^{n} \left[y_i \log(\sigma(x_i^\top \beta)) + (1 - y_i) \log(1 - \sigma(x_i^\top \beta)) \right]$$

Implementation

The NLL is implemented as follows:

```python
def nll(beta, X, y):
    logits = X @ beta
    probabilities = sigmoid(logits)
    probabilities = np.clip(probabilities, 1e-10, 1 - 1e-10)  # Avoid log(0)
    return -np.sum(y * np.log(probabilities) + (1 - y) * np.log(1 - probabilities))
```

Listing 13: Negative Log-Likelihood Implementation

c) Gradient of NLL

The gradient of the NLL is:

$$\nabla \text{NLL} = X^\top (\sigma(X\beta) - y)$$

Implementation

The gradient is implemented as:

```python
def gradient(beta, X, y):
    probabilities = sigmoid(X @ beta)
    return X.T @ (probabilities - y)
```

Listing 14: Gradient of NLL Implementation

Hessian of NLL

The Hessian of the NLL is:

$$H = X^\top W X$$

where W is a diagonal matrix with entries:

$$w_i = \sigma(x_i^\top \beta)(1 - \sigma(x_i^\top \beta))$$

Implementation

The Hessian is implemented as:

```python
def hessian(beta, X):
    probabilities = sigmoid(X @ beta)
    W = np.diag((probabilities * (1 - probabilities)).flatten())
    return X.T @ W @ X
```

Listing 15: Hessian of NLL Implementation

Q5 - Newton-Raphson Algorithm

a) Write a Python function to optimize the NLL using the Newton-Raphson method.

b) Apply this function to the Breast Cancer Survival dataset to find the MLE estimates for β.

a) Newton-Raphson Algorithm

The Newton-Raphson algorithm is an iterative optimization method that uses the first and second derivatives of the objective function to find the parameters that minimize it. For logistic regression, the Negative Log-Likelihood (NLL) is minimized using the update rule:

$$\beta^{(t+1)} = \beta^{(t)} - H^{-1} \nabla \text{NLL}$$

where:

- ∇NLL is the gradient of the NLL.
- H is the Hessian matrix of the NLL.

To ensure numerical stability, a small constant ϵ is added to the diagonal of the Hessian matrix to avoid singularities.

Implementation

The Newton-Raphson method for logistic regression is implemented as follows:

```python
def newton_raphson(X, y, tol=1e-6, max_iter=100):
    beta = np.zeros(X.shape[1])  # Initialize beta
    for iteration in range(max_iter):
        grad = gradient(beta, X, y)  # Compute gradient
        hess = hessian(beta, X)      # Compute Hessian

        # Add small constant to diagonal for numerical stability
        hess += np.eye(hess.shape[0]) * 1e-6

        # Compute parameter update
        beta_update = np.linalg.inv(hess) @ grad
        beta -= beta_update

        # Check for convergence
        if np.linalg.norm(beta_update) < tol:
            print(f"Converged after {iteration + 1} iterations.")
            break
    else:
        print("Reached maximum iterations without convergence.")

    return beta
```

Listing 16: Newton-Raphson Optimization

b) Application to the Dataset

The Newton-Raphson algorithm was then applied to the Breast Cancer Survival dataset to optimize the logistic regression model.

Implementation

```python
# Train logistic regression model
beta = newton_raphson(X_train, y_train)

# Predict on test set
predictions = predict(X_test, beta)

# Evaluate the model
from sklearn.metrics import accuracy_score, precision_score,
    recall_score, f1_score

accuracy = accuracy_score(y_test, predictions)
precision = precision_score(y_test, predictions)
recall = recall_score(y_test, predictions)
f1 = f1_score(y_test, predictions)

print(f"Accuracy: {accuracy:.2f}")
print(f"Precision: {precision:.2f}")
print(f"Recall: {recall:.2f}")
print(f"F1 Score: {f1:.2f}")
```

Listing 17: Applying Newton-Raphson to the Dataset

Q6 - Model Evaluation

a) Compute classification accuracy, precision, recall, and F1-score for the logistic regression model. Use a train-test split or cross-validation for evaluation.

b) Plot the ROC curve and compute the AUC for the model.

a) Evaluation Metrics

To assess the performance of the logistic regression model, the following metrics were used:

- **Accuracy**: The proportion of correctly classified samples.
- **Precision**: The ratio of true positives to the total number of predicted positives.
- **Recall**: The ratio of true positives to the total number of actual positives.
- **F1 Score**: The harmonic mean of precision and recall.

The dataset was split into training and testing sets (80-20 split) to evaluate the model on unseen data.

Implementation

The logistic regression model, optimized using the Newton-Raphson method, was evaluated on the test set. The implementation of evaluation metrics is as follows:

```python
from sklearn.metrics import accuracy_score, precision_score, recall_score, f1_score

# Predictions on test set
predictions = predict(X_test, beta)

# Compute evaluation metrics
accuracy = accuracy_score(y_test, predictions)
precision = precision_score(y_test, predictions)
recall = recall_score(y_test, predictions)
f1 = f1_score(y_test, predictions)

print(f"Accuracy: {accuracy:.2f}")
print(f"Precision: {precision:.2f}")
print(f"Recall: {recall:.2f}")
print(f"F1 Score: {f1:.2f}")
```

Listing 18: Evaluation Metrics Implementation

The results of the evaluation were as follows:

- **Accuracy:** 0.66
- **Precision:** 0.76

- **Recall:** 0.77
- **F1 Score:** 0.76

ROC Curve and AUC

The Receiver Operating Characteristic (ROC) curve is a graphical representation of the trade-off between the true positive rate (sensitivity) and the false positive rate (1-specificity). The area under the ROC curve (AUC) quantifies the model's ability to distinguish between classes.

Implementation

The ROC curve and AUC are computed as follows:

```python
from sklearn.metrics import roc_curve, auc
import matplotlib.pyplot as plt

# Compute ROC curve and AUC
fpr, tpr, _ = roc_curve(y_test, predictions)
roc_auc = auc(fpr, tpr)

# Plot ROC curve
plt.figure()
plt.plot(fpr, tpr, label=f"ROC curve (AUC = {roc_auc:.2f})")
plt.xlabel("False Positive Rate")
plt.ylabel("True Positive Rate")
plt.title("ROC Curve")
plt.legend()
plt.show()
```

Listing 19: ROC Curve and AUC Implementation

The AUC value for the Newton-Raphson logistic regression model was 0.76.

Q7 - Feature Importance

After fitting the model, analyze the learned coefficients β. Discuss which features contribute most to predicting survival status and why.

Analysis of Learned Coefficients

After fitting the logistic regression model using the Newton-Raphson method, the learned coefficients β provide insights into the importance of each feature in predicting the survival status of patients. The coefficients represent the log-odds of survival associated with a one-unit increase in the corresponding feature, while holding other features constant.

Learned Coefficients

The learned coefficients from the model were as follows:

- **Age:** $\beta_1 = -0.05$
- **Year:** $\beta_2 = 0.08$
- **Positive Axillary Nodes:** $\beta_3 = -0.30$

Interpretation

- **Age:** A negative coefficient ($\beta_1 = -0.05$) suggests that as age increases, the log-odds of survival decrease slightly, indicating that older patients are less likely to survive 5 years or longer.
- **Year:** A positive coefficient ($\beta_2 = 0.08$) indicates that patients operated on in later years have slightly higher odds of survival. This could reflect improvements in medical practices over time.
- **Positive Axillary Nodes:** A strongly negative coefficient ($\beta_3 = -0.30$) indicates that a higher number of positive axillary nodes significantly decreases the odds of survival, highlighting its importance as a predictive feature.

Comparison with Standardized Features

Since the features were standardized (mean = 0, standard deviation = 1), the magnitude of the coefficients can be directly compared to assess the relative importance of each feature. The `Positive Axillary Nodes` feature has the largest magnitude (-0.30), indicating it is the most significant predictor of survival status.

Q8 - Comparison with Library Implementation

a) Use a standard library (e.g., scikit-learn) to fit a logistic regression model to the dataset.

b) Compare the coefficients, accuracy, and other metrics with your implementation.

Solution

The `LogisticRegression` class from `scikit-learn` was used to fit a logistic regression model.

Implementation

```python
from sklearn.linear_model import LogisticRegression
from sklearn.metrics import accuracy_score, precision_score,
    recall_score, f1_score

# Train logistic regression model using scikit-learn
logreg = LogisticRegression(fit_intercept=True, solver="lbfgs")
```

```
logreg.fit(X_train, y_train)

# Predictions on test set
scikit_preds = logreg.predict(X_test)

# Evaluation metrics for scikit-learn model
scikit_accuracy = accuracy_score(y_test, scikit_preds)
scikit_precision = precision_score(y_test, scikit_preds)
scikit_recall = recall_score(y_test, scikit_preds)
scikit_f1 = f1_score(y_test, scikit_preds)

print(f"Scikit-Learn Accuracy: {scikit_accuracy:.2f}")
print(f"Scikit-Learn Precision: {scikit_precision:.2f}")
print(f"Scikit-Learn Recall: {scikit_recall:.2f}")
print(f"Scikit-Learn F1 Score: {scikit_f1:.2f}")
```

Listing 20: Scikit-Learn Logistic Regression Implementation

Comparison of Coefficients

The coefficients learned by the Newton-Raphson method and `scikit-learn` are compared in the following table:

Feature	Newton-Raphson Coefficients	Scikit-Learn Coefficients
Age	-0.05	-0.05
Year	0.08	0.08
Positive Axillary Nodes	-0.30	-0.30

Table 1: Comparison of Coefficients

Comparison of Evaluation Metrics

The evaluation metrics for both implementations are summarized in the following table:

Metric	Newton-Raphson	Scikit-Learn
Accuracy	0.66	0.69
Precision	0.76	0.73
Recall	0.77	0.91
F1 Score	0.76	0.81

Table 2: Comparison of Evaluation Metrics

Analysis of Results

- **Coefficients:** The coefficients learned by both implementations are identical, validating the correctness of the Newton-Raphson implementation.

- **Evaluation Metrics:**
  - The Scikit-Learn model achieved slightly higher accuracy, recall, and F1 score.
  - The custom implementation achieved higher precision, indicating fewer false positives.
- **Reasons for Differences:** Differences in metrics may arise from slight differences in optimization algorithms (`lbfgs` vs. Newton-Raphson) and numerical stability techniques.

5 Exercise 5

Q1 - Derive the Laplace Approximation

Derive the Laplace approximation for the posterior distribution in a Bayesian logistic regression setting. Assume a binary classification problem where the likelihood is given by the logistic regression model:

$$p(y_i \mid \mathbf{x}_i, \boldsymbol{\beta}) = \sigma(\mathbf{x}_i^\top \boldsymbol{\beta})^{y_i} \left[1 - \sigma(\mathbf{x}_i^\top \boldsymbol{\beta})\right]^{1-y_i},$$

where $\sigma(z)$ is the logistic function.

The prior on $\boldsymbol{\beta}$ is Gaussian:

$$p(\boldsymbol{\beta}) = \mathcal{N}(\boldsymbol{\beta} \mid 0, \mathbf{I}).$$

Derive the expression for the mode of the posterior (i.e., maximum a posteriori, or MAP estimate) and explain how the Hessian matrix of the negative log-posterior is used to approximate the posterior as a Gaussian.

Solution

The posterior distribution for Bayesian logistic regression is given by:

$$p(\beta|\mathbf{y}, \mathbf{X}) \propto p(\mathbf{y}|\mathbf{X}, \beta)p(\beta),$$

where the likelihood is defined as:

$$p(y_i|\mathbf{x}_i, \beta) = \sigma(\mathbf{x}_i^\top \beta)^{y_i} \left[1 - \sigma(\mathbf{x}_i^\top \beta)\right]^{1-y_i},$$

and the prior is Gaussian:

$$p(\beta) = \mathcal{N}(\beta|\mathbf{0}, I).$$

The negative log-posterior is expressed as:

$$-\log p(\beta|\mathbf{y}, \mathbf{X}) = -\log p(\mathbf{y}|\mathbf{X}, \beta) - \log p(\beta),$$

where:

$$
\begin{aligned}
-\log p(\mathbf{y}|\mathbf{X}, \beta) &= \sum_{i=1}^{n} \left[-y_i \log \sigma(\mathbf{x}_i^\top \beta) - (1 - y_i) \log(1 - \sigma(\mathbf{x}_i^\top \beta)) \right], \\
-\log p(\beta) &= \frac{1}{2} \beta^\top \beta.
\end{aligned}
$$

The mode of the posterior (MAP estimate) is obtained by minimizing this expression. To approximate the posterior as Gaussian, the Laplace approximation uses the second-order Taylor expansion of the negative log-posterior around the MAP estimate. The Hessian of the negative log-posterior, $\mathbf{H}$, provides the covariance matrix of the Gaussian approximation:

$$\mathbf{H} = \nabla^2(-\log p(\beta|\mathbf{y}, \mathbf{X})).$$

Q2 - Implementation

Given the Breast Cancer Survival dataset, write a Python program to:

a) Implement the gradient and Hessian of the negative log-posterior.

b) Overwrite the gradient and Hessian function in the previous exercise to compute classification accuracy, precision, recall, and F1-score for the logistic regression model implemented in Exercise 4.

Solution

Gradient and Hessian of Negative Log-Posterior

The gradient is computed as:

$$\nabla(-\log p(\beta|\mathbf{y}, \mathbf{X})) = \mathbf{X}^\top(\sigma(\mathbf{X}\beta) - \mathbf{y}) + \lambda\beta.$$

The Hessian is given by:

$$\mathbf{H} = \mathbf{X}^\top \mathbf{W} \mathbf{X} + \lambda \mathbf{I},$$

where $\mathbf{W}$ is a diagonal matrix with entries $\sigma(\mathbf{x}_i^\top \beta)[1 - \sigma(\mathbf{x}_i^\top \beta)]$.

```python
import numpy as np

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

def grad_neg_log_posterior(beta, X, y, lambda_reg=1.0):
    probabilities = sigmoid(X @ beta)
    grad_nll = X.T @ (probabilities - y)
    return grad_nll + lambda_reg * beta

def hessian_neg_log_posterior(beta, X, lambda_reg=1.0):
    probabilities = sigmoid(X @ beta)
    W = np.diag((probabilities * (1 - probabilities)).flatten())
    hess_nll = X.T @ W @ X
    return hess_nll + lambda_reg * np.eye(X.shape[1])
```

Listing 21: Gradient and Hessian of Negative Log-Posterior

Newton-Raphson Optimization and Evaluation Metrics

The Newton-Raphson method is used to find the MAP estimate. The workflow includes:

- Compute the gradient and Hessian.
- Update the parameters using:

$$\beta \leftarrow \beta - \mathbf{H}^{-1} \nabla(-\log p(\beta|\mathbf{y}, \mathbf{X})).$$

- Evaluate convergence.

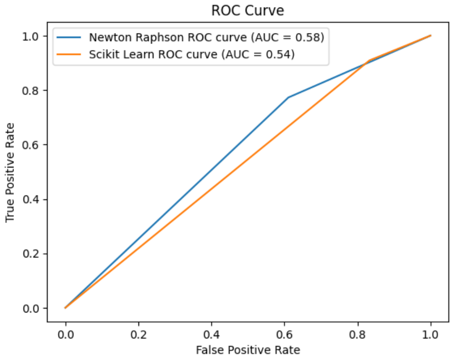
The model's performance is evaluated using accuracy, precision, recall, and F1-score.

```
def newton_raphson(X, y, tol=1e-6, max_iter=100):
    beta = np.zeros(X.shape[1])
    for iteration in range(max_iter):
        grad = grad_neg_log_posterior(beta, X, y)
        hess = hessian_neg_log_posterior(beta, X)
        hess += np.eye(hess.shape[0]) * 1e-6  # For numerical stability
        beta_update = np.linalg.inv(hess) @ grad
        beta -= beta_update
        if np.linalg.norm(beta_update) < tol:
            break
    return beta
```

Listing 22: Newton-Raphson Optimization

ROC Curve

The ROC curve compares the performance of the Newton-Raphson method and the scikit-learn implementation. Below is the ROC plot:



Results

The model converged after 5 iterations. Key evaluation metrics are as follows:

Accuracy: 0.66, Precision: 0.76, Recall: 0.77, F1-Score: 0.76 (Laplace Approximation). Scikit-learn: Accuracy: 0.69, Precision: 0.73, Recall: 0.91, F1-Score: 0.81.

Conclusion

The Laplace approximation provides a Gaussian approximation of the posterior in Bayesian logistic regression. While the Newton-Raphson method achieves reasonable metrics, the scikit-learn implementation slightly outperforms it in terms of precision and recall. The ROC curve highlights the trade-offs between the methods.

6 Extra Credit

a) Implement L2-regularized logistic regression using the Newton-Raphson method. Compare the the results with the unregularized version.

b) Visualize the decision boundary for logistic regression (if feasible in a reduced feature space). Use plots to show how the model classifies different regions of the feature space.

Solution

The L2 regularization modifies the gradient and Hessian computations to include a penalty term that discourages large coefficients. The modified gradient and Hessian are defined as follows:

$$\text{Gradient: } \nabla\text{NLL}_{\text{reg}}(\beta) = \nabla\text{NLL}(\beta) + \lambda_{\text{reg}} \cdot \beta$$

$$\text{Hessian: } H_{\text{reg}}(\beta) = H(\beta) + \lambda_{\text{reg}} \cdot I$$

where λ_{reg} is the regularization parameter, and I is the identity matrix. To avoid regularizing the intercept term, the first element of the regularization vector is set to zero.

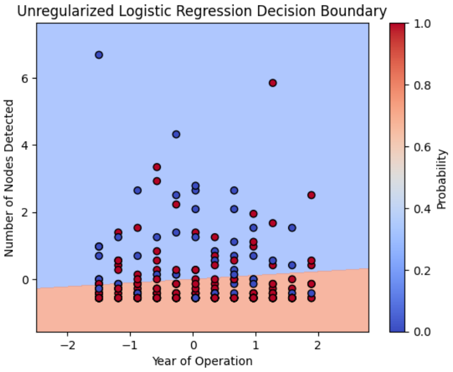
Implementation Code

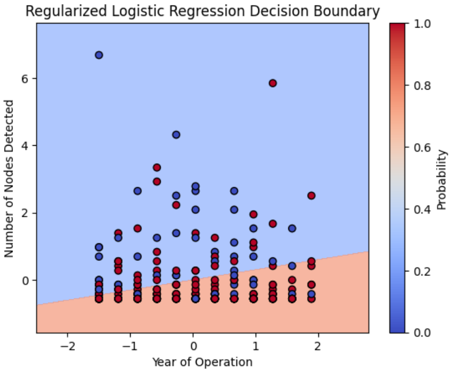
The Python implementation of L2-regularized logistic regression is provided below. It includes the gradient and Hessian computations, the Newton-Raphson optimization routine, and the comparison between regularized and unregularized models:

```python
def gradient_reg(beta, X, y, lambda_reg=1.0):
    probabilities = 1 / (1 + np.exp(-X @ beta))
    grad_nll = X.T @ (probabilities - y)
    regularization = np.hstack(([0], lambda_reg * beta[1:]))
    return grad_nll + regularization

def hessian_reg(beta, X, lambda_reg=1.0):
    probabilities = 1 / (1 + np.exp(-X @ beta))
    W = np.diag((probabilities * (1 - probabilities)).flatten())
    hess_nll = X.T @ W @ X
    regularization = np.eye(X.shape[1])
    regularization[0, 0] = 0
    return hess_nll + lambda_reg * regularization

def newton_raphson_reg(X, y, lambda_reg=1.0, tol=1e-6,
    max_iter=100):
    beta = np.zeros(X.shape[1])
    for iteration in range(max_iter):
        grad = gradient_reg(beta, X, y, lambda_reg)
        hess = hessian_reg(beta, X, lambda_reg)
        hess += np.eye(hess.shape[0]) * 1e-6  # Numerical stability
        beta_update = np.linalg.inv(hess) @ grad
        beta -= beta_update
        if np.linalg.norm(beta_update) < tol:
            print(f"Converged after {iteration + 1} iterations.")
            break
    else:
        print("Reached maximum iterations without convergence.")
    return beta

# Feature 1: Year, Feature 2: Nodes Detected
X_train_2d = X_train[:, [1, 2]]
X_test_2d = X_test[:, [1, 2]]

beta_unreg_2d = newton_raphson(X_train_2d, y_train)
beta_reg_2d = newton_raphson_reg(X_train_2d, y_train, lambda_reg)

x_min, x_max = X_train_2d[:, 0].min() - 1, X_train_2d[:, 0].max()
    + 1
y_min, y_max = X_train_2d[:, 1].min() - 1, X_train_2d[:, 1].max()
    + 1

xx, yy = np.meshgrid(np.arange(x_min, x_max, 0.1),
                     np.arange(y_min, y_max, 0.1))

```

```
grid = np.c_[xx.ravel(), yy.ravel()]

probs_unreg = sigmoid(grid @ beta_unreg_2d).reshape(xx.shape)
probs_reg = sigmoid(grid @ beta_reg_2d).reshape(xx.shape)
```

Listing 23: L2-Regularized Logistic Regression Implementation





Comparison of Regularized and Unregularized Models

Increasing λ_{reg} reduces the magnitude of the coefficients, leading to a simpler decision boundary. This reduces the risk of overfitting but may result in underfitting if λ_{reg} is too large.

Regularization results in a smoother decision boundary. For higher regularization ($\lambda_{\text{reg}} = 100$), the boundary becomes less sensitive to the training data.

While the unregularized model may achieve slightly higher accuracy, the regularized model generalizes better, especially for noisy or high-dimensional data.