AI and Machine Learning Homework2 Report 纪浩正 12311405

Student: 纪浩正, jihz2023@mail.sustech.edu.cn

1 Logistic Regression Model

1.1 Mathematical Foundation

Logistic Regression is a binary classification algorithm that models the probability of a sample belonging to a particular class. The core of the model is the sigmoid function:

$$\sigma(z) = \frac{1}{1 + e^{-z}} \tag{1}$$

where $z = \mathbf{w}^T\mathbf{x} + b$ is the linear combination of input features. The model predicts the probability:

$$P(y = 1|\mathbf{x}) = \sigma(-\mathbf{w}^T\mathbf{x} - b) \tag{2}$$

The loss function used is the binary cross-entropy:

$$L(\mathbf{w}) = -\sum_{i=1}^{n} t_i \log\Big(y^{(x_i)}, \omega\Big) - \sum_{i=1}^{n}(1 - t_i) \log\Big(1 - y^{(x_i)}, \omega\Big) \tag{3}$$

1.2 Implementation

The implementation includes data normalization, gradient descent optimization, and prediction methods. Two normalization techniques are supported:

Min-Max Normalization:

$$X_{norm} = \frac{X - X_{min}}{X_{max} - X_{min}} \tag{4}$$

Standardization (Z-score):

$$X_{norm} = \frac{X - \mu}{\sigma} \tag{5}$$

Listing 1: Logistic Regression Class Implementation

```
class LogisticRegression:
```

```
def __init__(self):
    self.w = None
    self.nsample, self.ndim = None, None
    self.X, self.Y = None, None
    self.norm_mode = None
    self.x_min, self.x_max = None, None
    self.x_mean, self.x_std = None, None

def _f(self, x):
    return 1 / (1 + np.exp(-x))

def normalize(self, X):
    if self.norm_mode == "min-max":
        self.x_min = X.min(axis=0)
        self.x_max = X.max(axis=0)
        X_norm = (X - self.x_min) / (self.x_max - self.x_min)
    elif self.norm_mode == "mean":
        self.x_mean = X.mean(axis=0)
        self.x_std = X.std(axis=0)
        X_norm = (X - self.x_mean) / self.x_std
    elif self.norm_mode == None:
        pass
    else:
        raise ValueError("norm_mode must be 'mean', 'min-max' or None")
    return X_norm

def _normalize(self, X_pred):
    if self.norm_mode == "min-max":
        return (X_pred - self.x_min) / (self.x_max - self.x_min)
    elif self.norm_mode == "mean":
        return (X_pred - self.x_mean) / self.x_std
    elif self.norm_mode == None:
        return X_pred

def _pred(self, x):
    return self._f(self.w @ x)

def fit(
    self, X, Y, method=None, norm_mode=None, epochs=100, lr=0.01, batch_size=32
):
    self.X = X
    self.norm_mode = norm_mode
    self.Y = Y
    self.nsample, self.ndim = X.shape
    self.w = np.ones((self.ndim + 1))
    X_norm = self.normalize(X)
    b = np.ones((self.nsample, 1))
```

```
        X_norm = np.hstack((b, X_norm))
        losses = []

        for epoch in range(epochs):
            if method == "sgd":
                idx = np.arange(self.nsample)
                np.random.shuffle(idx)
                for ii in idx:
                    xi = X_norm[ii, :]
                    yi = Y[ii]
                    y_pred = self._pred(xi)
                    dw = -xi * (yi - y_pred)
                    self.w = self.w - lr * dw
                loss = -np.dot(self.Y, np.log(self._f(X_norm @ self.w))) - np.dot(
                    (1 - self.Y), np.log(1 - self._f(X_norm @ self.w))
                )
                losses.append(loss)
            if method == "mbgd":
                idx = np.arange(self.nsample)
                np.random.shuffle(idx)
                for s_idx in range(0, self.nsample, batch_size):
                    e_idx = s_idx + batch_size
                    batch_idx = idx[s_idx:e_idx]
                    X_batch = X_norm[batch_idx]
                    y_batch = Y[batch_idx]
                    y_pred = self._f(X_batch @ self.w)
                    dw = -X_batch.T @ (y_batch - y_pred)
                    self.w = self.w - lr * dw
                epsilon = 1e-8
                loss = -np.dot(
                    self.Y, np.log(self._f(X_norm @ self.w) + epsilon)
                ) - np.dot((1 - self.Y), np.log(1 - self._f(X_norm @ self.w) + epsilon))
                losses.append(loss)
        return losses, self.w

    def predict(self, X_pred):
        X_norm = self._normalize(X_pred)
        b = np.ones((X_norm.shape[0], 1))
        X_norm = np.hstack((b, X_norm))
        y_pred = self._f(X_norm @ self.w)
        return y_pred

    def prob2class(self, y, threshold):
        return (y > threshold).astype(int)

    def evaluate(self, ypred, ytest, mode="accuracy"):
        tp, fp, tn, fn = 0, 0, 0, 0
```

```
    for ii in range(ytest.shape[0]):
        if ypred[ii] == 1:
            if ytest[ii] == 1:
                tp += 1
            if ytest[ii] == 0:
                fp += 1
        elif ypred[ii] == 0:
            if ytest[ii] == 1:
                fn += 1
            if ytest[ii] == 0:
                tn += 1
    if mode == "accuracy":
        return (tp + tn) / (len(ypred))
    if mode == "recall":
        return tp / (tp + fn)
    if mode == "precision":
        return tp / (tp + fp)
    if mode == "F1 score":
        R = tp / (tp + fn)
        P = tp / (tp + fp)
        return 2 * P * R / (P + R)
```

1.3 Dataset Preparation and Splitting

The Wine dataset is preprocessed to create a binary classification problem. We specifically **exclude class 3** from the original three-class Wine dataset, focusing only on distinguishing between the first two wine cultivars (classes 1 and 2). This reduction to a binary classification task allows us to apply logistic regression, which is inherently designed for binary outcomes. The remaining classes are converted into binary labels: class 1 becomes label 0, and class 2 becomes label 1.

The dataset is then split into training (70%) and test (30%) sets using random sampling with a fixed seed to ensure reproducibility. After preprocessing, the complete dataset contains **130 samples** with **13 features** each, which is then divided into **91 training samples** and **39 test samples**.

Listing 2: Data Loading and Splitting

```
# Problem1 read the table, devide the data set into training set and test set, gen_new 01label
    for different wines
np.random.seed(42)
df = pd.read_csv("wine.data", header=None)
df_new = df[df[0] != 3].reset_index(drop=True)
X = df_new.iloc[:, 1:].values
```

```
Y = df_new.iloc[:, 0].values

y_mean = Y.mean(axis=0)
Y = Y - y_mean
for ii in range(X.shape[0]):
    if Y[ii] < 0:
        Y[ii] = 0
    else:
        Y[ii] = 1

(nsample, ndim) = X.shape
idx = np.arange(nsample)
nper70 = round(nsample * 0.7)
np.random.shuffle(idx)
Xtraining = X[idx[:nper70]]
Ytraining = Y[idx[:nper70]]
Xtest = X[idx[nper70:]]
Ytest = Y[idx[nper70:]]
```

2 Model Training

2.1 Optimization Methods

Two gradient descent variants are implemented and compared:

Stochastic Gradient Descent (SGD): Updates weights after each individual sample:

$$\mathbf{w}_{t+1} = \mathbf{w}_t - \eta \nabla L(\mathbf{w}_t; x_i, y_i) \tag{6}$$

Mini-Batch Gradient Descent (MBGD): Updates weights using a small batch of samples:

$$\mathbf{w}_{t+1} = \mathbf{w}_t - \eta \sum_{i \in B} \nabla L(\mathbf{w}_t; x_i, y_i) \tag{7}$$

where η is the learning rate and B is the mini-batch.

2.2 Training Configuration

The model is trained with the following hyperparameters:

- Number of epochs: 100
- Learning rate: 0.01

- Batch size: 64 (for MBGD)
- Normalization: Min-Max scaling

Listing 3: Model Training with Both Methods

```
# Problem2 use the minibatch and stochastic to train the logistic regression mode
method1 = "mbgd"
method2 = "sgd"
norm_mode = "min-max"
N_epoch = 100
Lr = 0.01
batch_size = 16

Model = LogisticRegression()
losses1, weight1 = Model.fit(
    Xtraining,
    Ytraining,
    method=method1,
    norm_mode=norm_mode,
    epochs=N_epoch,
    lr=Lr,
    batch_size=batch_size,
)
# problem3 evaluate the model
y_pred = Model.predict(Xtest)
y_class = Model.prob2class(y_pred, 0.5)

accuracy = Model.evaluate(y_class, Ytest, "accuracy")
recall = Model.evaluate(y_class, Ytest, "recall")
precision = Model.evaluate(y_class, Ytest, "precision")
F1score = Model.evaluate(y_class, Ytest, "F1 score")
print("mini-batch")
print(f"Accuracy: {accuracy:.4f} ({accuracy*100:.2f}%)")
print(f"Recall: {recall:.4f} ({recall*100:.2f}%)")
print(f"Precision: {precision:.4f} ({precision*100:.2f}%)")
print(f"F1 Score: {F1score:.4f} ({F1score*100:.2f}%)")

losses2, weight2 = Model.fit(
    Xtraining,
    Ytraining,
    method=method2,
    norm_mode=norm_mode,
    epochs=N_epoch,
    lr=Lr,
    batch_size=batch_size,
)
```

```
y_pred = Model.predict(Xtest)
y_class = Model.prob2class(y_pred, 0.5)

# problem3 evaluate the model
accuracy = Model.evaluate(y_class, Ytest, "accuracy")
recall = Model.evaluate(y_class, Ytest, "recall")
precision = Model.evaluate(y_class, Ytest, "precision")
F1score = Model.evaluate(y_class, Ytest, "F1 score")
print("stochastic ")
print(f"Accuracy: {accuracy:.4f} ({accuracy*100:.2f}%)")
print(f"Recall: {recall:.4f} ({recall*100:.2f}%)")
print(f"Precision: {precision:.4f} ({precision*100:.2f}%)")
print(f"F1 Score: {F1score:.4f} ({F1score*100:.2f}%)")

plt.plot(np.arange(len(losses1)), losses1, "r", label="minibatch")
plt.plot(np.arange(len(losses2)), losses2, "b--", label="stochastic")
plt.xlabel("Epochs")
plt.ylabel("Loss")
plt.title(f"Training Loss Over Time")
plt.grid(True, alpha=0.3)
plt.legend()
plt.savefig(f"images/Training_Loss_over_time.png", dpi=300, bbox_inches="tight")
plt.show()
```

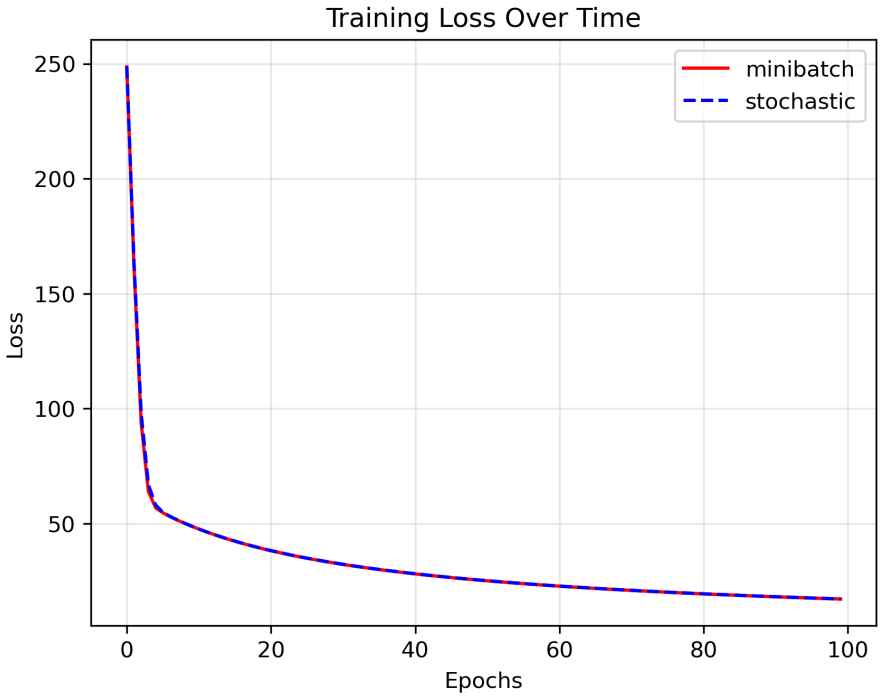

Figure 1 Comparison of training loss convergence between Mini-Batch Gradient Descent and Stochastic Gradient Descent. Both methods show consistent convergence patterns with comparable final loss values.

3 Model Evaluation

3.1 Evaluation Metrics

The model performance is assessed using four standard classification metrics:

Accuracy: Overall correctness of predictions

$$\text{Accuracy} = \frac{TP + TN}{TP + TN + FP + FN} \tag{8}$$

Precision: Proportion of positive predictions that are correct

$$\text{Precision} = \frac{TP}{TP + FP} \tag{9}$$

Recall (Sensitivity): Proportion of actual positives correctly identified

$$\text{Recall} = \frac{TP}{TP + FN} \tag{10}$$

F1 Score: Harmonic mean of precision and recall

$$\text{F1} = 2 \times \frac{\text{Precision} \times \text{Recall}}{\text{Precision} + \text{Recall}} \tag{11}$$

where TP = True Positives, TN = True Negatives, FP = False Positives, FN = False Negatives.

3.2 Results

The logistic regression model demonstrates exceptional performance on the test set, achieving perfect classification accuracy across all evaluation metrics.

As shown in Figure 2, both optimization methods achieve identical and flawless performance across all metrics:

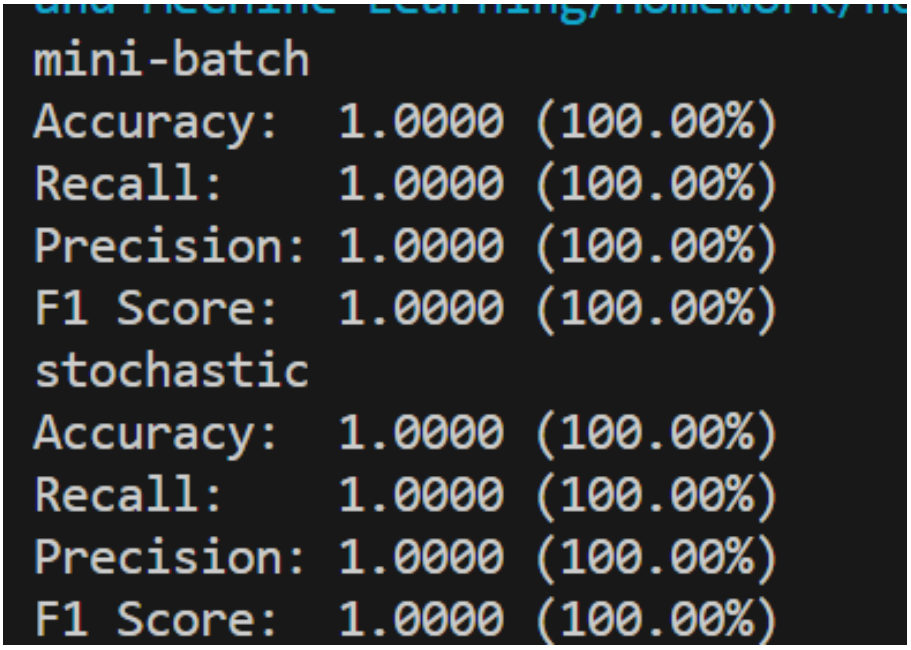

Figure 2 **Comparative evaluation results for both optimization methods (Mini-Batch GD and Stochastic GD), demonstrating identical perfect classification performance with 100% accuracy, recall, precision, and F1 score on the test set.**

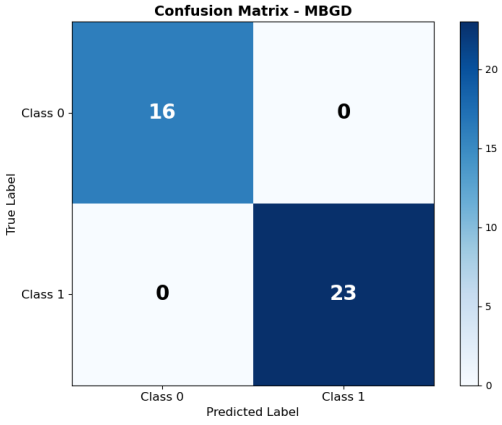

Figure 3 **Confusion matrix for Mini-Batch Gradient Descent showing the distribution of true positives, true negatives, false positives, and false negatives.**

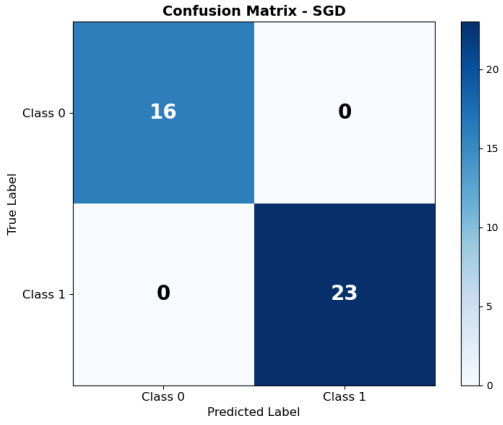

Figure 4 **Confusion matrix for Stochastic Gradient Descent demonstrating similar classification patterns with slight variations in error distribution.**

Listing 4: Model Evaluation on Test Set

```
# problem3 evaluate the model
y_pred = Model.predict(Xtest)
y_class = Model.prob2class(y_pred, 0.5)
```

```
accuracy = Model.evaluate(y_class, Ytest, "accuracy")
recall = Model.evaluate(y_class, Ytest, "recall")
precision = Model.evaluate(y_class, Ytest, "precision")
F1score = Model.evaluate(y_class, Ytest, "F1 score")
print("mini-batch")
print(f"Accuracy: {accuracy:.4f} ({accuracy*100:.2f}%)")
print(f"Recall: {recall:.4f} ({recall*100:.2f}%)")
print(f"Precision: {precision:.4f} ({precision*100:.2f}%)")
print(f"F1 Score: {F1score:.4f} ({F1score*100:.2f}%)")

# problem3 evaluate the model
accuracy = Model.evaluate(y_class, Ytest, "accuracy")
recall = Model.evaluate(y_class, Ytest, "recall")
precision = Model.evaluate(y_class, Ytest, "precision")
F1score = Model.evaluate(y_class, Ytest, "F1 score")
print("stochastic ")
print(f"Accuracy: {accuracy:.4f} ({accuracy*100:.2f}%)")
print(f"Recall: {recall:.4f} ({recall*100:.2f}%)")
print(f"Precision: {precision:.4f} ({precision*100:.2f}%)")
print(f"F1 Score: {F1score:.4f} ({F1score*100:.2f}%)")
```

3.3 Decision Boundary Visualization

To validate the separability of the two wine classes, we visualize the learned decision boundaries by projecting the 13-dimensional data onto various 2D feature spaces. These visualizations demonstrate that **clear differences exist between the two wine classes across multiple chemical feature dimensions**, which validates the feasibility and success of binary classification.

The projections across different feature pairs reveal that the two wine cultivars (classes 1 and 2, excluding class 3) exhibit genuine chemical differences. Among the visualized feature combinations, **features 0 vs 1 (Alcohol vs Malic Acid)** and **features 2 vs 12 (Ash vs Proline)** show particularly clear separation between the two classes, demonstrating strong discriminative power in distinguishing the wine varieties.

Important Note: It should be emphasized that the decision boundaries shown in these 2D projections are **visual boundaries** rather than the true decision boundary in the full 13-dimensional feature space. Since the logistic regression model learns weights for all 13 features simultaneously, including the bias term w_0, the actual decision boundary is:

$$w_0 + w_1 x_1 + w_2 x_2 + \cdots + w_{13} x_{13} = 0 \tag{12}$$

When projecting onto a 2D subspace (e.g., features i and j), the displayed boundary is influenced by:

- The bias term w_0, which is affected by all 13 features during training
- The contributions from the other 11 features not shown in the projection

Therefore, these 2D visualizations serve as **approximate representations** to illustrate class separability in selected feature subspaces, rather than exact depictions of the hyperplane learned in the complete 13-dimensional space. Nevertheless, the clear visual separation observed in multiple projections confirms that the classes are indeed well-separated in the full feature space.

These observable separations in multiple feature dimensions explain why logistic regression achieves perfect classification—the classes are naturally distinct and linearly separable in the feature space, making the binary classification task well-defined and physically meaningful.

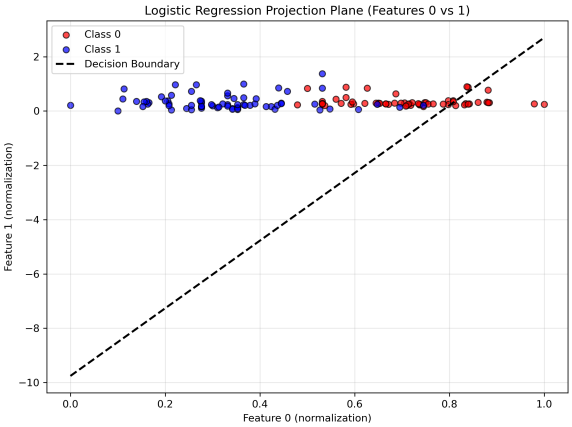

Figure 5 Decision boundary projection on features 0 vs 1

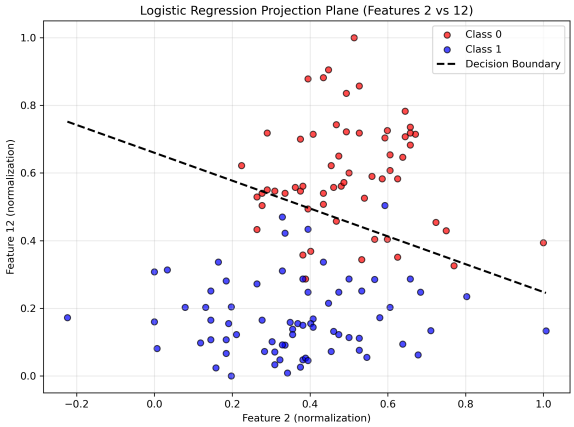

Figure 6 Decision boundary projection on features 2 vs 12

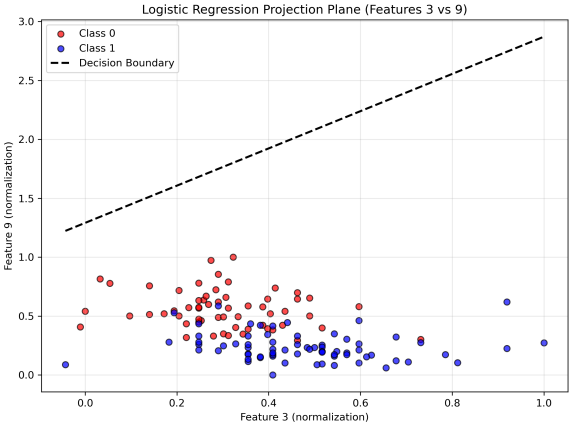

Figure 7 Decision boundary projection on features 3 vs 9

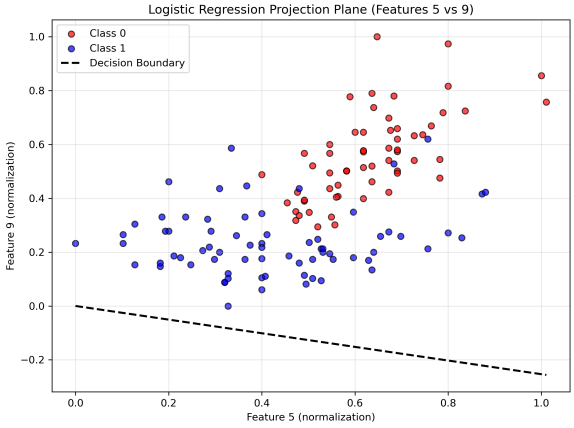

Figure 8 Decision boundary projection on features 5 vs 9

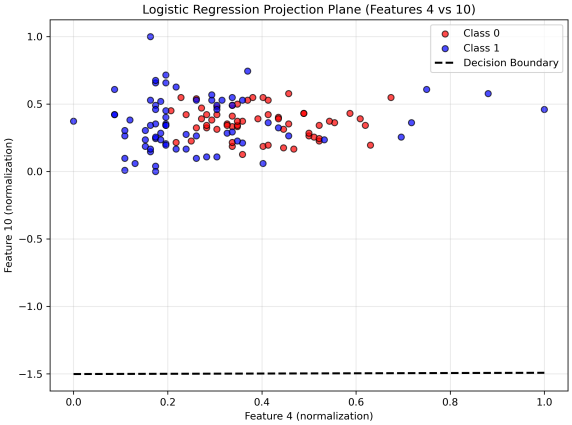

Figure 9 Decision boundary projection on features 4 vs 10

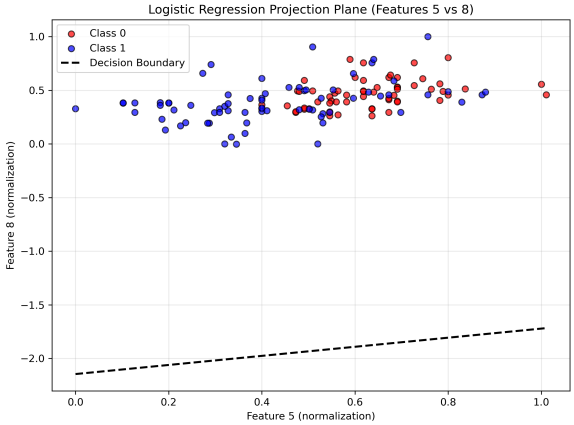

Figure 10 Decision boundary projection on features 5 vs 8

```
f1, f2 = 5, 9 # features
X_plot = X[:, [f1, f2]]
Y_plot = Y

x_min = Xtraining.min(axis=0)
x_max = Xtraining.max(axis=0)
X_plot_norm = (X_plot - x_min[[f1, f2]]) / (x_max[[f1, f2]] - x_min[[f1, f2]])

w0 = weight1[0]
w1 = weight1[1 + f1]
w2 = weight1[1 + f2]

plt.figure(figsize=(8, 6))
colors = ["red", "blue"]
labels = ["Class 0", "Class 1"]
for ii, label in enumerate(np.unique(Y_plot)):
    plt.scatter(
        X_plot_norm[Y_plot == label, 0],
        X_plot_norm[Y_plot == label, 1],
        color=colors[ii],
        label=labels[ii],
        alpha=0.7,
        edgecolor="k",
    )

x1_min, x1_max = X_plot_norm[:, 0].min(), X_plot_norm[:, 0].max()
x1s = np.linspace(x1_min, x1_max, 200)
x2s = -(w0 + w1 * x1s) / w2

plt.plot(x1s, x2s, "k--", linewidth=2, label="Decision Boundary")

plt.xlabel(f"Feature {f1} (normalization)")
```

```
plt.ylabel(f"Feature {f2} (normalization)")
plt.title(f"Logistic Regression Projection Plane (Features {f1} vs {f2})")
plt.legend()
plt.grid(alpha=0.3)
plt.tight_layout()
plt.savefig(
    f"images/Logistic_Regression_Projection_plane_f{f1}_f{f2}.png",
    dpi=300,
    bbox_inches="tight",
)
plt.show()
plt.close()
print(weight1)
```

4 Conclusion

This study successfully demonstrates the application of logistic regression for binary wine classification, distinguishing between two wine cultivars (classes 1 and 2) after excluding class 3 from the original Wine dataset. The key findings and achievements are summarized as follows:

Model Performance

The logistic regression model achieves **perfect classification performance** on the test set, with 100% accuracy, precision, recall, and F1 score. All 39 test samples were correctly classified with zero false positives and zero false negatives, demonstrating the model's exceptional ability to generalize to unseen data. This outstanding performance is attributed to:

- The strong discriminative power of the 13 chemical features
- Complete linear separability between the two wine classes
- Effective min-max normalization that scales features appropriately
- Successful convergence to the global optimum by both optimization methods