Logistic Regression — Coding Practice Questions

CM52054: Foundational Machine Learning
Coding practice set with fully worked answers

1) **Implement the sigmoid and logit functions.**
Write Python functions that:

   a) Implement the sigmoid function $\sigma(z) = 1/(1 + e^{-z})$.

   b) Implement the logit function $\text{logit}(p) = \log\left(\frac{p}{1-p}\right)$.

   c) Test that $\text{logit}(\sigma(z)) \approx z$ for a range of values of z.

   Answer:

```python
import numpy as np

def sigmoid(z):
    """
    Compute the logistic sigmoid function sigma(z) = 1 / (1 + exp(-z)).

    Parameters
    ----------
    z : float or np.ndarray
        Input value or array of values.

    Returns
    -------
    np.ndarray
        Sigmoid of the input, with the same shape as `z`.
    """
    # Use the standard definition: sigma(z) = 1 / (1 + exp(-z)).
    # NumPy will automatically apply exp element-wise if z is an array.
    return 1.0 / (1.0 + np.exp(-z))


def logit(p):
    """
    Compute the logit function logit(p) = log(p / (1 - p)).

    Parameters
    ----------
    p : float or np.ndarray
        Probabilities in the open interval (0, 1).

    Returns
    -------
    np.ndarray
        Logit of the input probabilities.
    """
```

```python
    # Convert input to a NumPy array for convenience in vectorised operations.
    p = np.asarray(p)

    # Basic input check:
    # We require probabilities to be strictly between 0 and 1 to avoid log(0).
    if np.any(p <= 0) or np.any(p >= 1):
        raise ValueError("All probabilities must be strictly between 0 and 1.")

    # Apply the logit transform element-wise.
    return np.log(p / (1.0 - p))


# ---- Simple test: logit(sigmoid(z)) \approx z ----

# Create a range of test values for z.
z_values = np.linspace(-5, 5, num=11)  # 11 points from -5 to 5

# Apply sigmoid, then logit.
sig_values = sigmoid(z_values)
reconstructed_z = logit(sig_values)

print("Original z values:       ", z_values)
print("Reconstructed z values:", reconstructed_z)
print("Difference:              ", reconstructed_z - z_values)
```

Listing 1: Sigmoid and logit implementations with a simple test

Explanation:

- The `sigmoid` function directly implements $1/(1 + e^{-z})$.
- The `logit` function applies $\log(p/(1 - p))$ and checks that p lies strictly in $(0, 1)$ to avoid numerical issues.
- The small test checks that $\operatorname{logit}(\sigma(z))$ approximately recovers z, up to floating point rounding.

2) **Implement logistic loss and gradient.**
Consider the logistic regression model with

$$p_i = \sigma(w^\top x_i), \qquad \ell(w) = -\frac{1}{N}\sum_{i=1}^{N}\big[y_i \log p_i + (1 - y_i)\log(1 - p_i)\big].$$

Assume:

- $X \in \mathbb{R}^{N\times M}$ has rows $x_i^\top$,
- $y \in \{0, 1\}^N$,
- $w \in \mathbb{R}^M$.

a) Implement a function that returns the average logistic loss $\ell(w)$.

b) Implement the gradient $\nabla_w \ell(w) = -\frac{1}{N}\sum_{i=1}^{N}(y_i - p_i)x_i$.

Answer:

```python
def logistic_loss_and_grad(w, X, y):
    """
    Compute the average logistic loss (binary cross-entropy) and its gradient.

    We assume a logistic regression model:
        p_i = sigmoid(w^T x_i)
    and loss:
        ell(w) = -1/N * sum_i [ y_i log(p_i) + (1 - y_i) log(1 - p_i) ].

    Parameters
    ----------
    w : np.ndarray of shape (M,)
        Parameter vector of the model.
    X : np.ndarray of shape (N, M)
        Design matrix; each row is a sample x_i.
    y : np.ndarray of shape (N,)
        Binary labels in {0, 1}.

    Returns
    -------
    loss : float
        The average logistic loss over the N samples.
    grad : np.ndarray of shape (M,)
        Gradient of the loss with respect to w.
    """
    # Number of samples.
    N = X.shape[0]

    # Compute linear scores z_i = w^T x_i for all samples at once.
    # X has shape (N, M), w has shape (M,), so X @ w -> (N,).
    z = X @ w

    # Convert scores into probabilities p_i = sigma(z_i).
    p = sigmoid(z)  # shape (N,)

    # Clip probabilities to avoid log(0) when computing the loss.
    eps = 1e-12 # 1*10^{-12}
    p_clipped = np.clip(p, eps, 1.0 - eps)

    # Compute average binary cross-entropy:
    #   loss = -1/N * sum_i [ y_i log(p_i) + (1-y_i) log(1-p_i) ].
    loss = -np.mean(y * np.log(p_clipped) + (1 - y) * np.log(1 - p_clipped))

    # Compute the gradient:
    #   grad = -1/N * sum_i (y_i - p_i) x_i.
    # First compute the vector of differences (y_i - p_i).
    diff = y - p  # shape (N,)

    # Then accumulate using X^T @ diff:
    #   X^T has shape (M, N); diff has shape (N,).
    #   Result is shape (M,), as desired.
    grad = -(1.0 / N) * (X.T @ diff)

    return loss, grad
```

Listing 2: Logistic loss and gradient (vectorised)

Explanation:

- We compute the vector of scores $\mathbf{z} = Xw$ and then apply the sigmoid to get probabilities p_i.
- Probabilities are clipped to avoid numerical issues when taking logs.
- The loss is implemented using vectorised operations and averaged over all samples.
- For the gradient, we build the vector $(y_i - p_i)$ and accumulate $\sum_i (y_i - p_i)x_i$ via $X^\top$`diff`.

3) **Train logistic regression with batch gradient descent.**
Write a function that:

   a) Takes (X, y), an initial weight vector w_{init}, a learning rate α, and a number of iterations `num_iters`.

   b) Performs batch gradient descent on the logistic loss:

$$w_{t+1} = w_t - \alpha \nabla_w \ell(w_t).$$

   c) Returns the final weights and the history of loss values.

Reuse `logistic_loss_and_grad` from the previous question.

Answer:

```python
def train_logistic_regression(X, y, w_init, alpha=0.1, num_iters=1000):
    """
    Train logistic regression via batch gradient descent.

    Parameters
    ----------
    X : np.ndarray of shape (N, M)
        Training data matrix.
    y : np.ndarray of shape (N,)
        Binary labels in {0, 1}.
    w_init : np.ndarray of shape (M,)
        Initial parameter vector.
    alpha : float, optional
        Learning rate (step size) for gradient descent.
    num_iters : int, optional
        Number of gradient descent iterations.

    Returns
    -------
    w : np.ndarray of shape (M,)
        Final parameter vector after training.
    loss_history : list of float
        List of loss values recorded at each iteration.
    """
    # Copy initial weights so we don't modify the caller's array.
    w = w_init.copy()

    # Store loss values to inspect convergence behaviour afterwards.
```

```python
    loss_history = []

    for t in range(num_iters):
        # Compute current loss and gradient for the current parameter vector.
        loss, grad = logistic_loss_and_grad(w, X, y)

        # Record loss.
        loss_history.append(loss)

        # Gradient descent update (we subtract because we minimise the loss).
        w = w - alpha * grad

        # Optional: print progress every fixed number of iterations.
        if (t + 1) % 100 == 0:
            print(f"Iteration {t+1:4d} / {num_iters}, loss = {loss:.4f}")

    return w, loss_history
```

Listing 3: Batch gradient descent for logistic regression

Explanation:

- The loop runs a fixed number of iterations of gradient descent.
- At each step, we compute both the current loss and its gradient.
- The update rule $w \leftarrow w - \alpha \nabla \ell(w)$ moves us in the direction of steepest descent of the loss.
- The list `loss_history` can be used to plot or check convergence.

4) **Prediction and accuracy for logistic regression.**
Implement the following functions:

a) `predict_proba(X, w)` returning predicted probabilities $p(y = 1 \mid x_i)$.

b) `predict_labels(X, w, threshold)` returning predicted labels in $\{0, 1\}$.

c) `accuracy(y_true, y_pred)` computing classification accuracy.

Answer:

```python
def predict_proba(X, w):
    """
    Predict class-1 probabilities p(y=1 | x) for all samples in X.

    Parameters
    ----------
    X : np.ndarray of shape (N, M)
        Data matrix.
    w : np.ndarray of shape (M,)
        Parameter vector.

    Returns
    -------
    p : np.ndarray of shape (N,)
        Predicted probabilities for class 1.
```

```
    """
    # Compute the linear scores z = Xw for all samples.
    z = X @ w

    # Convert scores to probabilities using the sigmoid function.
    p = sigmoid(z)
    return p


def predict_labels(X, w, threshold=0.5):
    """
    Predict binary labels in {0, 1} for all samples in X using logistic regression.

    Parameters
    ----------
    X : np.ndarray of shape (N, M)
        Data matrix.
    w : np.ndarray of shape (M,)
        Parameter vector.
    threshold : float, optional
        Decision threshold on the predicted probability for class 1.
        If p >= threshold, predict 1; otherwise 0.

    Returns
    -------
    y_pred : np.ndarray of shape (N,)
        Predicted labels in {0, 1}.
    """
    # Get predicted probabilities for each sample.
    p = predict_proba(X, w)

    # Compare each probability to the threshold to get class labels.
    y_pred = np.where(p >= threshold, 1, 0)
    return y_pred


def accuracy(y_true, y_pred):
    """
    Compute classification accuracy: fraction of correct predictions.

    Parameters
    ----------
    y_true : np.ndarray of shape (N,)
        True labels.
    y_pred : np.ndarray of shape (N,)
        Predicted labels.

    Returns
    -------
    acc : float
        Accuracy in [0, 1].
    """
    # Convert inputs to NumPy arrays for safety.
    y_true = np.asarray(y_true)
```

```
    y_pred = np.asarray(y_pred)

    # Accuracy is simply the fraction of positions where y_true == y_pred.
    return np.mean(y_true == y_pred)
```

Listing 4: Prediction utilities: probabilities, labels, accuracy

Explanation:

- `predict_proba` just performs the forward pass of the model.
- `predict_labels` thresholds probabilities at a user-specified value (default 0.5).
- `accuracy` calculates the fraction of correct predictions.

5) **End-to-end training on synthetic data.**
Write a small script that:

   a) Generates synthetic 2D data with a bias term, using a known "true" parameter vector w_{true}.

   b) Samples labels from Bernoulli probabilities defined by this w_{true}.

   c) Trains logistic regression with `train_logistic_regression`.

   d) Prints the learned weights and the training accuracy.

Answer:

```
import numpy as np

# 1) Generate synthetic data.
np.random.seed(42)  # Fix seed for reproducibility.

N = 200  # Number of samples.

# True parameters for synthetic data, including bias term:
# w_true = [bias, w1, w2].
w_true = np.array([-0.5, 2.0, -1.0])

# Generate 2D features x1, x2 from a standard normal distribution.
X_raw = np.random.randn(N, 2)  # Shape (N, 2).

# Add a column of ones to encode the bias term as a feature.
# Resulting X has shape (N, 3): [1, x1, x2].
X = np.hstack([np.ones((N, 1)), X_raw])

# Compute linear scores and probabilities according to w_true.
z_true = X @ w_true
p_true = sigmoid(z_true)

# Sample labels y from Bernoulli(p_true).
# For each sample i, y_i ~ Bernoulli(p_true[i]).
y = (np.random.rand(N) < p_true).astype(int)

print("First 5 synthetic labels:", y[:5])

# 2) Train logistic regression via gradient descent.

# Initial weights, e.g., all zeros.
```

```python
w_init = np.zeros(X.shape[1])

# Hyperparameters for training.
alpha = 0.1         # Learning rate.
num_iters = 1000   # Number of gradient descent steps.

w_learned, loss_history = train_logistic_regression(
    X, y, w_init, alpha=alpha, num_iters=num_iters
)

print("\nLearned weights:", w_learned)
print("True weights:    ", w_true)

# 3) Evaluate accuracy on the training set.

y_pred = predict_labels(X, w_learned, threshold=0.5)
train_acc = accuracy(y, y_pred)

print(f"\nTraining accuracy: {train_acc * 100:.2f}%")
```

Listing 5: End-to-end logistic regression on synthetic data

Explanation:

- A true parameter vector w_{true} defines the ground-truth decision boundary.
- 2D features are drawn from a Gaussian, and a bias column of ones is added.
- Probabilities are computed via $\sigma(Xw_{\text{true}})$, and labels are sampled from the corresponding Bernoulli distributions.
- Logistic regression is trained starting from zero weights. The learned weights are compared to the true ones, and training accuracy is reported.

6) **Optional: Visualise the learned decision boundary.**
Extend the script from the previous question to:

   a) Plot the 2D data points in the (x_1, x_2) plane, coloured by class label.

   b) Overlay the decision boundary given by $w_0 + w_1x_1 + w_2x_2 = 0$ for the learned parameter vector w.

Answer:

```python
import matplotlib.pyplot as plt

def plot_decision_boundary(X_raw, y, w):
    """
    Plot 2D data points and the decision boundary for a logistic
       regression model.

    Parameters
    ----------
    X_raw : np.ndarray of shape (N, 2)
        The original 2D features (without the bias column).
    y : np.ndarray of shape (N,)
        Binary labels in {0, 1}.
    w : np.ndarray of shape (3,)
        Learned parameters [bias, w1, w2].
    """
```

```python
    # Build masks for each class to plot them differently.
    class0 = (y == 0)
    class1 = (y == 1)

    # Create a new figure for plotting.
    plt.figure(figsize=(6, 5))

    # Scatter plot for class 0 samples.
    plt.scatter(
        X_raw[class0, 0], X_raw[class0, 1],
        marker='o', alpha=0.7, label='Class 0'
    )

    # Scatter plot for class 1 samples.
    plt.scatter(
        X_raw[class1, 0], X_raw[class1, 1],
        marker='s', alpha=0.7, label='Class 1'
    )

    # Extract parameters: w = [b, w1, w2].
    b, w1, w2 = w

    # Decision boundary satisfies b + w1*x1 + w2*x2 = 0.
    # Solve for x2 in terms of x1: x2 = -(b + w1*x1) / w2.
    x1_min, x1_max = X_raw[:, 0].min() - 1.0, X_raw[:, 0].max() + 1.0
    x1_vals = np.linspace(x1_min, x1_max, 100)

    # If w2 is very close to zero, the boundary is almost vertical.
    if np.abs(w2) < 1e-8:
        x1_boundary = -b / w1
        plt.axvline(x=x1_boundary, linestyle='--', label='Decision boundary')
    else:
        x2_vals = -(b + w1 * x1_vals) / w2
        plt.plot(x1_vals, x2_vals, 'k--', label='Decision boundary')

    plt.xlabel('x1')
    plt.ylabel('x2')
    plt.legend()
    plt.title('Logistic Regression Decision Boundary')
    plt.grid(True)
    plt.show()


# Usage with the synthetic data and learned parameters from Q5:
plot_decision_boundary(X_raw, y, w_learned)
```

Listing 6: Plotting the 2D data and learned decision boundary

Explanation:

- Data points for the two classes are plotted with different markers, making the separation visible.
- The decision boundary is obtained from the equation $w_0 + w_1 x_1 + w_2 x_2 = 0$.
- We either plot x_2 as a function of x_1, or draw a vertical line if the boundary is nearly vertical.