Lab Record: Deep Learning Models and Applications

Name: _____

Register Number: _____

Department: _____

Section: _____

Specialisation: _____

Topic 1 : Implementing Logistic Regression from Scratch

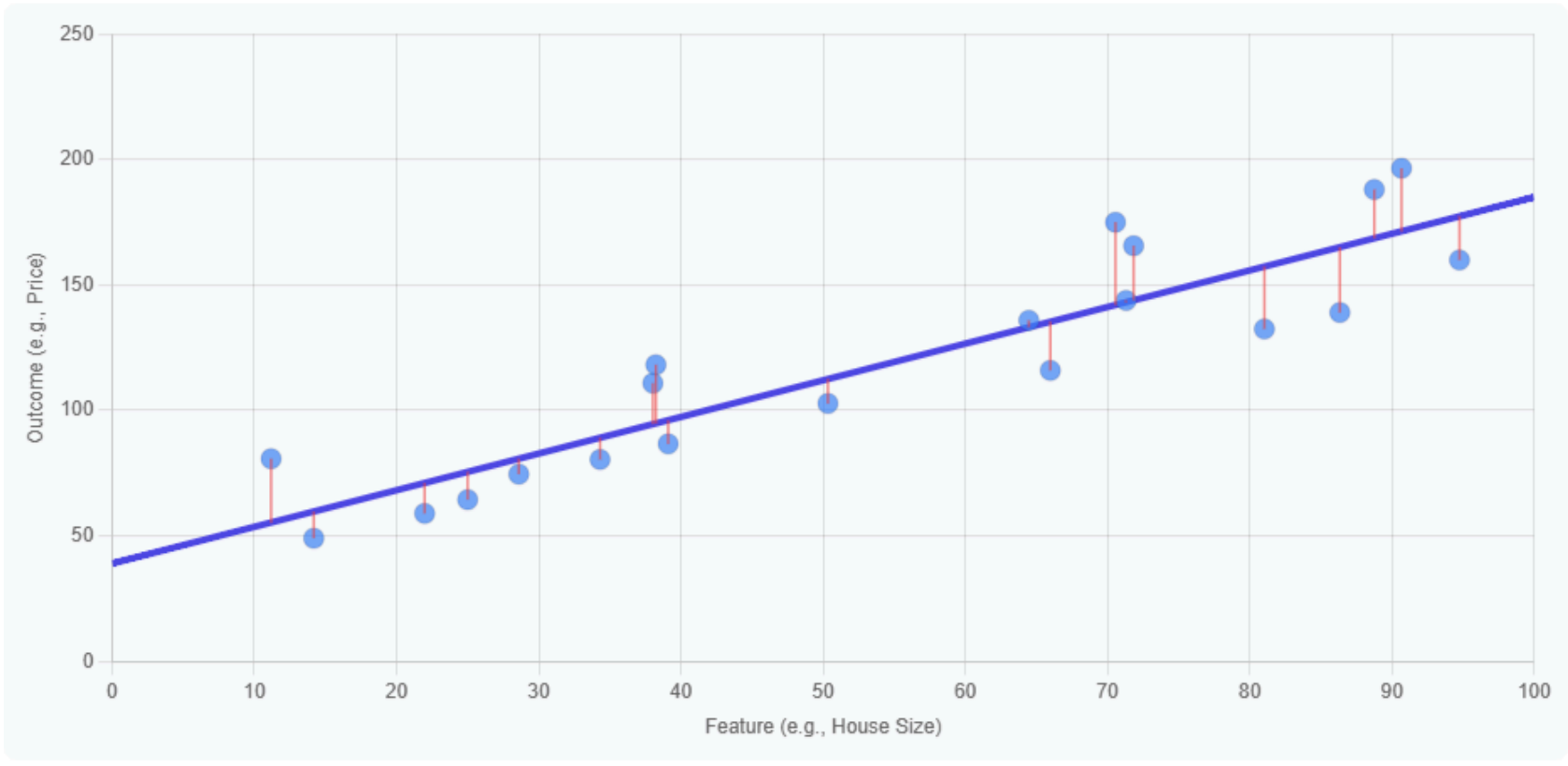
Objective: To build a complete logistic regression classifier from the ground up using only core Python. This exercise focuses on understanding the mathematical and algorithmic foundations without relying on external machine learning libraries.

Prerequisites: A basic understanding of Python lists, loops, functions, and classes.

Section 1: The Foundation - The Linear Hypothesis

Question 1: What is the fundamental intuition behind a linear model?

Answer: The intuition is to model a direct, linear relationship between input features and an output. Think of predicting a house's price based on its square footage. A linear model assumes that for every additional square foot, the price increases by a fixed amount. This creates a straight-line relationship. This line is our "model" or "hypothesis."



Question 2: How is this linear relationship represented mathematically? Explain the formula and its components.

Answer: The relationship is defined by the **linear hypothesis function**. For a single feature, it is:

$$h(X) = \theta_0 + \theta_1 X_1$$

This is the familiar equation of a line (`y = c + mx`).

- $h(X)$: The predicted output value. [also; "the **Hypothesis**" (rings a bell?)]
- X_1 : The input feature value (e.g., square footage).
- θ_1 (`theta_1`): This is the **weight** or **coefficient**. It corresponds to the slope (`m`) and defines how much the output `h(X)` changes for a one-unit change in X_1.
- θ_0 (`theta_0`): This is the **bias** or **intercept**. It corresponds to the *y-intercept* (`c`) and is the *baseline prediction* when all input features are zero.

Code:

```
# Linear Hypothesis for single variables as we have discussed in the class
def h(theta_0, theta_1, X):
    """
    A minimalist implementation of the fundamental Linear Hypothesis.
    Input: theta_0, theta_1, X
```

```
    Output: theta_0 + theta_1 * X
    """
    return theta_0 + theta_1*X
```

To handle multiple features (e.g., square footage, number of bedrooms), we generalize the formula into a weighted sum, which we'll call `z`:

$z = \theta_0 + \sum_{i=1}^{n}$

$=> z = \theta_0 + \theta_1 X_2 + \theta_n X_2 + \ldots + \theta_n X_n$

Question 3: How do we implement this generalized linear formula in core Python?

Answer: We can write a function that takes the bias (`theta_0`), a list of weights (`theta_weights`), and a list of feature values for a single sample (`x_sample`) and computes `z`.

```
def _compute_z(theta_0, theta_weights, x_sample):
    """
    Computes the linear part: z = theta_0 + (theta_weights . x_sample)
    p.s. We have used theta_1 and X as the parameter names during the class.
    the purpose of which is to make you understand that whichever names a variable might be of,
    the concepts remain the same.
    """
    # First, check for compatible lengths |
    # i.e., "Cardinality" as discussed during the class.

    if len(theta_weights) != len(x_sample):
        raise ValueError("""Mismatch in length of weights and features\n
                            Or, Mismatch in the length of theta_1 and X""")

    z = theta_0
    for i in range(len(theta_weights)):
        z += theta_weights[i] * x_sample[i]
    return z
```

Section 2: Understanding Inputs and Outputs

Question 4: What is the expected structure of the input data (`X`) and target data (`y`) for a model?

Answer:

- **Input Data (`X`):** The input is a list of lists. The outer list contains all the data samples, and each inner list contains the feature values for one specific sample.
    - *Example:* `X_data = [[1500, 3], [2100, 4], [1200, 2]]` (3 houses with sq. footage and bedroom count).
- **Target Data (`y`):** The target is a single list where each element corresponds to the true outcome for each sample in `X`.
    - *Example:* `y_data = [450000, 600000, 310000]` (the prices for the 3 houses).

Question 5: What kind of output does a linear model produce, and why is it unsuitable for classification tasks?

Answer: A standard linear model, as defined above, outputs an unbounded, continuous numerical value (the value `z`). For example, it could predict `42.7`, `981.5`, or `-150.2`.

This is perfect for **regression** (predicting a quantity like price), but it is unsuitable for **classification**. In classification, we need to predict a discrete class, like "Spam" (1) or "Not Spam" (0). An output of `-150.2` is meaningless in this context. We need to convert the unbounded output into a probability between 0 and 1.

Section 3: From Linear to Logistic Regression

Question 6: What is the primary modification needed to convert a linear regression model into a logistic regression model?

Answer: The primary modification is to take the unbounded linear output `z` and "squash" it into a value that lies between 0 and 1. This is achieved by feeding `z` into an **activation function**. For logistic regression, this function is the **Sigmoid Function**. The resulting output can then be interpreted as a probability.

Question 7: What is the formula for the Sigmoid function, and how does it behave?

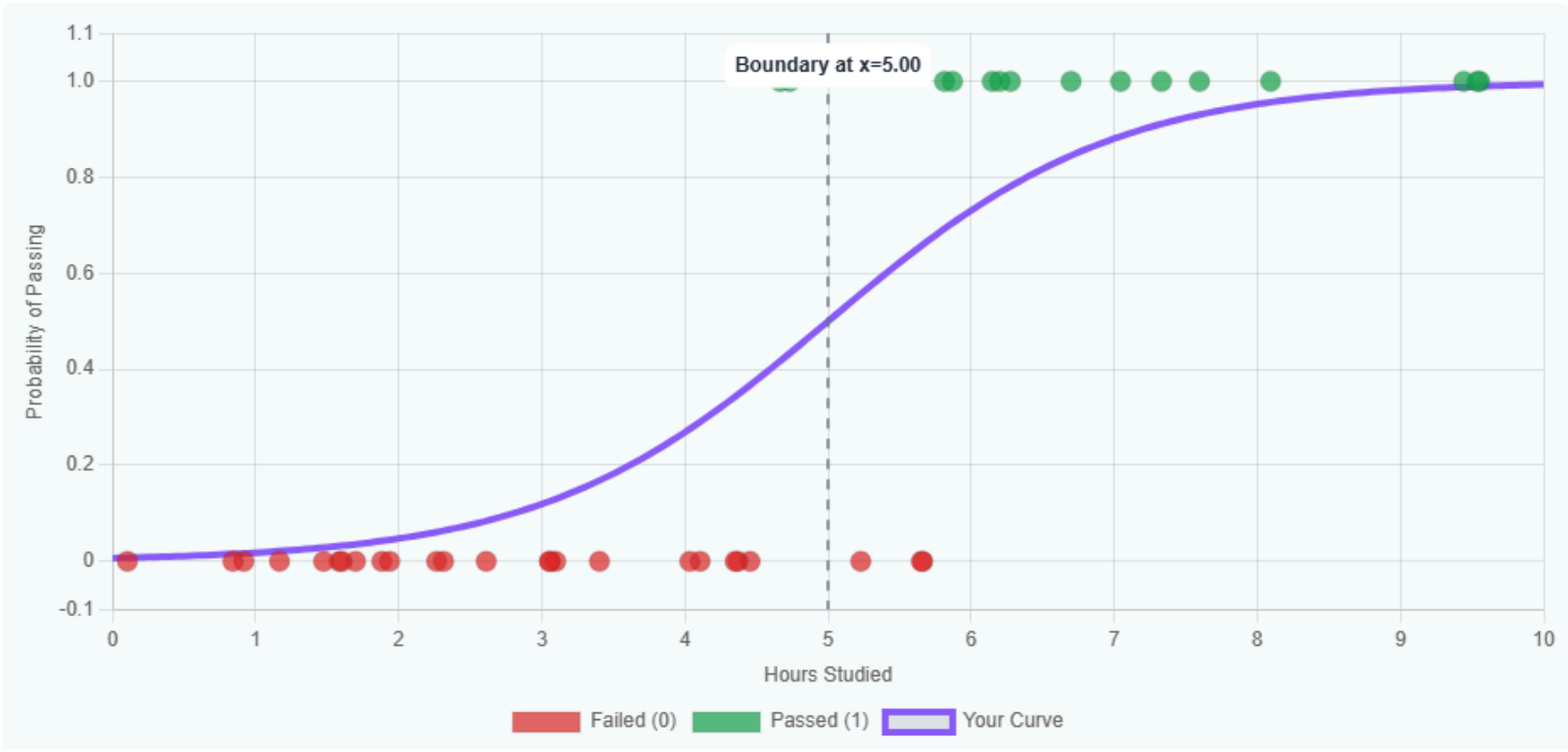
Answer: The Sigmoid (or Logistic) function is defined as:

$g(z) = 1/(1 + e^{-z})$

Where e is Euler's number. It has a characteristic "S" shape:

- As z becomes very large positive, e^{-z} approaches 0, so $g(z)$ approaches $1/(1 + 0) = 1$.
- As z becomes very large negative, e^{-z} approaches infinity, so $g(z)$ approaches $1/\infty = 0$.
- When $z = 0$, $e^0 = 1$, so $g(z)$ is $1/(1 + 1) = 0.5$.

The new hypothesis for logistic regression is $h(x) = g(z)$.



Question 8: Show the Python implementation of the Sigmoid function and the new hypothesis.

Answer: The Python implementation includes guards against numerical `OverflowError` that can occur when `z` is very large or small.

```
import math

def _sigmoid(z):
    """
    The Sigmoid activation function.
    Includes guards for numerical stability to prevent overflow.
    """
    if z > 700:  # e^(-700) is effectively 0
        return 1.0
    elif z < -700: # e^(700) is effectively infinity
        return 0.0
    else:
        return 1.0 / (1.0 + math.exp(-z))
```

The new hypothesis function simply combines the linear calculation with the sigmoid activation:

```
def _predict_probability(theta_0, theta_weights, x_sample):
    """
    Our full hypothesis: h(x) = sigmoid(z)
    """
    z = _compute_z(theta_0, theta_weights, x_sample)
    return _sigmoid(z)
```

Section 4: The Probabilistic Interpretation of the Sigmoid

Question 9: The Sigmoid function produces a number between 0 and 1. How can we be sure this represents a valid probability and isn't just an arbitrary value?

Answer: The choice of the Sigmoid function is not arbitrary. It has a direct mathematical relationship with probability through the concepts of **Odds** and **Log-Odds**. This connection ensures that its output is a principled probability.

Question 10: Define Probability, Odds, and Log-Odds. What are their mathematical formulas and ranges?

Answer:

1. **Probability (p):** The likelihood of an event occurring.
    - **Formula:** `p`

- **Range:** `[0, 1]`

2. **Odds:** The ratio of the probability of an event happening to it *not* happening.
   - **Formula:** `Odds = p / (1-p)`
   - **Range:** `[0, ∞)` (e.g., an odds of 4 means 4-to-1 likelihood).
3. **Log-Odds (Logit):** The natural logarithm of the odds.
   - **Formula:** `Log-Odds = log(p / (1-p))`
   - **Range:** `(-∞, +∞)`

The crucial insight is that the range of Log-Odds is unbounded, just like the output `z` from our linear model.

Question 11: Derive the Sigmoid function from the core assumption of Logistic Regression.

Answer: The core assumption of Logistic Regression is that **the log-odds of an event are a linear function of the input features.**

This allows us to set our two unbounded values equal to each other: $log(p/(1-p)) = z$

Now, we can algebraically solve for `p` (probability) to find the function that links them:

1. **Exponentiate both sides:** $e^{log(p/(1-p))} = e^z => p/(1-p) = e^z$
   [*We have already discussed about the formulae in class*]
2. **Multiply by** $(1-p)$**:** $p = e^z * (1-p)$
3. **Distribute:** $p = e^z - p * e^z$
4. **Isolate** `p` **terms:** $p + p * e^z = e^z$
5. **Factor out** `p`**:** $p(1+e^z) = e^z$
6. **Solve for** `p`**:** $p = e^z/(1+e^z)$
7. **Divide numerator and denominator by** `e^z`**:** $p = (e^z/e^z)/((1/e^z) + (e^z/e^z))$ -> $p = 1/(e^{-z}+1)$

This derivation proves that $p = 1/(1+e^{-z})$, which is exactly the Sigmoid function. This confirms it is the correct function for converting the linear model's output z into a valid probability p.

Section 5: The Complete Model and Lab Experiment

Question 12: How are all these components assembled into a trainable model? Explain the role of the Cost Function and Gradient Descent.

Answer: To create a trainable model, we need two more components:

1. **Cost Function (`_compute_cost`):** A function to measure how "wrong" the model's predictions are compared to the true labels. For logistic regression, we use **Binary Cross-Entropy (Log Loss)**. It heavily penalizes predictions that are confidently wrong (e.g., predicting 0.99 when the true label is 0).
2. **Gradient Descent (`fit` and `_compute_gradients`):** An optimization algorithm that "learns" the best values for `theta_0` and `theta_weights`. It works by: a. Calculating the predictions for the entire dataset. b. Calculating the total cost (error). c. Computing the gradients (the direction of the steepest increase in cost). d. Updating the theta parameters in the *opposite* direction of the gradient, thereby nudging the model towards a lower cost. e. Repeating this process for a set number of iterations.

Question 13: Provide the complete, commented Python code for the `CoreLogisticRegression` class.

Answer:

```
import math

class CoreLogisticRegression:

    def __init__(self, learning_rate=0.01, n_iterations=1000):
        """Initializes the model's hyperparameters."""
        self.learning_rate = learning_rate
        self.n_iterations = n_iterations
        self.theta_0 = 0.0
        self.theta_weights = []
        self.cost_history = []

    def _sigmoid(self, z):
        """Computes the sigmoid function with numerical stability."""
        if z > 700: return 1.0
        elif z < -700: return 0.0
        else: return 1.0 / (1.0 + math.exp(-z))

    def _compute_z(self, x_sample):
```

```
    """Computes the linear combination of inputs and weights."""
    z = self.theta_0
    for i in range(len(self.theta_weights)):
        z += self.theta_weights[i] * x_sample[i]
    return z

def _predict_probability(self, x_sample):
    """Makes a probability prediction for a single sample."""
    z = self._compute_z(x_sample)
    return self._sigmoid(z)

def _compute_cost(self, y_true, y_pred_probs):
    """Computes the binary cross-entropy (log loss)."""
    m = len(y_true)
    if m == 0: return 0.0
    total_cost, epsilon = 0.0, 1e-9
    for i in range(m):
        h = max(epsilon, min(1.0 - epsilon, y_pred_probs[i])) # Clipping
        cost_sample = -y_true[i] * math.log(h) - (1 - y_true[i]) * math.log(1 - h)
        total_cost += cost_sample
    return total_cost / m

def _compute_gradients(self, X_data, y_true, y_pred_probs):
    """Computes the gradients of the cost function."""
    m, n_features = len(y_true), len(self.theta_weights)
    grad_theta_0 = 0.0
    grad_theta_weights = [0.0] * n_features
    for i in range(m):
        error = y_pred_probs[i] - y_true[i]
        grad_theta_0 += error
        for j in range(n_features):
            grad_theta_weights[j] += error * X_data[i][j]
    grad_theta_0 /= m
    for j in range(n_features): grad_theta_weights[j] /= m
    return grad_theta_0, grad_theta_weights

def fit(self, X_data, y_data, verbose=True):
    """Trains the model using batch gradient descent."""
    n_features = len(X_data[0])
    self.theta_0 = 0.0
    self.theta_weights = [0.0] * n_features
    self.cost_history = []

    for i in range(self.n_iterations):
        y_pred_probs = [self._predict_probability(x) for x in X_data]
        cost = self._compute_cost(y_data, y_pred_probs)
        self.cost_history.append(cost)
        grad_theta_0, grad_theta_weights = self._compute_gradients(X_data, y_data, y_pred_probs)

        self.theta_0 -= self.learning_rate * grad_theta_0
        for j in range(n_features):
            self.theta_weights[j] -= self.learning_rate * grad_theta_weights[j]

        if verbose and i % (self.n_iterations // 10) == 0:
            print(f"Iteration {i}: Cost = {cost:.4f}")

def predict_proba(self, X_data):
    """Predicts probabilities for new data."""
    return [self._predict_probability(x) for x in X_data]

def predict(self, X_data, threshold=0.5):
    """Predicts class labels (0 or 1) based on a threshold."""
    probabilities = self.predict_proba(X_data)
    return [1 if prob >= threshold else 0 for prob in probabilities]
```

Question 14: How do we test this model? Provide the setup for a sample experiment.

Answer: We can test the model on a simple, linearly separable dataset. In this example, we'll try to predict if a student passed an exam (`y=1`) based on the hours they studied (`X`).

```
if __name__ == "__main__":

    print("--- Testing CoreLogisticRegression ---")

    # 1. Create a simple dataset
    X_train = [[1.0], [1.5], [2.0], [2.5], [4.5], [5.0], [5.5], [6.0]]
    y_train = [0, 0, 0, 0, 1, 1, 1, 1] # Students who studied > 4 hours passed

    # 2. Initialize and train the model
    model = CoreLogisticRegression(learning_rate=0.1, n_iterations=5000)
    print("Starting training...")
    model.fit(X_train, y_train)
    print("Training complete.")

    # 3. Print the final learned parameters
    print(f"\nFinal Bias (theta_0): {model.theta_0:.4f}")
    print(f"Final Weights (theta_1): {model.theta_weights[0]:.4f}")

    # 4. Make predictions on new, unseen data
    X_test = [[0.5], [3.0], [3.5], [7.0]]
    probs = model.predict_proba(X_test)
    labels = model.predict(X_test)

    print("\n--- Test Results ---")
    for i in range(len(X_test)):
        print(f"Input: {X_test[i][0]} hours | "
              f"Prob(Pass): {probs[i]:.4f} | "
              f"Prediction: {labels[i]}")
```

Section 6: Analysis and Conclusion

Question 15: What is the expected output of the experiment, and how should it be interpreted?

Answer: The expected output will show:

1. **Training Progress:** A series of printouts showing the `Cost` decreasing over iterations, indicating that the model is learning successfully.
2. **Final Parameters:** The learned values for `theta_0` (bias) and `theta_weights`. For this problem, `theta_1` should be a positive number (more hours -> higher chance of passing).
3. **Test Results:** Predictions on the test data.
    - For `0.5` hours, the probability should be very low (prediction: 0).
    - For `7.0` hours, the probability should be very high (prediction: 1).
    - For `3.0` and `3.5` hours, the probability will be near `0.5`. This point is the **decision boundary** that the model has learned to separate the two classes.

Question 16: What is the main limitation of this core Python implementation, and what have we learned by building it from scratch?

Answer: The main limitation is **speed**. The heavy use of Python `for` loops makes this implementation computationally slow, especially on large datasets. Libraries like NumPy and Scikit-learn perform these calculations using highly optimized, low-level code (often C or Fortran) through "vectorized" operations, which are orders of magnitude faster.

By building it from scratch, we have learned the exact mechanics that these high-level libraries hide from us. We now understand precisely how the linear model output is transformed into a probability, how error is measured, and how gradient descent uses that error to iteratively improve the model's parameters. This provides a deep and fundamental understanding of how a classifier actually works.

Question 17: What happens if you change the `learning_rate` or `n_iterations`?

Answer: This is an open question for experimentation.

- **High Learning Rate (e.g., `1.5`):** The model may fail to learn. The cost might explode (`NaN`) because the updates to theta are too large, overshooting the minimum cost every time.
- **Low Learning Rate (e.g., `0.0001`):** The model will learn very slowly. After 5000 iterations, the cost may still be high, and the model's predictions will be poor. It would need many more iterations to reach a good solution.
- **Low `n_iterations` (e.g., `100`):** The model will not have enough time to minimize the cost, resulting in an "underfit" model that has not learned the patterns in the data effectively." model that has not learned the patterns in the data effectively.

```
1 Start coding or generate with AI.
```