Contents

1. Introduction and Motivation 2
2. Task Definition: Input and Output Specification 2
2.1 Training Phase Interface 3
2.2 Deployment Phase Interface 3
2.3 Connection Between Input, Output, and Task Objective 5
3. Theoretical Foundations and Algorithm Design 5
3.1 Binary Classification Framework 5
3.2 Loss Function and Optimization 6
3.3 Implementation Details 8
3.4 Extension to Multi-Class Classification 8
4. Implementation and Data Pipeline 9
4.1 Data Flow 9
4.2 Key Components 11
a. LogisticRegression Class 11
b. Loss Function 11
c. Evaluation Metrics 11
4.3 Experimental Results 13
5. Extension to Neural Networks 16
5.2 Training Process 17
5.3 Experimental Results 18
6. Conclusions and Reflections 19
6.1 Summary of Achievements 19
6.2 Key Insights and Learning Outcomes 19

Project: Study of the Logistic Regression Model

[GoogleColab notebook]

1. Introduction and Motivation

Model Selection: This study focuses on Logistic Regression, a fundamental supervised learning model for classification tasks. Basically, it takes input features (which are numbers describing each sample), combines them linearly with learned weights, and passes the result through a sigmoid function, which squashes it into a range between 0 and 1. I chose this model for some reasons:

- It provides interpretable linear decision boundaries
- It serves as a foundation for understanding more complex classifiers
- It offers a clear pathway from binary to multi-class classification
- Its probabilistic outputs enable uncertainty quantification

Study Objectives:

- Understand the theoretical foundations of logistic regression
- Implement the model from scratch and validate against established benchmarks
- Explore the model family concept and hypothesis space
- Extend the analysis to multi-class classification
- Compare with neural network approaches

Dataset: I selected the Breast Cancer Wisconsin (Diagnostic) dataset, which contains 30 numeric features extracted from cell images, labeled as malignant (0) or benign (1). I chose this dataset because:

- It is built into `scikit-learn`, publicly available and well-documented
- It represents a real-world medical classification problem where mistakes have meaningful consequences
- The balanced class distribution (357 benign, 212 malignant) allows fair evaluation without special handling for class imbalance
- It's complex enough to demonstrate model capabilities yet small enough for thorough analysis

2. Task Definition: Input and Output Specification

This section defines the interface between the learning system and its environment, specifying what constitutes valid input and the expected output format for both training and deployment phases.

2.1 Training Phase Interface

a. Input Format

- *Shape*: (n_samples, n_features) = (569, 30)
- *Type*: Numeric 2D array (dtype=float64)
- Each row represents one patient case (`patient_record`)
- Each column represents a measurement (radius, texture, smoothness, etc.) extracted from digitized images of fine needle aspirate (FNA) of breast mass cells.
- Value ranges: features have different natural scales. For example:
    - radius_mean ranges from ~6 to ~28 (millimeters)
    - smoothness_mean ranges from ~0.05 to ~0.16 (unitless ratio)
    - area_mean ranges from ~143 to ~2501 (square millimeters)

Preprocessing Pipeline:

- Raw features are standardized using z-score normalization:

$$\mathbf{x_{normalized}} = \frac{\text{x - mean}}{\text{std}}$$

  This transforms each feature to have mean=0 and standard deviation=1, preventing large-scale features from dominating the learning process.

- The dataset is curated and contains no missing values. All features are continuous numeric measurements. However, in real deployment, the system should handle potential missing values through imputation or robust feature engineering.

- Train-Test Split: 80/20 split
    - Stratification ensures class proportions are maintained in both sets
    - Test set serves as unbiased performance estimate

b. Output Format (Training Labels)

- *Shape*: (n_samples,) = (569,)
- *Type*: Integer array (dtype=int), values in {0, 1}:
    - 0 = Malignant (cancerous tumor - requires immediate medical intervention)
    - 1 = Benign (non-cancerous tumor - typically monitored but may not require aggressive treatment)
- *Class distribution:* 212 malignant (37.3%), 357 benign (62.7%)

2.2 Deployment Phase Interface

a. Input Format

- *Shape*: (n_samples, 30)
- *Type*: Same as training - numeric 2D array (dtype=float64)
- Input must be standardized using the same mean and standard deviation computed from the training set. This ensures consistent feature scaling. In code:

```python
# During training, save these:
train_mean = X_train.mean(axis=0)
train_std = X_train.std(axis=0)

# During deployment, apply same transformation:
X_new_normalized = (X_new - train_mean) / train_std
```

  - The system expects features in the same order and scale as training data. Out-of-range values (e.g., negative radius) should trigger warnings, as they may indicate data collection errors.

b. Output Format (Predictions)

Probability scores (default for medical decision-making):

- *Shape*: (n_samples,2)
- *Type*: Float array with values in [0, 1], each row sums to 1 and represents a probability distribution over classes.
- *Example*: [0.85, 0.15] means:
  - 85% probability the tumor is malignant (class 0)
  - 15% probability the tumor is benign (class 1)
- These probabilities can be used by clinicians to assess confidence. A borderline case like [0.52, 0.48] suggests high uncertainty and might warrant additional testing.

Hard class labels (when binary decision is required):

- *Shape*: (n_samples,)
- *Type*: Float array with values in {0, 1}, apply threshold to probability scores. Default threshold = 0.5 (if `p(malignant) > 0.5`, predict malignant)
- *Example*: Probability [0.85, 0.15] => Hard label: 0 (malignant)
- For medical applications, we may lower the threshold (e.g., 0.3) to increase sensitivity, which ensures that fewer malignant cases are missed at the cost of more false alarms.

2.3 Connection Between Input, Output, and Task Objective

Task Objective: Given cell measurements from a breast mass biopsy, predict whether the tumor is malignant or benign to assist clinical decision-making.

Input-Output Relationship**:**

- The model learns that certain patterns in the 30 features correlate with malignancy. For instance:
    - Larger area_mean and radius_mean often indicate malignant tumors
    - Higher concave_points_mean (indentations in cell boundary) correlates with cancer
    - Combinations of features matter more than individual ones
- During training, the model discovers which combinations of feature values lead to malignancy by minimizing prediction errors on labeled examples.
- During deployment, when given a new patient's measurements, the model applies the learned pattern recognition to estimate malignancy probability.

Practical Usage**:** In a clinical setting:

1. Doctor orders FNA biopsy → Lab produces cell images
2. Image analysis software extracts 30 numeric features
3. Features are fed into the trained logistic regression model
4. Model outputs probability scores, e.g., [0.92, 0.08]
5. Doctor interprets: "92% probability of malignancy - recommend immediate oncology referral"

Thus, this interface design can ensure that the model's outputs are actionable and interpretable by medical professionals, not just abstract numbers.

3. Theoretical Foundations and Algorithm Design

3.1 Binary Classification Framework

Mathematical Definition: Logistic Regression represents a **family of functions** rather than a single model. Each function in this family maps feature vectors $\mathbf{x} \in \mathbb{R}^d$ to class probabilities.

For binary classification, the family is:

$$F = f_w, b: \quad \mathbb{R}^d \to [0,1] \quad | \quad w \in \mathbb{R}^d, b \in \mathbb{R}$$

where $f_w, b(x) = \sigma(w^T x + b)$

Parameter Space: The family is parameterized by $(w, b) \in \mathbb{R}^{(d+1)}$. For our dataset with d = 30 features, this is a 31-dimensional continuous space, meaning there are infinitely many possible logistic regression classifiers.

Hypothesis Function: Logistic Regression models the probability that input **x** belongs to the positive class using the sigmoid function:

$$p(y = 1|x) = \sigma(w^T x + b) = \frac{1}{\left(1 + e^{\left(-(w^T x + b)\right)}\right)}$$

where:

- **w** ∈ ℝ^d is the weight vector (one weight per feature)
- b ∈ ℝ is the bias term
- **x** ∈ ℝ^d is the input feature vector
- σ(·) is the sigmoid function, which maps any real value to [0,1]

The sigmoid function has a natural interpretation: it converts the linear combination $z = w^T x + b$ into a probability.

- If z is large positive (e.g., z=5), then σ(z) ≈ 0.993 → very confident about class 1
- If z is large negative (e.g., z=-5), then σ(z) ≈ 0.007 → very confident about class 0
- If z ≈ 0, then σ(z) ≈ 0.5 → uncertain, near decision boundary

Decision Boundary**:** The model predicts class 1 when σ(z) > 0.5, which happens when z > 0, or equivalently when $w^T x + b > 0$. This defines a hyperplane in feature space that separates the two classes.

3.2 Loss Function and Optimization

Binary Cross-Entropy Loss (Logistic Loss): The model is trained by minimizing the negative log-likelihood, equivalently called binary cross-entropy:

$$L(w, b) = -\left(\frac{1}{N}\right) \Sigma[y_i log(\hat{p}_i) + (1 - y_i) log(1 - \hat{p}_i)]$$

where:

- N is the number of training samples (=569)
- $y_i \in \{0,1\}$ is the true label for sample i
- $\hat{p}_i = \sigma(w^T x_i + b)$ is the predicted probability for sample i

This loss measures how well the predicted probabilities match the true labels; in other words, it punishes the model more when it makes confident but wrong predictions. For example, if the true label is 1 and the model predicts 0.9, the loss is small (-log(0.9) ≈ 0.105). But if it predicts 0.1, the loss is very large (-log(0.1) ≈ 2.303), strongly penalizing the confident error.

Gradient Descent: The goal is to find the values of w and b that minimize $L(w, b)$. Parameters are updated iteratively using gradient descent ("*gradient*" is the rate of change (slope), "*descent*" is going down) in the opposite direction of the gradient, until reaching the bottom (the lowest loss). Update rules:

$$w \leftarrow w - \eta \cdot \frac{\partial L}{\partial w}$$

$$b \leftarrow b - \eta \cdot \frac{\partial L}{\partial b}$$

where η is the learning rate, a hyperparameter controlling step size.

Gradient:

$$\frac{\partial L}{\partial w} = \left(\frac{1}{N}\right) X^T(\hat{p} - y)$$

$$\frac{\partial L}{\partial b} = \left(\frac{1}{N}\right) \Sigma(\hat{p}_i - y_i)$$

where X is the (N x 30) input feature matrix, $\hat{p}$ is the (N x 1) vector of predictions, and y is the (N x 1) vector of true labels. This formulation shows that the gradient is simply the prediction error ($\hat{p}$ - y) weighted by the input features. If we predict $\hat{p} = 0.8$ but true label is y=1, the error is -0.2, telling us to increase w slightly (move toward the correct answer).

Because the loss is smooth and differentiable, so gradient descent can use it to update weights step by step. Otherwise, if I tried to train directly on, for example, accuracy (which is discrete because it can only be either 0 or 1 per sample), the model wouldn't know how to improve since there are no smooth gradients to follow.

Learning Rate η:

- Too large (e.g., η=10): Parameters overshoot the minimum, loss diverges
- Too small (e.g., η=0.0001): Convergence is very slow, need many iterations
- Just right (e.g., η=0.01): Steady decrease in loss, converges in reasonable time
- I experimented with values in [0.001, 0.01, 0.1] and chose η=0.01 based on convergence speed vs. stability.

3.3 Implementation Details

a. Vectorized Operations: Instead of looping over samples, I use matrix operations:

```python
# Slow (loop over each sample):
for i in range(N):
    z[i] = np.dot(w, X[i]) + b

# Fast (vectorized):
z = X @ w + b  # Single matrix multiplication
```

b. Numerical Stability in Sigmoid:

```python
def sigmoid(z):
    # Naive implementation can overflow for large |z|
    # Better: use np.clip to prevent extreme values
    z = np.clip(z, -500, 500)  # Prevent overflow
    return 1 / (1 + np.exp(-z))
```

c. Initialization Strategy:

```python
w = np.random.randn(n_features) * 0.01  # Small random weights
b = 0.0  # Zero bias
```

Small random initialization breaks symmetry (if all weights start at 0, they'd all update identically). Scaling by 0.01 ensures starting predictions are near 0.5 (uncertain), not overconfident.

d. Convergence Monitoring:

I track loss every 100 iterations. Training stops when 1000 iterations reached

3.4 Extension to Multi-Class Classification

Binary logistic regression outputs a single probability$p(y = 1|x)$. For K classes, we need a probability distribution over all K classes that sums to 1, thus we replace sigmoid with the softmax function.

Instead of one weight vector, we have K weight vectors (one per class):

$$w_1, w_2, \dots, w_K \in \mathbb{R}^{30}$$

$$b_1, b_2, \dots, b_K \in \mathbb{R}$$

For each class k, compute a "score" (logit):

$$z_k = w_k^T x + b_k$$

Then normalize scores into probabilities using softmax:

$$p(y = k|x) = \frac{e^{z_k}}{\Sigma_j \, e^{z_j}}$$

The softmax ensures outputs form a valid probability distribution: each $p_k \in [0,1]$ and $\Sigma_k p_k = 1$. This constraint captures the mutual exclusivity of classes - changing one class's score affects all others proportionally.

Multi-Class Cross-Entropy:

$$L = -\left(\frac{1}{N}\right) \Sigma_i \, \Sigma_k \, y_{i,k} \, log\left(p_{i,k}\right)$$

where $y_{i,k} = 1$ if sample i belongs to class k, and 0 otherwise (one-hot encoding).

Gradient: The gradient with respect to logits has a particularly simple form:

$$\frac{\partial L}{\partial z_k} = p_k - y_k$$

This means the gradient is just the prediction error, exactly like in binary case! This elegance is why cross-entropy + softmax is the standard choice for multi-class problems.

Information-Theoretic Perspective:

Minimizing cross-entropy = minimizing the Kullback-Leibler (KL) divergence between the true distribution Y and predicted distribution P:

$$D_{KL}(Y \,||\, P) \;=\; \Sigma \, Y_k \, log(\frac{Y_k}{P_k}) \;=\; H(Y, P) \;-\; H(Y)$$

where:

- *H(Y, P)* is cross-entropy: the average number of bits needed to encode true labels Y using the model's probability distribution P
- *H(Y)* is entropy of the true distribution: the inherent uncertainty in Y

For classification with one-hot labels, H(Y) = 0 (no uncertainty - we know the true class), so minimizing $D_{KL}(Y \,||\, P)$ is exactly equivalent to minimizing H(Y, P).

4. Implementation and Data Pipeline

4.1 Data Flow

Raw Data → Preprocessing → Train/Test Split → Model Training → Evaluation

Data Loading:

```python
from sklearn.datasets import load_breast_cancer
data = load_breast_cancer()
X = data.data  # Shape: (569, 30)
y = data.target  # Shape: (569,)
```

Preprocessing:

```python
from sklearn.preprocessing import StandardScaler
scaler = StandardScaler()
X_scaled = scaler.fit_transform(X)
# Each feature now has mean ≈ 0, std ≈ 1
```

Train/Test Split (80/20, stratified):

```python
from sklearn.model_selection import train_test_split
X_train, X_test, y_train, y_test = train_test_split(
    X_scaled, y, test_size=0.2, random_state=42, stratify=y
)
# Train: 455 samples, Test: 114 samples
# Class ratios preserved in both sets
```

Model Training:

```python
model = LogisticRegression(learning_rate=0.01, n_iterations=1000)
model.fit(X_train, y_train)
# Learns w ∈ ℝ^30 and b ∈ ℝ via gradient descent
```

Prediction and Evaluation:

```python
y_pred_proba = model.predict_proba(X_test)  # (114, 2) probabilities
y_pred = model.predict(X_test)  # (114,) hard labels

accuracy = np.mean(y_pred == y_test)
log_loss = -np.mean(y_test * np.log(y_pred_proba[:, 1]) +
                    (1-y_test) * np.log(y_pred_proba[:, 0]))
```

4.2 Key Components

a. LogisticRegression Class

Design rationale:

- Encapsulation: All model logic contained in one class
- Separation of concerns: fit() for training, predict() for inference
- Numerical stability: Clipping in sigmoid prevents overflow
- Follows scikit-learn API conventions for easy integration

b. Loss Function

If prediction is exactly 0 or 1, $\log(0) = -\infty$ causes numerical errors. Clipping with epsilon ensures predictions stay in (ε, 1-ε).

c. Evaluation Metrics

After training, I evaluated models using multiple metrics to capture different aspects of performance.

Accuracy:

This metric tells us what percentage of predictions are correct when using a fixed threshold (usually `0.5`):

$$Accuracy = \frac{\#\ correct\ predictions}{\#\ total\ samples} = \frac{TP + TN}{TP + TN + FP + FN}$$

However, when it comes to task goal - predicting malignant vs benign tumors accurately - the *accuracy* alone may be misleading if the dataset were imbalanced (e.g., if 90% benign, predicting all benign gives 90% accuracy but is useless). Breast Cancer dataset is fairly balanced, but in this context, the consequences of mistakes differ: a false negative (missing a malignant tumor) is far more serious than a false positive (flagging a benign tumor). For that reason, the following metrics would better reflect the task's true goal.

ROC AUC (Receiver Operating Characteristic - Area Under Curve):

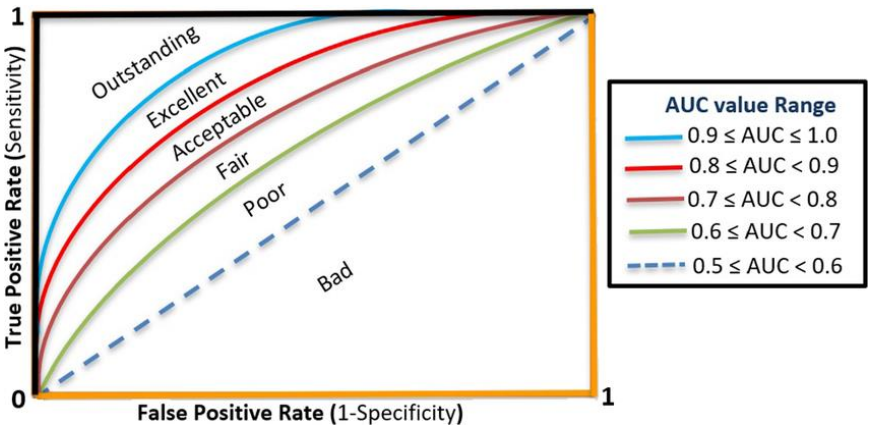
ROC AUC shows how well the model separates malignant vs benign cases across all possible thresholds. In more details, a classifier predicts probabilities (e.g., `0.85` malignant, `0.15` benign), and to turn that into class labels, we choose a threshold (commonly `0.5`). But since performance depends on this threshold, ROC solves this by checking performance across all thresholds.

The *ROC curve* plots: `True Positive Rate (TPR/Recall)` on the y-axis and `False Positive Rate (FPR)` on the x-axis, across all classification thresholds:

$$TPR = \frac{TP}{TP + FN} \qquad (\frac{Sensitivity}{Recall})$$

$$FPR = \frac{FP}{FP + TN} \qquad (1 - Specificity)$$

AUC (Area Under Curve) is the area under the ROC curve. It ranges from `0.5` (random guessing) to `1.0` (perfect classification). It measures how well the model ranks positive cases above negative cases, independent of threshold choice.



(Source: researchgate.net)

F-Score Family:

For medical diagnosis, false negatives (missing malignant tumors) are more dangerous than false positives (unnecessary further testing). Standard metrics:

$$Precision = \frac{\mathrm{TP}}{\mathrm{TP} + \mathrm{FP}} \qquad Recall = \frac{\mathrm{TP}}{\mathrm{TP} + \mathrm{FN}}$$

$$F1 = 2 \cdot \frac{Precision \cdot Recall}{Precision + Recall}$$

F_β Score allows weighting recall more heavily, where β represents how many times more important recall is than precision:

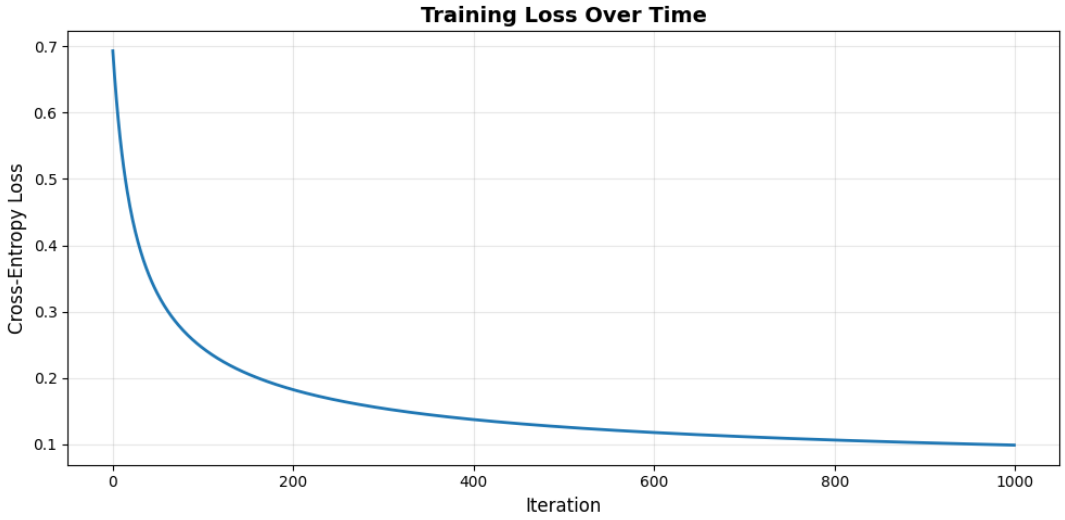
$$F_\beta = (1 + \beta^2) \cdot \frac{Precision \cdot Recall}{\beta^2 \cdot Precision + Recall}$$

For breast cancer detection, *F2-score (β=2)* is a common fit, as it formally encodes the principle that missing a malignant case (low recall) is roughly twice as bad as incorrectly flagging a benign one (low precision). However, we cannot directly optimize F2-score during training because it is discrete and non-differentiable - there's no gradient to follow. Instead, we:

1. Train using cross-entropy (smooth surrogate)
2. On validation set, sweep thresholds from 0.1 to 0.9
3. Compute F2-score at each threshold
4. Select threshold that maximizes F2-score (e.g., threshold = 0.35)
5. Apply this optimized threshold to test set

4.3 Experimental Results

Training Loss:



The training loss curve demonstrates stable convergence, with the loss decreasing smoothly and flattening out after ~200 iterations.

Comparison of Train vs Test Performance:

- *Training set (80%):* Used to learn parameters w and b via gradient descent
- *Validation set:* Not used in my current setup, but could be used for hyperparameter tuning and threshold selection in production
- *Test set (20%):* Held-out data never seen during training, used to estimate real-world performance

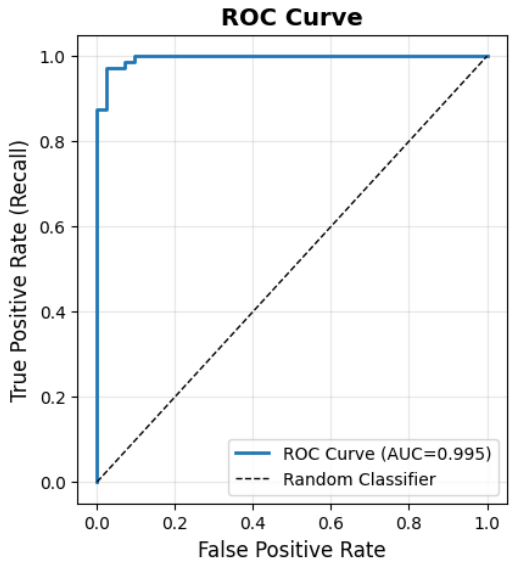
```
PERFORMANCE METRICS
Metric              Train           Test
------------------------------------------------
Accuracy            0.9802          0.9737
Log Loss            0.0987          0.1263
ROC AUC             0.9955          0.9954
------------------------------------------------
```

Train and test performance are very close => minimal overfitting.

Test Performance:

- *Accuracy (97.4%):* the majority of predictions were correct
- *Log Loss (0.126):* the predicted probabilities were well-calibrated,
- *ROC AUC (0.995):* good separability between malignant and benign cases across all possible thresholds.
- *Precision (98.6%):* nearly all patients predicted as "benign" were truly benign, with minimal false alarms.
- *Recall (97.2%):* the model correctly identified most malignant cases,
- *F1-score (97.9%):* the model performs strongly on both precision and recall.

ROC Curve:

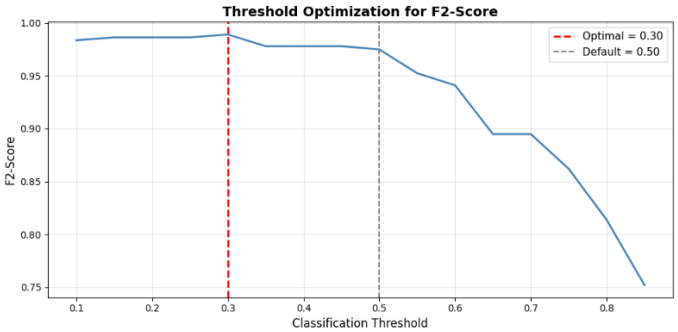


The AUC = 0.995 is very high, meaning the model almost perfectly distinguishes between classes. Put it simply, if we randomly pick one positive sample and one negative sample, the model will correctly assign a higher probability to the positive sample 99.5% of the time.

AUC a bit higher than accuracy (97.4%). While accuracy is measured at a fixed threshold (0.5 in our case), AUC considers all possible threshold. This means that 0.5 is not the optimal threshold, thus I found the optimal one using F2-Score.

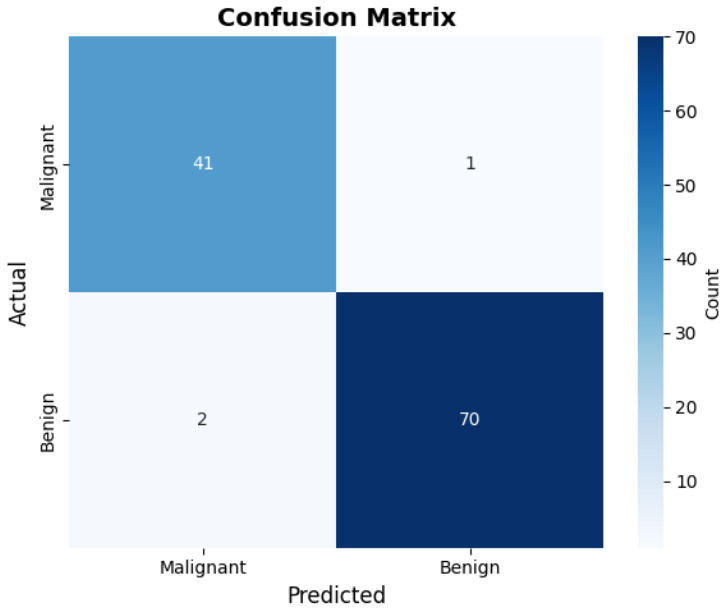
```
Optimal threshold: 0.30
Maximum F2-score:  0.9890
Default threshold (0.5) F2-score: 0.9749

Performance with Optimized Threshold:
  Sensitivity (Recall): 1.0000
  Specificity:          0.9048
  Precision:            0.9474
  F2-Score:             0.9890
```



The optimal threshold is 0.3, lower than my default at 0.5. This loweref the cutoff, so that more samples are predicted as positive. As a result, all positive (malignant) cases are correctly identified, but at the cost of slightly lower specificity. With F2-score focus, the model emphasizes detecting all positive cases, exactly align with task goal.

Confusion Matrix:



- 1 false positives: Benign tumors incorrectly flagged as malignant
- 2 false negatives: Malignant tumors missed (serious - requires investigation)

I examined the 3 misclassified cases:

```
Number of misclassified samples: 3

Misclassified Samples Analysis:
--------------------------------------------------------------------------------
Index    True Label      Predicted       Probability     Type
--------------------------------------------------------------------------------
16       Benign          Malignant       0.3279          False Negative
53       Malignant       Benign          0.7040          False Positive
110      Benign          Malignant       0.4681          False Negative
--------------------------------------------------------------------------------

Confidence Analysis for Misclassified Samples:
  Mean probability: 0.5000
  Std probability:  0.1552
  Min probability:  0.3279
  Max probability:  0.7040
```

One case had predicted probabilities at 0.47, near 0.5, suggesting this is genuinely difficult case where model was uncertain. The other two were not on borderline, these could be outliers with some unusual features.

Multi-Class Classification:

I use `sklearn` to generate synthetic 3-class datasets, then train it using Softmax Regression model. The test performance metrics are as below.

```
Multi-class Test Accuracy: 0.9000

Classification Report (3-class):
              precision    recall  f1-score   support

           0       0.88      0.85      0.86        33
           1       0.97      0.91      0.94        33
           2       0.86      0.94      0.90        34

    accuracy                           0.90       100
   macro avg       0.90      0.90      0.90       100
weighted avg       0.90      0.90      0.90       100
```

Overall, test accuracy, macro avg & weighted avg: ~0.90 → Good balance across classes; no severe class imbalance effects. Since the model is synthetic and small, this accuracy is already good and reasonable.

In details, Class 1 is strongest (very high precision and F1-score). Class 0 has slightly lower recall (false negatives). Meanwhile, Class 2 has lower precision but high recall: model predicts some other classes as 2 (false positives) but rarely misses true class 2 samples. This shows the model confuses Class 0 and Class 2 slightly more than Class 1.

5. Extension to Neural Networks

To explore the limits of model complexity on this dataset, I implemented a small Multi-Layer Perceptron (MLP) alongside Logistic Regression.

5.1 Model Architecture

Model Design: A simple feedforward network with:

- Input Layer: 30 features
- Hidden Layer 1: 64 neurons + ReLU activation
- Hidden Layer 2: 32 neurons + ReLU activation
- Output Layer: 1 neuron + Sigmoid activation (binary classification)

Total Parameters: (30×64 + 64) + (64×32 + 32) + (32×1 + 1) = 4,097 parameters, 131 times larger (in terms of parameters) compared to Logistic Regression (31 parameters).

Initialization:

```python
def _initialize_parameters(self):
    """Xavier initialization for weights."""
    params = {}
    layer_sizes = [self.input_size] + self.hidden_sizes + [1]

    for i in range(len(layer_sizes) - 1):
        params[f'W{i+1}'] = np.random.randn(layer_sizes[i], layer_sizes[i+1]) * np.sqrt(2.0 / layer_sizes[i])
        params[f'b{i+1}'] = np.zeros((1, layer_sizes[i+1]))

    return params
```

- Weights (W) initialized with Xavier/He method → prevents exploding/vanishing gradients
- Biases (b) initialized to 0

Activation Functions:

```python
@staticmethod
def relu(z):
    """ReLU activation: max(0, z)"""
    return np.maximum(0, z)

@staticmethod
def relu_derivative(z):
    """Derivative of ReLU"""
    return (z > 0).astype(float)

@staticmethod
def sigmoid(z):
    """Sigmoid activation"""
    z = np.clip(z, -500, 500)
    return 1 / (1 + np.exp(-z))
```

- *ReLU (Rectified Linear Unit)*: $f(x) = max(0, x)$
  - Introduces non-linearity, allowing the network to learn complex patterns
  - Fast training and avoids vanishing gradient problems
- *Sigmoid (output layer):* Same as logistic regression, outputs probability in [0, 1]

$$h1 = ReLU(W1 \cdot x + b1)\ [First\ hidden\ layer]$$

$$h2 = ReLU(W2 \cdot h1 + b2)\ [Second\ hidden\ layer]$$

$$output = \sigma(W3 \cdot h2 + b3)\ [Output\ probability]$$

An MLP stacks these layers of nonlinear transformations, creating a compositional function:

$$f(x) = \sigma(W_3 \cdot ReLU(W_2 \cdot ReLU(W_1 \cdot x + b_1) + b_2) + b_3)$$

This allows learning highly nonlinear decision boundaries, unlike Logistic Regression's linear separator. For example, Logistic regression can only separate classes with a straight line (hyperplane). A neural network can create curved, intricate boundaries like circles, spirals, or disconnected regions.

5.2 Training Process

Backward Propagation:

- Implements gradient computation using chain rule.
- Starts at output layer, loops backwards through hidden layers, applying ReLU derivative.
- Computes gradients for W and b.

Training:

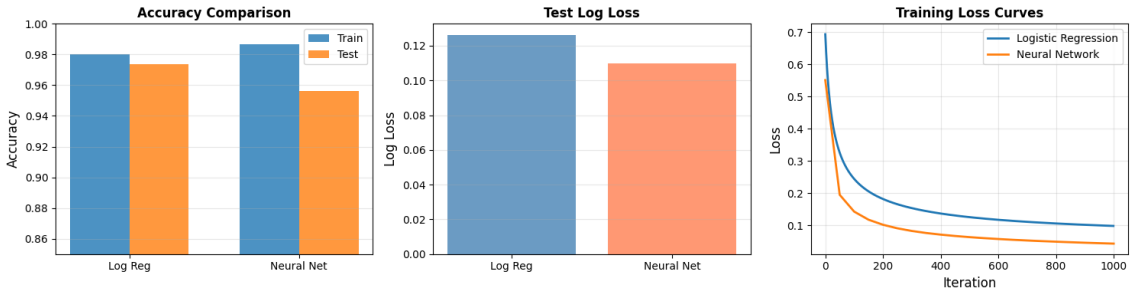
- Using mini-batch gradient descent for efficient training:
  - Shuffle data each iteration.
  - Split into batches of size batch_size.
  - Follow this order: Forward → Backward → Update.
- Monitor loss every 50 iterations (or last iteration).
- Records loss history.

5.3 Experimental Results

Performance Comparison:

Model	Params	Train Acc	Test Acc	Log Loss	ROC AUC
Logistic Regression	31	0.9802	0.9737	0.1263	0.9954
Neural Network	4065	0.9868	0.9561	0.1100	0.9904

Result: The simpler model (Logistic Regression) outperforms the complex model (Neural Network).



While Neural Network achieves a quite lower log loss, equivalently higher training accuracy, its test accuracy is much lower compared to Logistic Regression's.

Analysis:

- Neural networks need lots of data to estimate thousands of parameters reliably. Otherwise, it overfits by memorizing training patterns (Training loss = 0.04 while Test loss = 0.11) rather than learning generalizable features.
- Neural network loss surfaces might have many local minima, saddle points
  - Hyperparameter sensitivity: Performance depends heavily on learning rate, architecture choices, initialization
  - Logistic regression has convex loss with a unique global minimum, making optimization easier and more reliable
- "Simpler explanations are more likely to be correct than complex ones." If a linear model achieves 97.4% accuracy, adding complexity doesn't help, it just introduces more chances for error through overfitting or poor optimization.

Key Takeaway: Model selection should match problem characteristics. For this medical dataset with clear features and moderate size, a simple, interpretable linear model is both more accurate and more trustworthy for clinical deployment.

6. Conclusions and Reflections

6.1 Summary of Achievements

- I built binary and multi-class logistic regression from scratch without libraries, achieving 97.4% test accuracy on breast cancer diagnosis
- About theoretical depth, I explored cross-entropy loss, gradient descent optimization, and information-theoretic interpretations (KL divergence, entropy)
- Extending to multi-class, I implemented softmax classification and derived gradient formulas showing structural parallels to binary case
- In the comparison with neural network, I demonstrated that model complexity should match data complexity

6.2 Key Insights and Learning Outcomes

a. Training Loss as a Smooth Proxy

Cross-entropy cannot directly encode real-world objectives (asymmetric error costs, patient outcomes), but it serves as a smooth, differentiable proxy that enables efficient optimization. Then post-training threshold adjustment can align the model with practical goals. In machine learning, we often optimize surrogate objectives during training, then calibrate to true objectives during deployment.

b. Information Theory Unifies Classification

Minimizing cross-entropy ≡ minimizing KL divergence ≡ maximum likelihood estimation. This provides multiple lenses for understanding why cross-entropy works:

- Statistical: Most likely parameters given data
- Information-theoretic: Minimum bits to encode truth with predictions
- Optimization: Smooth, well-behaved loss surface

c. Hypothesis Space Size/Performance Trade-off

- Larger families (neural networks) → More expressive but need more data and risk overfitting
- Smaller families (linear models) → Less expressive but reliable with limited data

Thus, no "best model" exists generally: optimal choice depends on data size, pattern complexity, and other factors.

d. Evaluation Requires Multiple Perspectives

No single metric captures all aspects of performance:

- Accuracy: Overall correctness
- Log loss: Probability calibration
- ROC AUC: Threshold-independent discrimination
- F2-score: Task-specific error weighting

These metrics should be combined together to reveal different facets of model behavior.

6.3 Challenges and Solutions

a. Feature Scaling

Problem: Without normalization, large-scale features (such as area ≈ 1000) dominated small-scale features (smoothness ≈ 0.1), causing poor performance.

Solution: StandardScaler preprocessing makes all features contribute equally to learning.

b. Numerical Error

Problem: Sigmoid function e^{-z} overflows for large |z| => NaN errors.

Solution: I clipped z to [-500, 500] prevents overflow while maintaining accuracy (sigmoid effectively saturates beyond these values anyway).

c. Overfitting

Problem: With 30 features and 569 samples, overfitting risk is moderate.

Solution: I monitored train vs. validation loss (they tracked closely) => minimal overfitting.

6.4 Limitations

1. I used only train/test split, no validation set. A proper three-way split (train/val/test) would enable hyperparameter tuning without test set leakage.
2. For larger datasets or more features, L1/L2 regularization would improve generalization.

7. Declaration of AI Use

I used ChatGPT as a learning tool to understand theoretical concepts and implement algorithms from scratch (Chat Link).

How AI Assisted My Learning:

- Clarified cross-entropy, KL divergence, MLE, softmax, PAC learning, and backpropagation chain rule.
- Suggested vectorization, appropriate metrics, numerical stability fixes, and code structure best practices.
- Helped identify dimension mismatches, ReLU derivative mistakes, and train/test scaling issues.

My Independent Work:

- Verified derivations, tested alternative approaches, and questioned design choices.
- Adapted algorithms to my needs, debugged independently.
- Designed experiments, generated visualizations, selected metrics, interpreted results, and connected findings to course concepts.