Study of Logistic Regression Model

Kevin Nochez | 25461447

Google Collab:
https://colab.research.google.com/drive/1SfqEsO1yIOWiPo6cFirVb_doU2zjopJh?usp=sharing
ChatGPT Chat: https://chatgpt.com/share/68e64c5d-bde4-800c-a36b-c28a27e85fef

Journal Week 1 | Initial Setup
Focus: Project direction, problem selection, input/output formats, dummy function.
Expected Outcome: Clarify task and system expectations

1. **Project Focus**

The focus of this project is Option 1: "**Studying a fundamental machine learning model"**. Option 1's approach emphasises on developing a deep understanding of the theoretical and computational aspect of a specific learning algorithm, aiming to explore how and why a model works by building it from scratch. Emphasising on learning theory, understanding concepts such as hypothesis spaces, loss functions, and optimization algorithms will be covered as they are fundamental.

1.1 **Project Limitations**

Use of off-the-shelf machine learning libraries (such as scikit-learn, TensorFlow, or PyTorch) are not permitted for Option 1. Only basic programming and numerical libraries like NumPy, Matplotlib, and Pandas will be used. Though allowed if needed, a real-world dataset won't be used for analysis as the aim of this project is based on understanding the model rather than implementing one and solving a problem, so mostly toy datasets will be used.

1.2 **Expected Outcomes**

- Understanding and explain the mathematical background of the chosen model
- Implement the algorithm to showcase how theorical formulas translate into code
- Evaluate the model using toy dataset/s to demonstrate learning behaviour, loss convergence, and prediction outcomes.

2. **Chosen ML Model and Background**

The chosen algorithm for this project is: **Logistic Regression**.
Logistic regression is a simple, fast method for classification, most commonly binary classification (e.g., spam vs. not spam). Despite the name "regression," its output is a probability (between 0 and 1), which you can turn into a class label.

2.1 **Input & Output**

2.1.1 **Input**

In Logistic Regression, the model receives one or more input features that describe an observation. We represent these features as a vector $\boldsymbol{x} = \boldsymbol{x_1, x_2, \ldots, x_n}$, where each x_j is a numerical value such as height, weight, or temperature.

The model also includes a set of weights $(w_1, w_2, \ldots, w_n)$ and a bias term b, which together form a linear combination of the inputs:

$$\boldsymbol{z = w_1 x_1 + w_2 x_2 + \cdots + w_n w_n + b}$$

2.1.2 **Output**

The result z is a real number called the logit, which is then passed through the sigmoid function:

$$\hat{p} = \frac{1}{1 + e^{-z}}$$

This converts the logit into a probability value between 0 and 1. The output $\hat{\mathbf{y}}$ represents the model's prediction: if the probability is greater than 0.5, the model predicts class 1 (positive); otherwise, class 0 (negative). In this way, Logistic Regression transforms numerical input data into a probability that indicates how likely it is that the input belongs to the positive class.

2.2 Dummy Function

I asked ChatGPT to help me create a dummy function that mimics the structure of a Logistic Regression model. The goal was to verify that the model's input and output types were correct before implementing the actual algorithm. The function, named "`predict_disease()`", accepts a list of numerical features and outputs a binary class label, `1` for "disease present" and `0` for "no disease." It calculates the average of the input features and applies a threshold of 0.5 to simulate a decision boundary. I also generated a data loading and preprocessing function using the built-in *Breast Cancer Wisconsin* dataset from scikit-learn. The function loads the dataset, scales the features for consistency, and returns them in a format compatible with the dummy model.

```python
def predict_disease(features: List[float]) -> int:
    """
    Dummy function that imitates the behavior of a logistic regression model.
    Predicts 1 if average feature value > 0.5, else 0.
    """
    score = sum(features) / len(features)
    if score > 0.5:
        return 1  # "disease present"
    else:
        return 0  # "no disease"
```

*Comment - The entire script can be found in the google collab file under the "**Basic Dummy Function | W/ Data loading & Pre-processing function**" section*

To test the setup, I ran the dummy function on the first five samples of the dataset and compared the predictions with the actual results. The predictions were poor, which was expected since the dummy model does not learn weights, apply the sigmoid function, or perform any optimization it simply mimics the structure of Logistic Regression without the learning process.

```
First 5 Predictions vs Actual:
Sample 1: Predicted = 1, Actual = 0
Sample 2: Predicted = 0, Actual = 0
Sample 3: Predicted = 1, Actual = 0
Sample 4: Predicted = 1, Actual = 0
Sample 5: Predicted = 1, Actual = 0
```

My personal thoughts on why the results were poor:

- **No training or optimization:** The dummy model does not learn weights or bias from data, instead it simply uses a fixed threshold, so it cannot adapt to patterns in the dataset.
- **Too simple logic:** The model only checks if the average of the features is above 0.5, which is far too simple for a dataset with many features.
- **No understanding of the data:** The dummy function treats all features equally and ignores which ones are more important for predicting cancer.

Journal Week 2 | ML Framework
Focus: Mapping Logistic Regression & understanding its concepts
Expected Outcome: An overview of the Hypothesis, Loss, and Optimizer

To progress from the dummy model generated earlier, I asked ChatGPT to help me build a proper Logistic Regression model from scratch using only NumPy. The goal of this step was to replace the simplified function with a complete learning pipeline that includes the key components of the machine learning framework: the hypothesis function, loss function, and optimizer. By implementing these elements directly in code, I aimed to understand how predictions are generated, how model errors are measured, and how parameters are updated through optimization. This process also allowed me to identify and label the specific code blocks corresponding to each theoretical concept and reflect on how they interact within the full model.

1. **Hypothesis**

In Logistic Regression, the hypothesis represents how the model makes predictions from input data. It assumes a linear relationship between the input features and the log-odds of the output. Mathematically, this is expressed as:

$$\hat{y} = \sigma(w^T x + b)$$

Where w is the weight vector, b is the bias term, and $\sigma(z) = \frac{1}{1+e^{-z}}$ is the sigmoid function that convers the linear score $z = w^T x + b$ into a probability between 0 and 1. In other words, the model takes a list of numbers as input, multiplies each one by a weight, adds them all together (plus a small constant b), and then squashes the result into a number between 0 and 1. If that number is close to 1, the model predicts the sample likely belongs to the positive class; if it's close to 0, it predicts the opposite.

In the code, this idea is implemented through the functions *sigmoid(z)* and *predict_proba(X, w, b).* The *predict_proba* function computes the linear combination of the input features and applies the sigmoid to produce probabilities that indicate how likely each sample belongs to the positive class. This part of the code directly represents the model's forward pass, the stage where raw input data is transformed into interpretable predictions.

2. **Loss**

The loss function measures how far off the model's predictions are from the actual results. In Logistic Regression, this is done using the Binary Cross-Entropy Loss, which compares the predicted probability $\hat{y}$ with the true label y. The formula is:

$$L = -\frac{1}{m}\sum_{i=1}^{m}[\, y^{(i)} \log(\hat{y}^{(i)}) + (1 - y^{(i)}) \log(1 - \hat{y}^{(i)})\,]$$

Here, m is the number of samples, $y^{(i)}$ is the true label (0 or 1), and $y\hat{}^{(i)}$ is the predicted probability for that sample. The goal of training is to find the values of w and b that minimize this loss, meaning the predictions get as close as possible to the actual labels. In simpler terms, the loss tells us "*how wrong"* the model is. If the model predicts a probability close to the correct answer (for example, predicting 0.9 when the true label is 1), the loss is small. But if the model is very confident and wrong (predicting 0.9 when the true label is 0), the loss is large.

In the code, this is implemented in the function *compute_loss(y_true, y_pred)*. This function calculates the average Binary Cross-Entropy across all samples and includes a small adjustment (called *clipping*) to prevent taking the logarithm of zero, which would cause an error.

3. Optimizer

The optimizer is the part of the model that adjusts its internal parameters, specifically the weights (w) and bias (b) to make better predictions over time. In this project, the optimizer used is Gradient Descent, a method that updates the parameters by moving them in the direction that reduces the loss. Mathematically, the update rules are:

$$w := w - \eta \frac{1}{m} X^T(\hat{y} - y), \quad b := b - \eta \frac{1}{m} \sum (\hat{y} - y)$$

Here, η (eta) is the learning rate, which controls how big each update step is. If it's too large, the model might overshoot and fail to converge; if it's too small, learning becomes very slow. In other words, the optimizer works like trial and error, and it looks at how wrong the model's predictions are (the loss) and slightly changes the weights and bias to make the next predictions a bit better. Over many repetitions, the model gradually "learns" the correct parameter values.

In the code, this process happens inside the functions *gradient_descent(X, y, w, b, learning_rate)* and *train_logistic_regression(X, y, ...)*. The first computes the gradients and updates the parameters, while the second runs this process in a loop for many epochs, printing the loss along the way to show how the model improves over time.

4. Reflection and Questions

While building this Logistic Regression model, I started to see how the hypothesis, loss, and optimizer all connect to make the model learn. The hypothesis gives a prediction, the loss tells how far off the prediction is, and the optimizer slowly adjusts the model to do better. I still have some questions though. I wonder how much the learning rate really affects the model's learning speed and if there is a simple way to choose a good value for it. I also noticed that the loss sometimes goes up or down a little instead of always going down, and I am not sure if that is normal or if I did something wrong. Lastly, I am curious if the model could keep improving if I trained it for longer or if it would just stop getting better after a while.

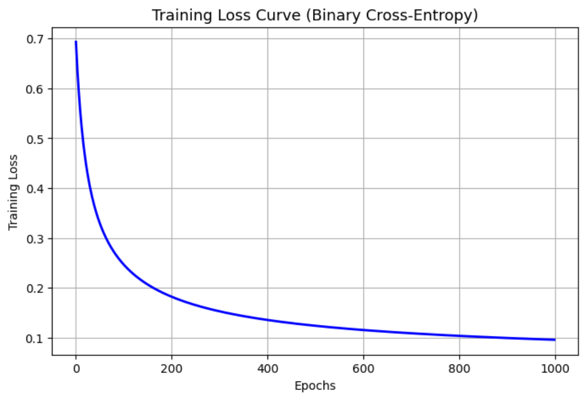
Journal Week 3 | Study the Loss (Criterion)
Focus: Studying and understanding the Loss
Expected Outcome: An in-depth understanding of Loss

1. Training and Evaluation Criteria

For training, the model uses the Binary Cross-Entropy (BCE**)** loss function, which measures how far the predicted probabilities are from the actual labels. This loss is minimized using gradient descent, allowing the model to adjust its weights and bias to make better predictions. For evaluation, the model uses accuracy, which measures how many predictions match the true labels. These two metrics are NOT the same, but the difference makes sense. BCE provides continuous feedback for optimization, helping the model learn smoothly, while accuracy gives a clear and simple way to measure overall performance after training.

If one was to replace BCE with Mean Squared Error (MSE)**,** the model would still function, but learning would be slower and less accurate because MSE is meant for regression, not classification.

Therefore, using BCE for training and accuracy for evaluation is both appropriate and effective for this task.



The graph above shows how the training loss decreases over time. The loss starts around 0.69, which is typical for random predictions, and steadily drops to about 0.10 by the final epoch. This consistent downward trend demonstrates that the optimizer is successfully minimizing the loss, and the model is learning effectively. The final accuracy of 96.49% supports this, showing that the model generalizes well to unseen data.

2. Does the evaluation criterion truly reflect the goals of the task?

In this task, the goal is to reflect on whether accuracy truly measures what matters for the cancer classification problem. While accuracy is simple and useful, it may not always be the best metric. In the Breast Cancer dataset, most samples are labelled as "*benign*", meaning the dataset is slightly imbalanced. Because of this, a model could achieve high accuracy even if it occasionally misses malignant cases, which are the most critical to detect.

In medical contexts, recall (also called *sensitivity*) is often more important than overall accuracy because it measures how many actual positive cases (malignant tumours) the model correctly identifies. High recall means the model rarely misses a cancer diagnosis, even if it sometimes predicts "malignant" when it shouldn't. On the other hand, improving recall can lower precision, since predicting more positives may include more false alarms.

Therefore, while accuracy gives a general idea of performance, it doesn't fully reflect the real-world goal of minimizing missed cancer cases. Metrics like precision, recall, or the F1-score (which balances both) would better capture how well the model aligns with the task's true objective of identifying malignant tumours correctly.

3. Possible Alternatives and Justification

Even though accuracy works well for this project, other evaluation metrics can provide a deeper understanding of the model's performance. One good alternative is to use Log Loss, which is the same as the Binary Cross-Entropy used during training. This would make the evaluation more consistent with the training process, as both would measure how confident the model's probability predictions are rather than just whether they are correct or not.

Another useful metric could be the ROC-AUC which is short for: *Receiver Operating Characteristic – Area Under the Curve* score. ROC-AUC evaluates how well the model separates the two classes across different decision thresholds, rather than using a fixed cutoff like 0.5. A higher AUC value means the model is better at distinguishing between malignant and benign cases.

This metric is especially useful in medical problems, where the cost of false negatives (missing a cancer case) is much higher than that of false positives.

4. Attempting To Apply New Criterion

Using an alternative evaluation metric like accuracy or F1-score directly as the training criterion is not feasible for this model. These metrics are not differentiable, meaning the model cannot calculate smooth gradients needed for optimization through gradient descent. In simple terms, the optimizer needs a continuous surface to "walk down" toward lower loss values, but metrics like accuracy change in sudden jumps when predictions switch from right to wrong.

This is why Binary Cross-Entropy (BCE) remains the preferred loss function for Logistic Regression. It provides a smooth, continuous surface that allows the optimizer to make small, precise adjustments to the weights and bias. Metrics such as accuracy, precision, recall, or F1-score are still very useful, but they are better suited for evaluation after training**,** not for the optimization process itself.

Journal Week 4 | Information-Theoretic Loss
Focus: Research on Information-Theoretic Loss
Expected Outcome: Understanding of Loss

1. Output Format of Q

In a multi-class classification problem, the model produces several output values instead of just one. These outputs represent the predicted probability for each class. For example, if there are three possible classes, the model's output can be written as $\hat{Q} = [q_1, q_2, q_3]$ where each q_i is the probability that the input belongs to class i. The values are calculated using the SoftMax function, which ensures that all probabilities are between 0 and 1 and add up to 1.

In other words, rather than choosing one class outright, the model gives a list of probabilities showing how confident it is in each option. For instance, if the model outputs [0.8,0.1,0.1][0.8, 0.1, 0.1][0.8,0.1,0.1], it means it is 80% confident in the first class and 10% confident in each of the others. This probability vector is then compared with the true class label during training using a suitable loss function, such as categorical cross-entropy.

2. Multi-Output Regression and Multi-Class Classification Loss
In multi-output regression, each output is independent, and the total loss is just the sum of individual losses, such as:

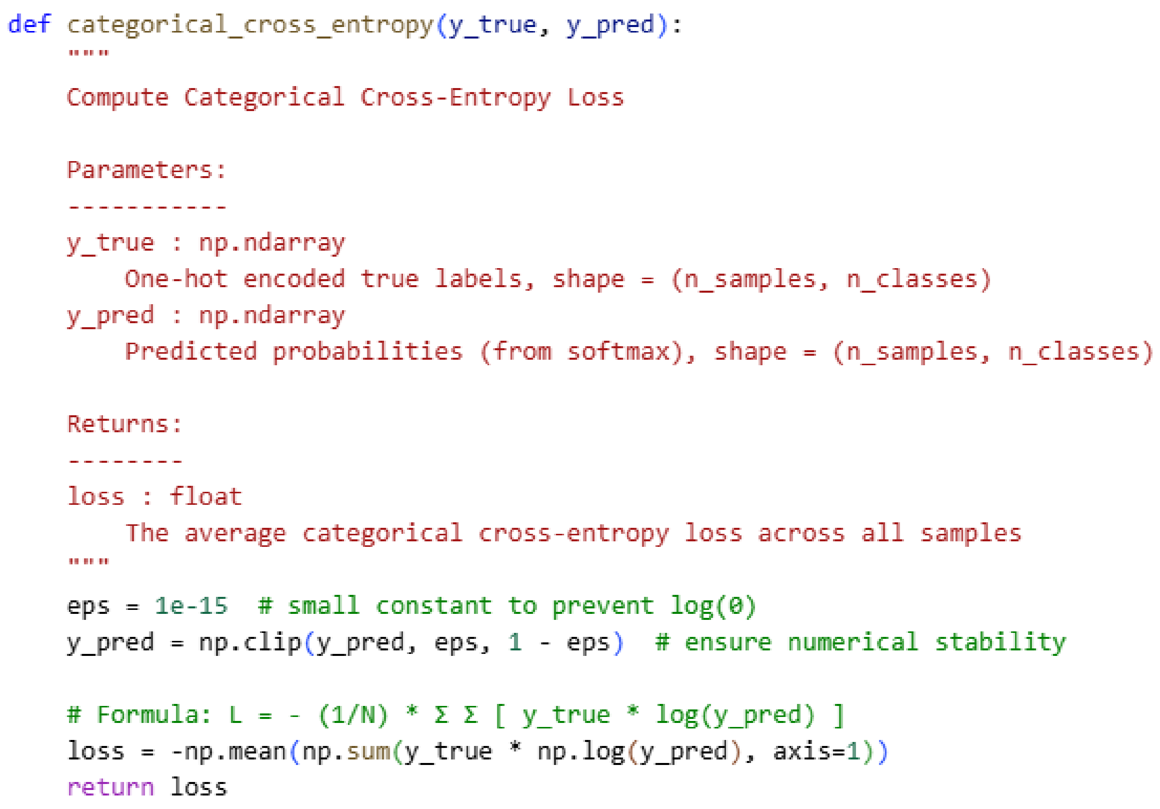
$$L_{\text{Multi-Real}} = L(q_1) + L(q_2) + L(q_3)$$

Ultimately, changing one output doesn't affect the others.

In multi-class classification, the outputs are connected through the softmax function, which forces all predicted probabilities to add up to 1. Increasing one class's probability decreases the others. This coupling makes the loss *information-theoretic*, since it measures how well the predicted probability distribution matches the true distribution of the classes.

3. Implementation of Multi-Class Classification Loss

To explore how loss functions work for multi-class problems, I asked ChatGPT to help me implement the categorical cross-entropy loss using NumPy. The function computes the average negative log-likelihood between the model's predicted probabilities and the true class labels. A small toy dataset with three samples and three classes was used to demonstrate how the function works. The predicted probabilities (y_{pred}) were compared with the one-hot encoded true labels (y_{true}), and the computed loss value was approximately 0.36. This shows that when the model's predictions are fairly close to the true labels, the resulting loss is low.

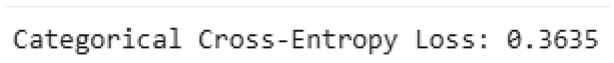


```python
def categorical_cross_entropy(y_true, y_pred):
    """
    Compute Categorical Cross-Entropy Loss

    Parameters:
    -----------
    y_true : np.ndarray
        One-hot encoded true labels, shape = (n_samples, n_classes)
    y_pred : np.ndarray
        Predicted probabilities (from softmax), shape = (n_samples, n_classes)

    Returns:
    --------
    loss : float
        The average categorical cross-entropy loss across all samples
    """
    eps = 1e-15  # small constant to prevent log(0)
    y_pred = np.clip(y_pred, eps, 1 - eps)  # ensure numerical stability

    # Formula: L = - (1/N) * Σ Σ [ y_true * log(y_pred) ]
    loss = -np.mean(np.sum(y_true * np.log(y_pred), axis=1))
    return loss
```



```
Categorical Cross-Entropy Loss: 0.3635
```

*Comment - The entire script can be found in the google collab file under the "**Categorical Cross-Entropy Loss (NumPy Implementation)**" section*

3.1 AI Explanation on Loss Implementation, and Verifying Output

I asked AI to explain how categorical cross-entropy connects to information theory. It described that entropy measures uncertainty in true labels, cross-entropy measures how costly it is to represent those labels using the model's predicted probabilities, and minimizing cross-entropy is equivalent to minimizing KL divergence between the true and predicted distributions. For one-hot labels, this becomes simply the negative log of the predicted probability for the correct class (e.g., -log(0.7) in my toy example).

Based on what on AI's explanation and my understanding on it, I came up with these questions:

- Does log base affect optimization or only units?
- Should class weights be used if the dataset is imbalanced?
- What changes if label smoothing is used?

Journal Week 5 | Model Family – Part 1
Focus: Family of Functions
Expected Outcome: Model Family Outcome

1. Logistic Regression as a Family of Functions

The Logistic Regression model can be described as a family of functions defined by

$$f_{w,b}(x) = \sigma(w^T x + b)$$

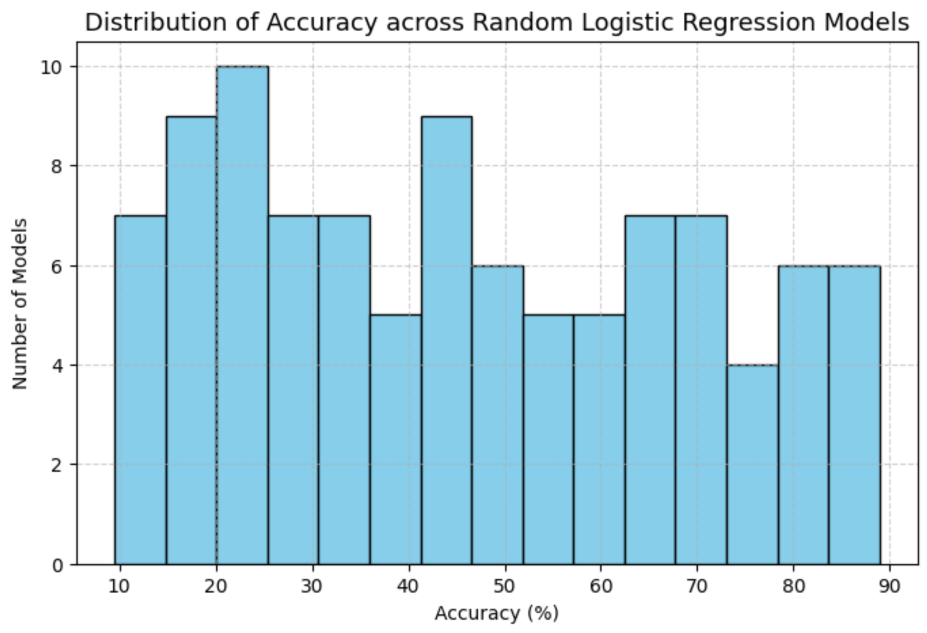
Where σ is the sigmoid function, w represents the weights, and b is the bias. Each unique combination of w and b produces a different function within this family.

The input of these functions is a numeric feature vector $x \in R^n$, and the output is a probability value $\hat{y} \in (0,1)$ representing the likelihood of the positive class. The parameters w and b are real numbers, meaning the parameter space of this family is R^{n+1}.

2. Exploring the Model Family

To explore the Logistic Regression model as a family of functions, I generated 100 random combinations of weight vectors (w) and bias values (b) without any training or optimization. Each combination represents a different function $f_{w,b}(x) = \sigma(w^T x + b)$ within the same model family. Using the Breast Cancer dataset, I calculated the prediction accuracy for each randomly sampled function.

The results showed a wide variation in performance, with accuracies ranging from 9.49% to 88.93%, and an average of about 45.96%. The histogram below shows this distribution, illustrating that most random functions perform poorly while a few happen to perform moderately well by chance. This demonstrates that while all functions share the same mathematical structure, only specific parameter combinations produce meaningful results, highlighting the importance of training to find optimal parameters.



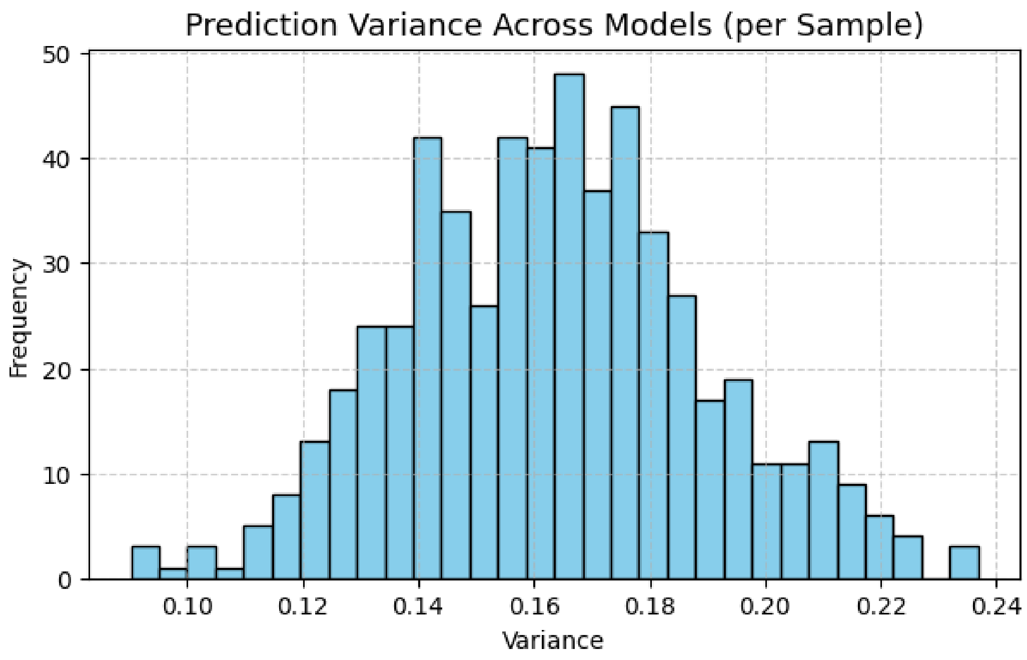
3. Comparing Functions within the Model Family

To better understand how different functions in the Logistic Regression family behave, I selected one random function and compared its predictions to the other 99. Each function produces a vector of predicted probabilities for the same dataset, and the differences between these vectors can be measured using a simple metric such as the mean squared difference. Most functions showed large differences from one another, indicating that their predictions vary widely depending on the randomly chosen parameters. A few functions, however, produced similar outputs, suggesting that certain random parameter values lead to comparable decision boundaries. This comparison highlights how sensitive Logistic Regression is to its parameters and reinforces why proper training is required to find parameter values that produce consistent and accurate predictions.

Journal Week 6 | Model Family – Part 2
Focus: Visualise 100 random functions
Expected Outcome: Model Family Outcome

To analyse the collective behaviour of the Logistic Regression model family, I asked ChatGPT to generate 100 random functions (each defined by unique w and b) and computed their predicted probabilities for all 569 samples in the Breast Cancer dataset. This formed a 100 × 569 prediction matrix, where each row represented one model's predictions across all samples.

The results showed an average prediction variance of 0.163 and an average pairwise correlation of 0.009, indicating that the models were highly diverse and made largely uncorrelated predictions. The histogram below illustrates this variation, showing that prediction behaviours differ significantly across the randomly sampled models.



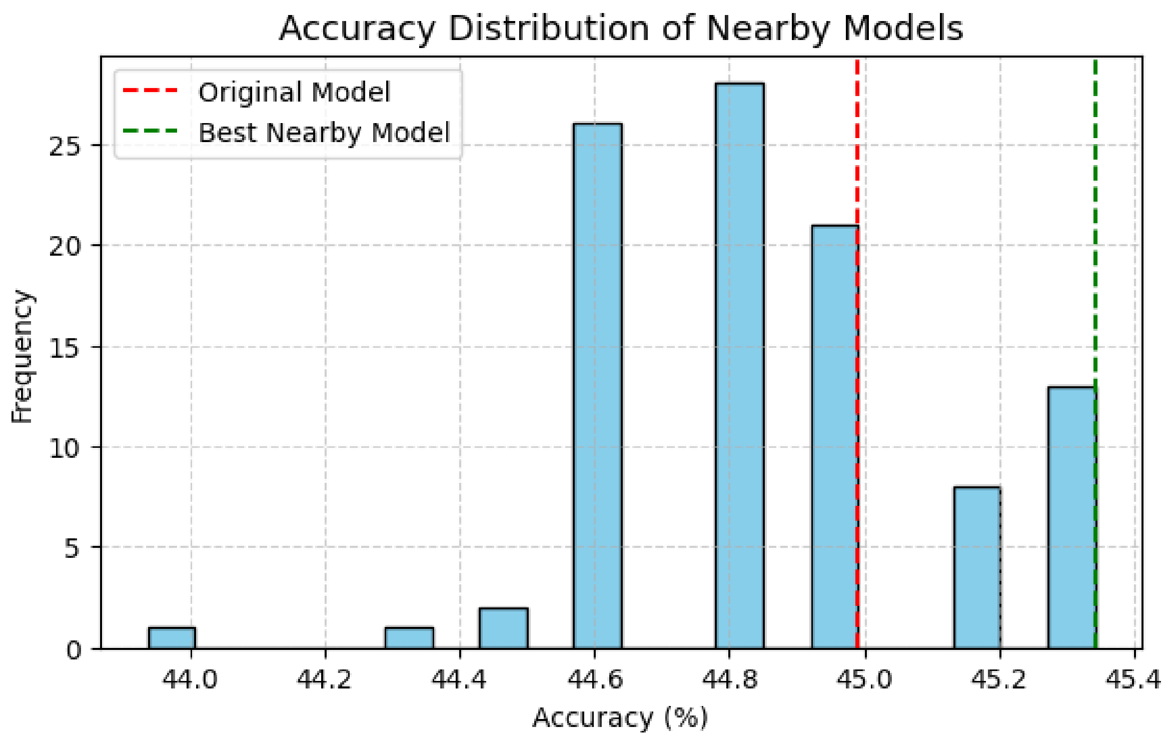
Expanding the model family (for example, by increasing the range of random weights and biases) would further increase prediction diversity, as models would produce a wider spread of outputs. Conversely, reducing the family (by narrowing the parameter range) would lead to more similar predictions and a higher concentration of behaviour. These observations highlight how parameter variability shapes the diversity and concentration of prediction patterns within a model family.

Journal Week 7 | Model Training
Focus: Model Training
Expected Outcome: Model Training Outcome

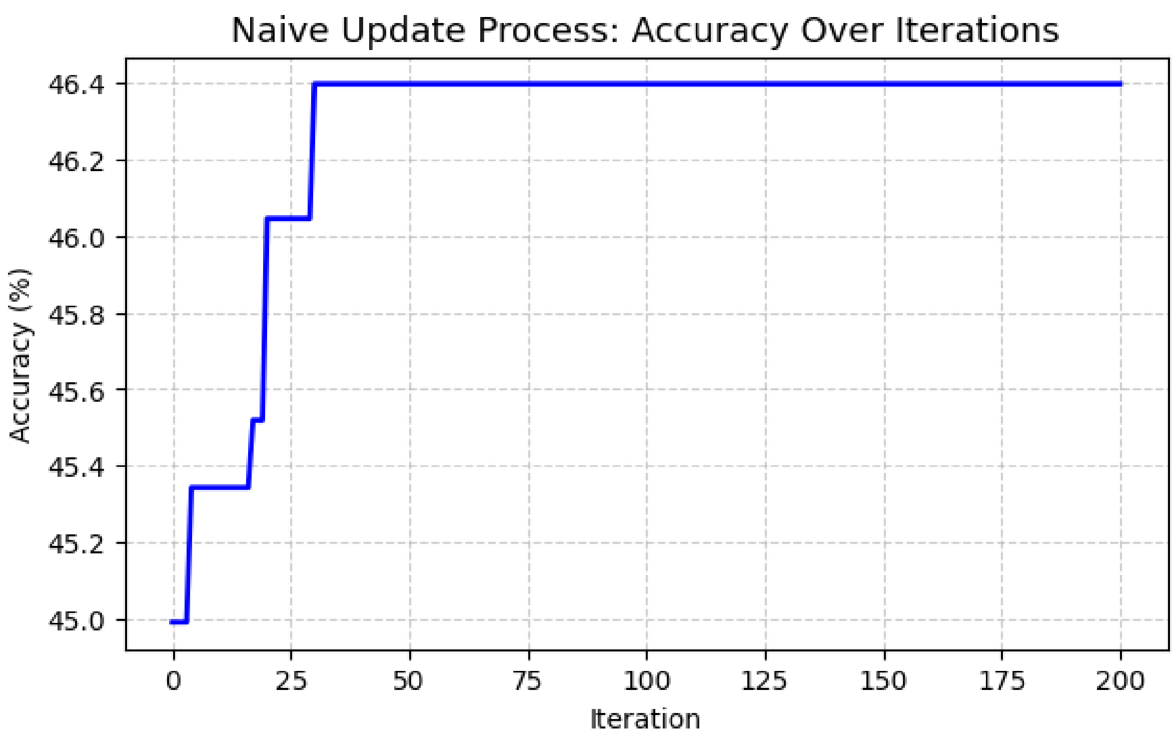
To begin exploring how training improves a model, I selected one random Logistic Regression function f defined by random weights and bias values. From this model, I generated 100 nearby functions by adding small Gaussian noise to the parameters ($N(0, 0.1)$). Each nearby model was evaluated on the Breast Cancer dataset, and their accuracies were compared to that of the original model.

The initial model achieved an accuracy of 44.99%, while the best nearby model reached 45.34%, an improvement of roughly 0.35%. Although small, this demonstrates that minor parameter changes can slightly improve performance.

The histogram below shows the distribution of accuracy among the nearby models, with most performing similarly to the original. This experiment illustrates what it means for models to be "nearby" in parameter space and shows that learning can be seen as a guided search for better parameter configurations.

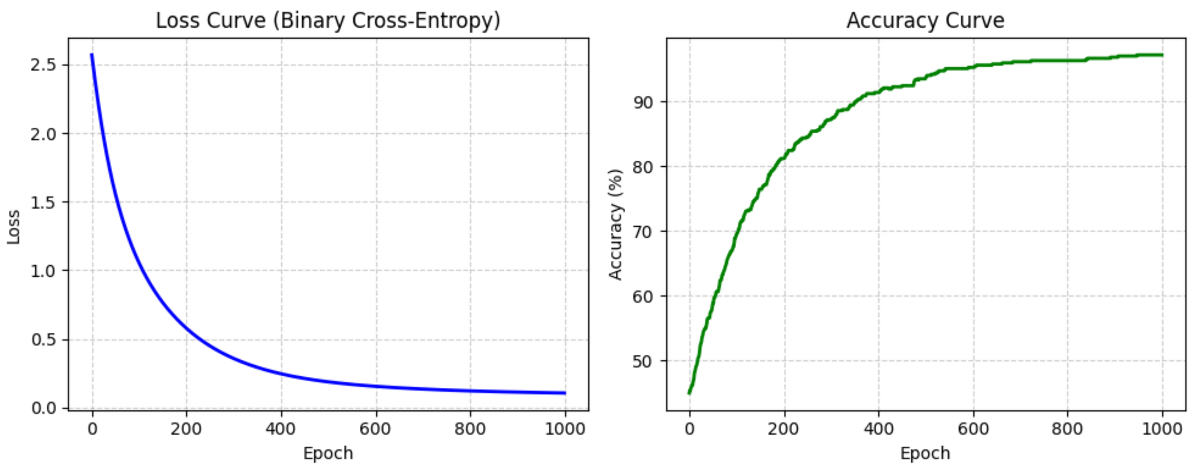


Naive Update Training
To simulate the idea of learning, I implemented a simple *Naive Update* process where the model's parameters were adjusted through small random changes. At each iteration, a slight perturbation was applied to the weights and bias, and the new model was kept only if it improved accuracy on the dataset. This process was repeated for 200 iterations, gradually increasing accuracy from 44.99% to 46.40%.
The plot below shows how accuracy improved in small steps before stabilizing, illustrating how local exploration in parameter space can lead to modest gains. However, this approach is extremely inefficient because it relies entirely on random guesses rather than any mathematical guidance. This experiment demonstrates the concept of *learning through iterative improvement* and serves as a foundation for understanding proper optimization techniques, such as gradient descent.



Model Training with Gradient Descent
After testing the Naive Update method, I replaced random parameter adjustments with a proper gradient descent optimizer. Using the Binary Cross-Entropy loss function, the model iteratively computed the gradients of the loss with respect to the weights and bias, updating them according to the rules: w = w - alpha * (dL/dw) and b = b - alpha * (dL/db).

Over 1000 epochs, the loss steadily decreased from 2.56 to 0.10, while accuracy increased from roughly 45% to 97%. The learning curves below clearly show this improvement—loss declined smoothly while accuracy rose sharply in the first few hundred epochs before stabilizing. This demonstrates how gradient descent efficiently guides the model toward optimal parameters, unlike the slower, random updates of the Naive approach.



Journal Week 8 | Construction and Reflection
Focus: Project Reflection
Expected Outcome: Possible Improvements and Project Reflection

If this project were to be repeated using a modern approach, a suitable alternative to Logistic Regression would be a Feedforward Neural Network (Multi-Layer Perceptron). A neural network can learn more complex, non-linear relationships between features and outcomes, whereas Logistic Regression is limited to modelling linear boundaries. Other popular modern methods such as Gradient Boosting Machines (e.g., XGBoost or LightGBM) could also achieve higher predictive performance by combining many weak learners into a strong ensemble.

However, Logistic Regression remains valuable for its simplicity, interpretability, and strong theoretical foundation. Implementing it from scratch provided clear insight into how model parameters, loss functions, and optimizers work together in training. Through this project, I gained practical understanding of Binary Cross-Entropy loss, gradient descent optimization, and the relationship between random parameter exploration and systematic learning.

Overall, this work demonstrated the complete construction of a machine-learning model, that is, from theoretical formulation to a working training pipeline, and deepened my understanding for how modern frameworks build upon these same mathematical principles.

Journal Week 9 | Project Promotion
Focus: Develop ways of promoting the project through various sources
Expected Outcome: GitHub Repo, project website, and project summary

GitHub Repository: https://github.com/knochez/Study-of-Logistic-Regression-Model/tree/main

Project Website: https://knochez.github.io/Study-of-Logistic-Regression-Model/index.html

Cloud Deployment Awareness
If this project were to be shared interactively, a simple deployment could be done using platforms like Streamlit, Gradio, or Hugging Face Spaces, allowing users to upload input data and view model predictions directly in their browsers. Alternatively, hosting the notebook on Google Collab would provide an accessible, reproducible environment for anyone to explore the training process. While deployment isn't necessary for this theoretical project, understanding these options demonstrates how even a simple model like Logistic Regression can be made accessible through modern cloud tools.

Project Summary and Research Roadmap
This project successfully explored the mathematical foundations and implementation of Logistic Regression, showing how a simple linear model learns through optimization. The process, from random parameter exploration to a fully functional gradient descent training pipeline, provided hands-on understanding of key machine learning principles such as hypothesis formulation, loss minimization, and model evaluation. Future work could expand on this foundation by studying regularization techniques to prevent overfitting, multiclass classification using the softmax function, or by progressing toward neural networks to understand how non-linear decision boundaries are learned. This project serves as a stepping stone for deeper study in optimization, model generalization, and modern deep learning methods.