**Logistic Regression for Classification**

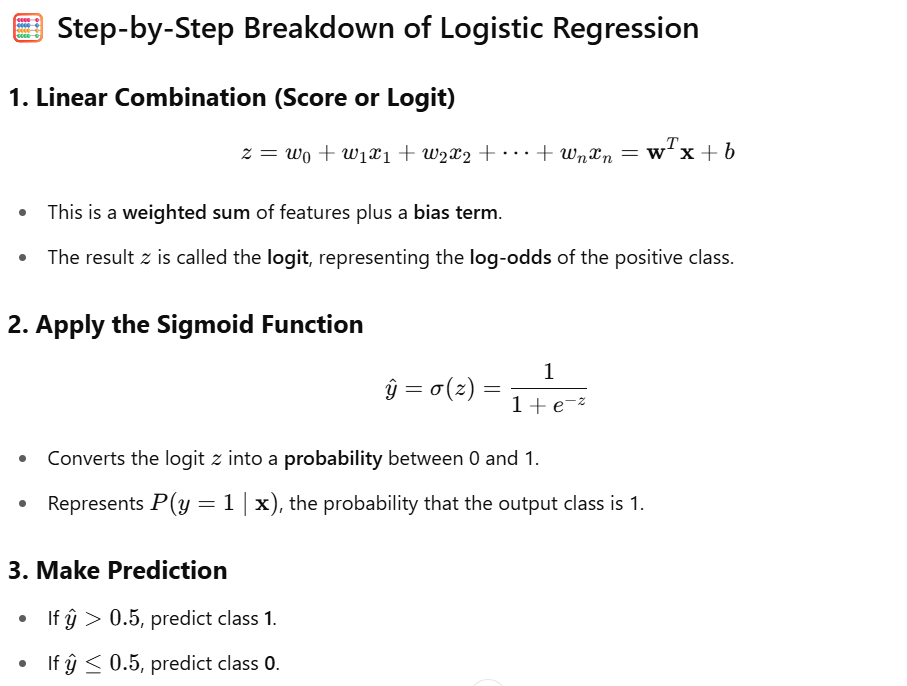
**What is Logistic Regression?**

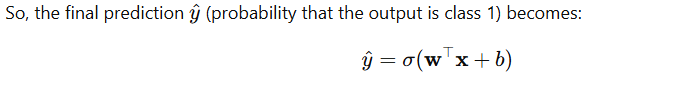
**Logistic Regression** is a **supervised machine learning algorithm** used for **classification tasks**. Unlike Linear Regression, which predicts continuous values, Logistic Regression is used to **predict the probability of a categorical outcome**, typically a **binary** or **multi-class** label.

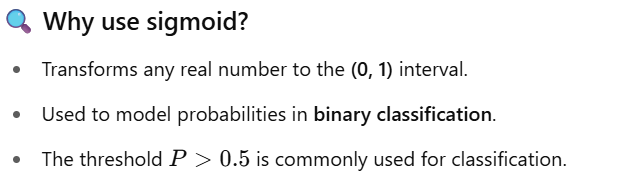
It models the relationship between one or more independent variables (**features**) and a categorical dependent variable (**class/label**) using the **logistic (sigmoid) function**, which outputs values between 0 and 1.

**📐 How It Works (Algorithm Intuition):**

1. **Linear Combination** of input features:

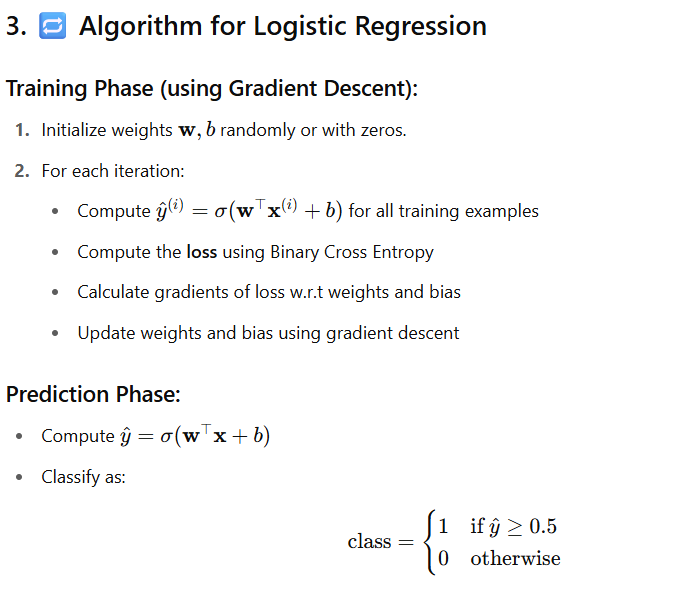
****

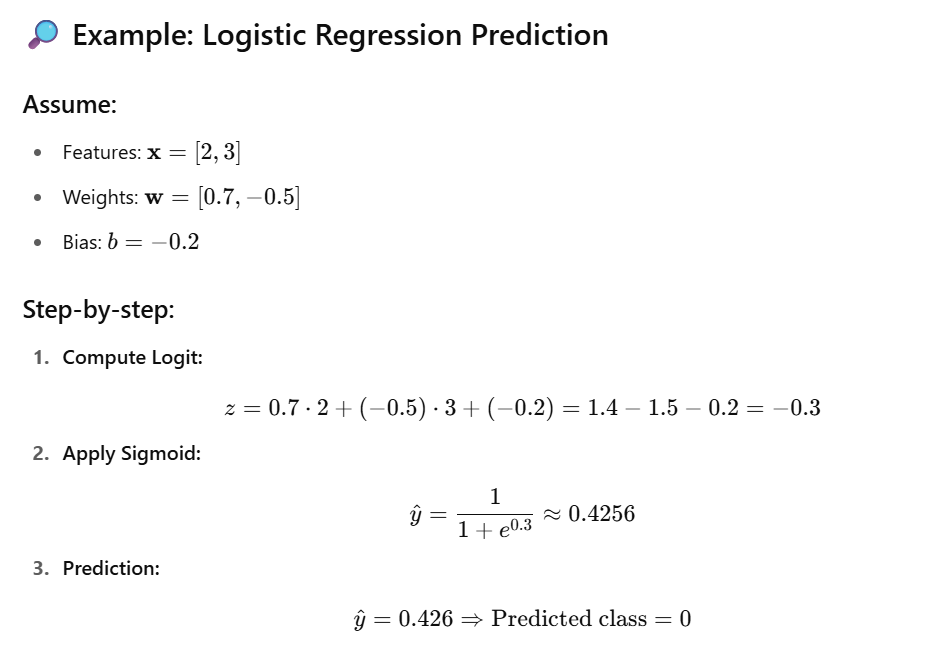
****

****

The output probability is then used to **classify** the sample:

* + Binary: If P>0.5P > 0.5P>0.5, classify as class 1; otherwise class 0.
  + Multiclass: Uses **softmax** for multiple categories.





**📌 Interpretation of Weights**

* **Positive weight** increases the probability of class 1 as the feature increases.
* **Negative weight** decreases the probability of class 1.
* Larger **magnitude** = more influence on the prediction.

import numpy as np

def sigmoid(z):

return 1 / (1 + np.exp(-z))

# Inputs

x = np.array([2, 3])

w = np.array([0.7, -0.5])

b = -0.2

# Step 1: Linear combination

z = np.dot(w, x) + b

# Step 2: Apply sigmoid

y\_prob = sigmoid(z)

# Step 3: Predict class

predicted\_class = 1 if y\_prob > 0.5 else 0

print(f"Logit (z): {z:.3f}")

print(f"Probability (y=1): {y\_prob:.3f}")

print(f"Predicted class: {predicted\_class}")

**📦 Use Cases of Logistic Regression:**

**✅ Binary Classification:**

* **Email spam detection** (Spam vs. Not Spam)
* **Customer churn prediction** (Will a customer leave? Yes/No)
* **Disease diagnosis** (Has diabetes? Yes/No)
* **Credit scoring** (Default on loan? Yes/No)

**✅ Multi-class Classification (with extensions):**

* **Digit recognition** (0–9)
* **Classifying types of wheat** (as in the Seeds Dataset)
* **Sentiment analysis** (Positive/Negative/Neutral)

**🧭 When to Apply Logistic Regression:**

You should use Logistic Regression when:

* ✅ Your target variable is **categorical** (binary or multi-class).
* ✅ The relationship between features and the log-odds of the outcome is approximately **linear**.
* ✅ You want a **fast and interpretable** baseline model.
* ✅ Your features are **numerical and scaled** (scaling helps convergence and accuracy).
* ✅ The dataset is **not too large** and doesn't require deep learning or complex modeling.

**⚠️ When NOT to Use:**

* ❌ When the outcome is **continuous** — use Linear Regression instead.
* ❌ When the data has complex **non-linear relationships** — consider decision trees, SVMs, or neural networks.
* ❌ When data has **high multicollinearity** — try regularization (L1/L2) or dimensionality reduction.

Would you like a visual explanation (e.g., sigmoid curve or decision boundary illustration)?

Let's walk through applying **Logistic Regression** to the **Seeds Dataset** from the UCI Machine Learning Repository. This dataset contains measurements of geometrical properties of kernels belonging to three different varieties of wheat: Kama, Rosa, and Canadian.

**📥 1. Import Libraries and Load Data**

python

CopyEdit

import pandas as pd

import numpy as np

from sklearn.linear\_model import LogisticRegression

from sklearn.preprocessing import StandardScaler

from sklearn.model\_selection import train\_test\_split

from sklearn.metrics import confusion\_matrix, accuracy\_score

# Load the dataset

url = 'https://archive.ics.uci.edu/ml/machine-learning-databases/00236/seeds\_dataset.txt'

column\_names = ['Area', 'Perimeter', 'Compactness', 'KernelLength', 'KernelWidth',

'AsymmetryCoeff', 'KernelGrooveLength', 'Class']

df = pd.read\_csv(url, sep='\s+', names=column\_names)

**⚙️ 2. Normalize Data (Feature Scaling)**

python

CopyEdit

# Features and target

X = df.drop('Class', axis=1)

y = df['Class']

# Normalize features

scaler = StandardScaler()

X\_scaled = scaler.fit\_transform(X)

**🔀 3. Split Data – Train/Test**

python

CopyEdit

# Split into training and testing sets

X\_train, X\_test, y\_train, y\_test = train\_test\_split(

X\_scaled, y, test\_size=0.2, random\_state=42, stratify=y)

**🤖 4. Apply Logistic Regression**

python

CopyEdit

# Initialize and train logistic regression model

lr = LogisticRegression(multi\_class='multinomial', solver='lbfgs', max\_iter=1000)

lr.fit(X\_train, y\_train)

# Print model coefficients

print("Model Coefficients:\n", lr.coef\_)

**📊 5. Predict and Evaluate**

python

CopyEdit

# Predict on test set

y\_pred = lr.predict(X\_test)

# Confusion matrix and accuracy

conf\_matrix = confusion\_matrix(y\_test, y\_pred)

accuracy = accuracy\_score(y\_test, y\_pred)

print("Confusion Matrix:\n", conf\_matrix)

print(f"Accuracy: {accuracy:.2f}")

**🧠 Algorithm Explanation**

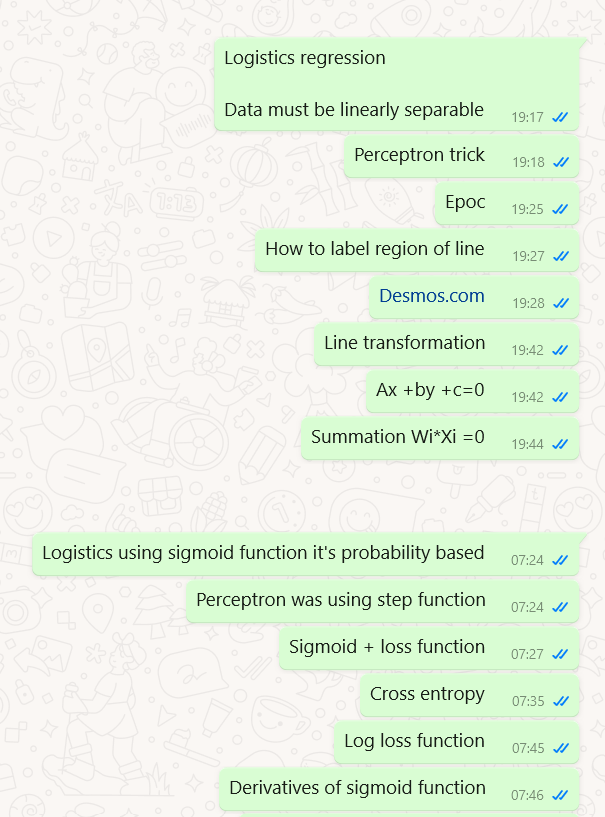
**Logistic Regression** is a classification algorithm that models the probability of a sample belonging to a particular class. In the case of the Seeds dataset, we're dealing with a multiclass classification problem (three classes). The algorithm estimates the parameters (coefficients) that best fit the training data by maximizing the likelihood of the observed data.

* **Coefficients (lr.coef\_)**: These represent the weight of each feature in predicting the classes. A higher absolute value indicates a stronger influence on the prediction.
* **Confusion Matrix**: This table allows us to visualize the performance of the algorithm by showing the actual vs. predicted classifications.
* **Accuracy**: This metric indicates the proportion of correct predictions made by the model.

By analyzing the confusion matrix and accuracy, we can assess how well the logistic regression model is performing on the Seeds dataset.

If you have any further questions or need assistance with other datasets, feel free to ask!

**Some important term that are used in logistic regression**



**1. Perceptron Trick**

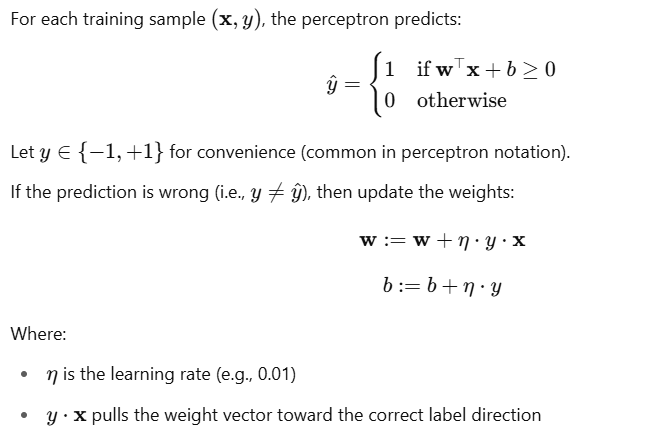
**🧠 What is it?**

The **Perceptron Trick** is the **weight update rule** used in the **Perceptron algorithm**, a very early binary classifier.

It’s used to:

* Push incorrectly classified data points toward the correct side of the decision boundary.
* Adjust the decision boundary incrementally in response to mistakes.

**🧮 Mathematical Formulation**



**🔍 What’s Happening Intuitively?**

If the model misclassifies a positive example:

* Increase the weights in the direction of that input → makes the model more likely to classify similar points as positive in the future.

If the model misclassifies a negative example:

* Decrease the weights in the direction of that input → moves the decision boundary away.

**📌 This trick is the basis for:**

* Online learning (real-time updates)
* Stochastic Gradient Descent in logistic regression

**🔹 2. Epoch (EPOC)**

(Epoch is often miswritten as **EPOC**; correct term: **Epoch**.)

**🧠 What is an Epoch?**

An **epoch** refers to **one complete pass through the entire training dataset**.

* If you have 1000 training samples and you pass them all through the model once, that's **1 epoch**.
* Typically, models train for **many epochs** (e.g., 100 or more), adjusting weights repeatedly.

**🌀 Difference Between Epoch, Batch, and Iteration:**

| **Term** | **Description** |
| --- | --- |
| **Epoch** | One full pass over **all** training samples |
| **Iteration** | One update of weights based on **one batch** |
| **Batch Size** | Number of samples processed before updating weights |

🔁 For example:

* Dataset = 1000 samples
* Batch size = 100
* 1 epoch = 10 iterations

**🔧 How Epochs Affect Learning:**

* Too **few epochs** → underfitting (model hasn't learned enough)
* Too **many epochs** → overfitting (model memorizes training data)

**🔹 3. Line Transformation (Feature Mapping)**

**🧠 Why is it needed?**

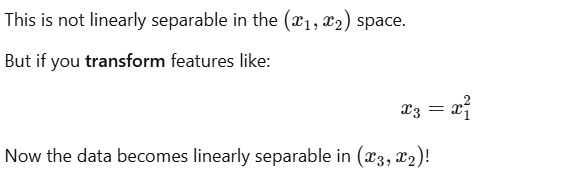
Logistic Regression and Perceptron draw **linear** boundaries — but what if data is **non-linearly separable**?

📌 **Line Transformation** (also called **feature transformation** or **feature mapping**) is a trick to **map inputs to a higher-dimensional space** where the data becomes linearly separable.

**🧮 Example**

Say we have a dataset like this:

| **x1x\_1x1​** | **x2x\_2x2​** | **Class** |
| --- | --- | --- |
| 1 | 1 | 0 |
| 2 | 4 | 0 |
| 3 | 9 | 0 |
| -1 | 1 | 1 |
| -2 | 4 | 1 |
| -3 | 9 | 1 |



**✨ This is the core idea behind:**

* **Polynomial Features** in logistic regression
* **Kernel Trick** in SVMs
* **Neural Networks** (they automatically learn these transformations!)

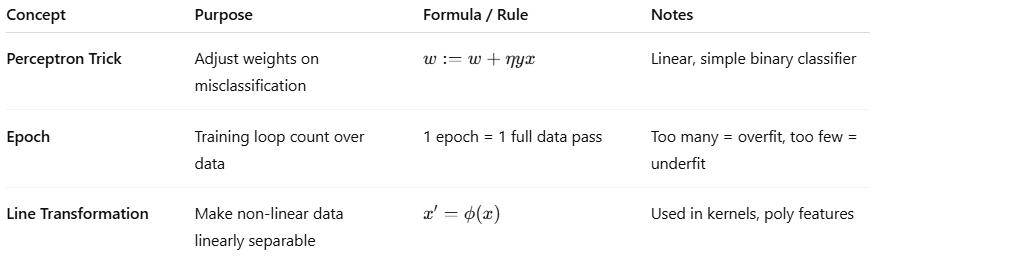
**📌 Visual Intuition:**

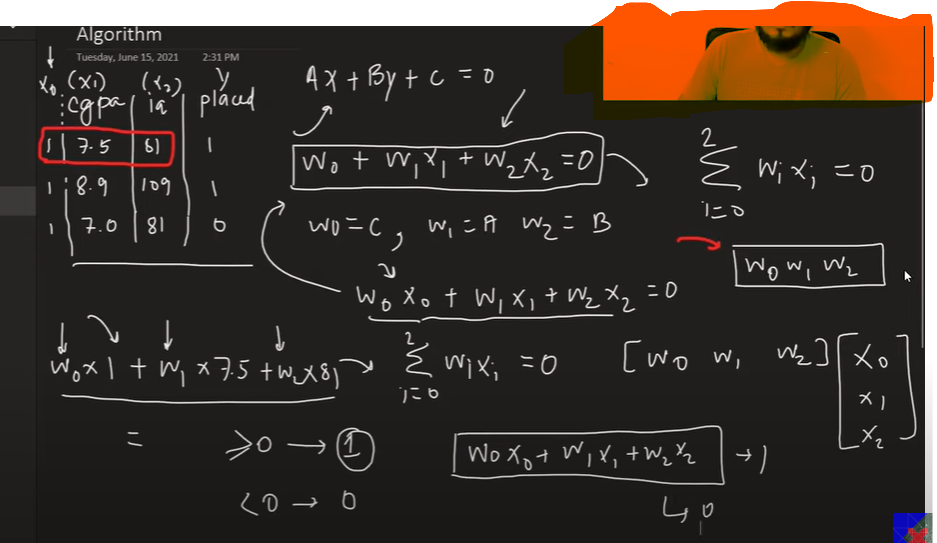
Imagine:

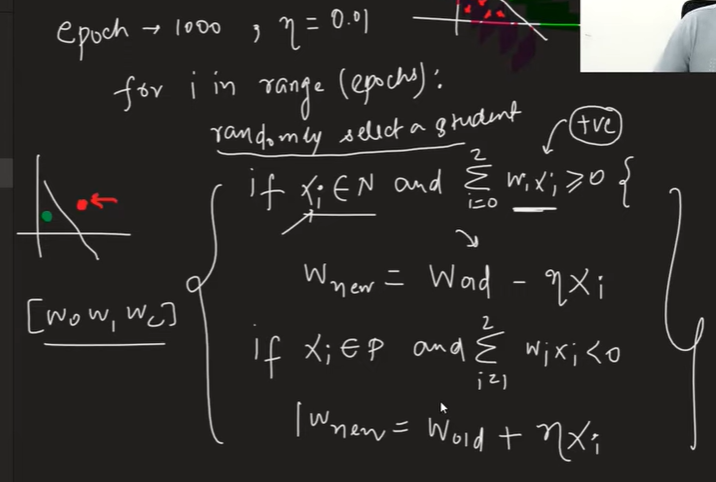
* In 2D: You can't separate data with a line.
* Transform it into 3D → now you can use a plane to separate it.

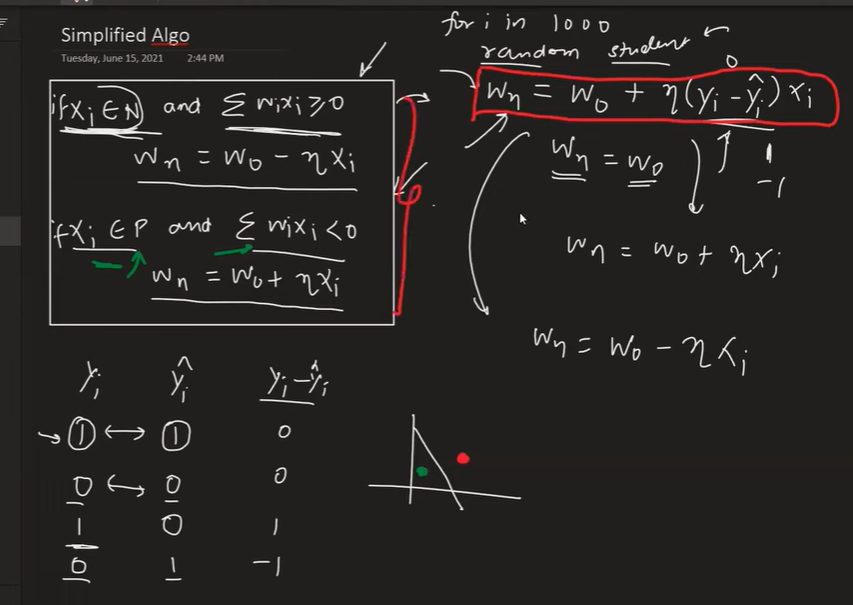
Once separated, you can **project the decision back into 2D**, and it becomes a **non-linear curve**.

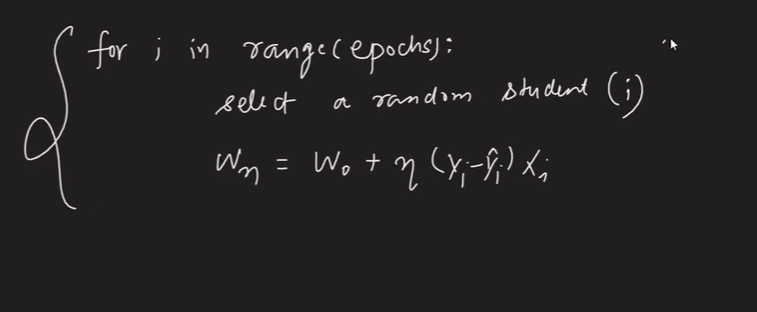
**✅ Summary Table**



****

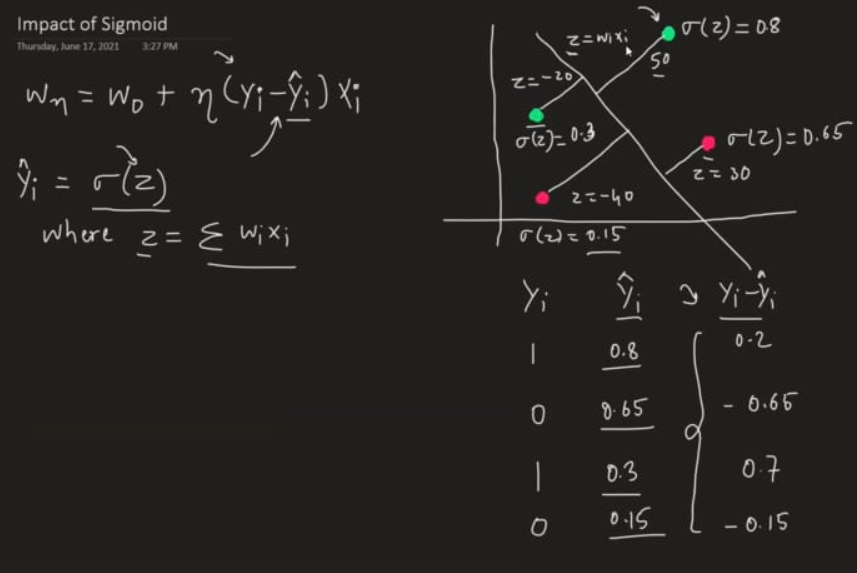
****

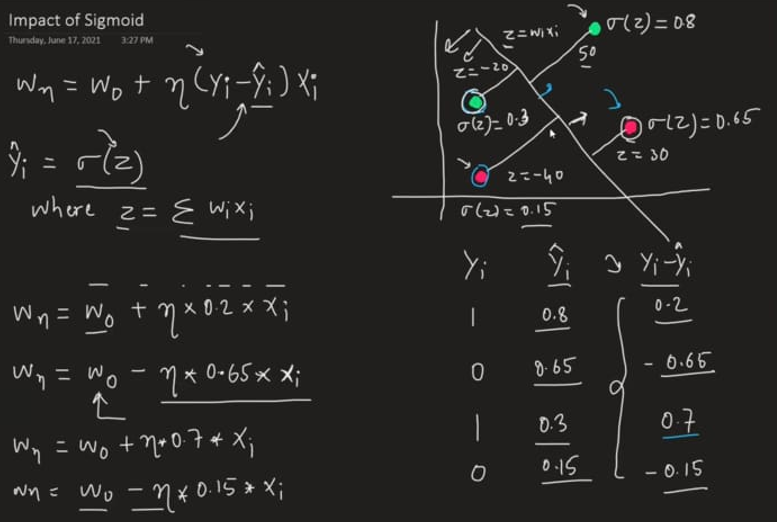
****

****

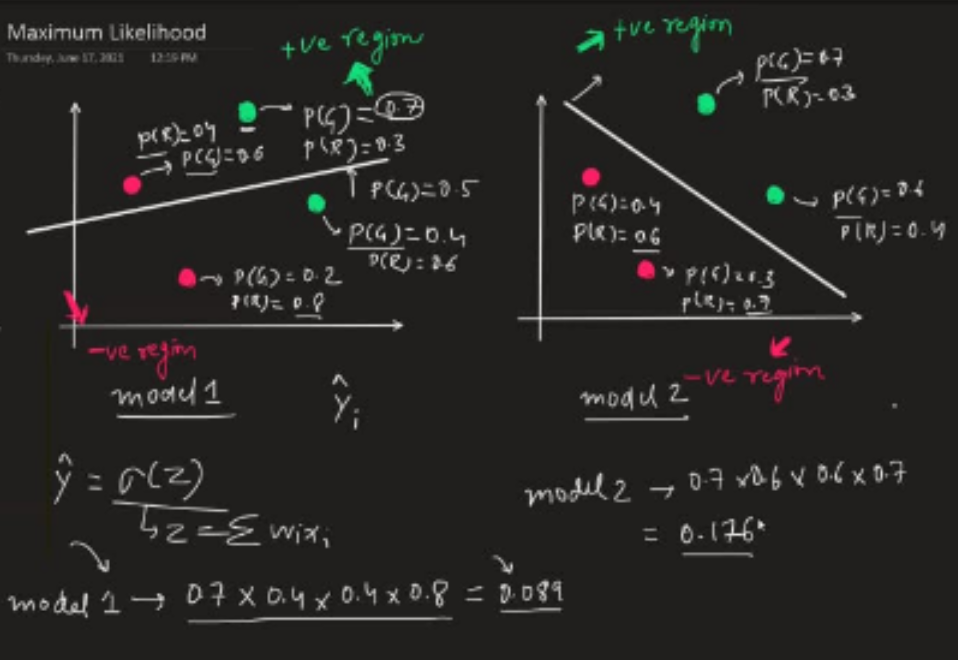
**This approach have some issues**

**Use sigmoid instead of step function**

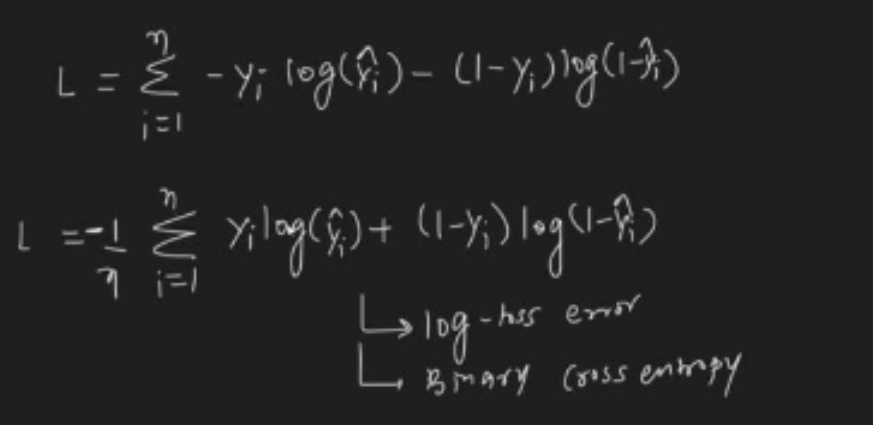
****

****

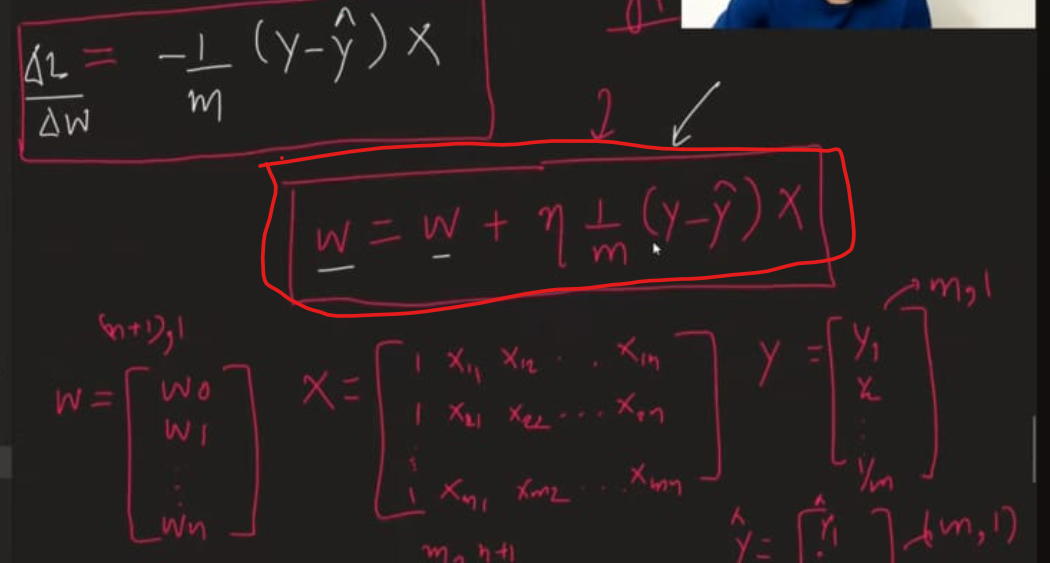
**Using sigmoid still we are not getting perfect predicted line which can divide both blue and green point from the middle so we need a loss function to calculate minima which can tell this line has the minimum loss**

****

**Log loss function**

****

**Derivative of log loss function (using gradient decent)**

****

**We can use this formula to write python code which will work like sklearn**

**Code -** [**https://github.com/campusx-official/100-days-of-machine-learning/tree/main/day58-logistic-regression**](https://github.com/campusx-official/100-days-of-machine-learning/tree/main/day58-logistic-regression)

**Intuition Behind Logistic Regression: Deep Dive**

**1. Why Logistic Regression?**

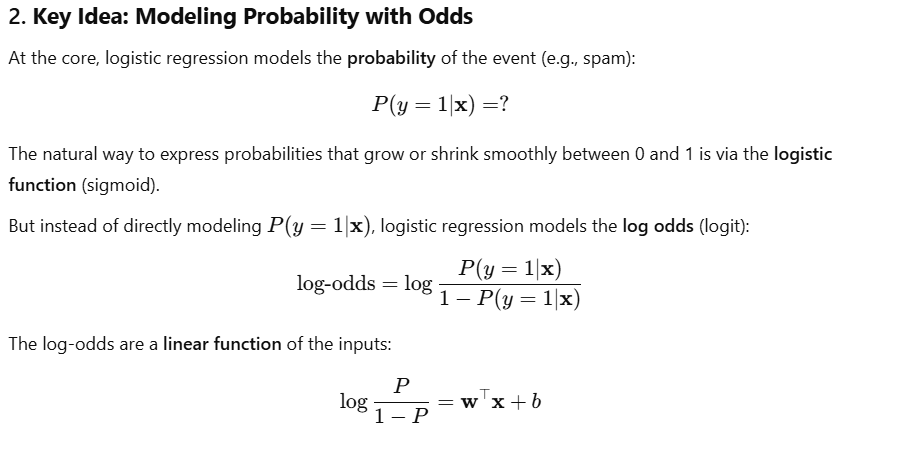
Imagine you have a problem where you want to decide whether an email is spam or not spam.

* You have features like number of links, sender domain, presence of certain keywords.
* The output is **binary**: spam (1) or not spam (0).

A linear model (like linear regression) might predict some real-valued number, say 0.7 or -0.3, but that’s not a probability and doesn’t fit binary classification directly.

You want:

* **A model that outputs probabilities** (between 0 and 1).
* These probabilities can then be turned into classes by choosing a threshold (usually 0.5).



**3. Why Log-Odds?**

* Odds represent how likely the event is relative to it not happening.
* Taking the **logarithm** makes it a linear combination of features.
* This linearity lets you use all the nice tools of linear models while keeping output interpretable probabilistically.

**4. Sigmoid as a “Squashing” Function**

This is perfect for probabilities.

* When input zzz is very large positive, sigmoid output →1\to 1→1.
* When input zzz is very large negative, sigmoid output →0\to 0→0.
* Around zero, sigmoid smoothly transitions from 0 to 1.

**5. Decision Boundary**

The model draws a **boundary** in feature space where the predicted probability is exactly 0.5:



* This boundary separates the classes.
* Points on one side are predicted as class 1, other side class 0.

**6. Learning the Parameters**

How does the model learn weights w and bias b?

* By **maximizing the likelihood** (or equivalently minimizing the cross-entropy loss).
* This means the model finds parameters that make the observed data most probable.

This is a form of **probabilistic modeling** — logistic regression assumes the data is generated from a Bernoulli distribution whose parameter depends on input features.

**7. Connection to Linear Regression**

* Linear regression predicts continuous values directly.
* Logistic regression predicts probabilities via a **non-linear transformation** of a linear combination of features.
* The key twist is the **sigmoid function**, which lets us interpret outputs probabilistically and classify binary outcomes.

**8. Why Not Just Use Linear Regression for Classification?**

Because:

* Linear regression outputs can be less than 0 or greater than 1 — invalid probabilities.
* The error surface (loss function) is different; logistic regression uses cross-entropy, which better fits classification problems.
* Logistic regression probabilities **saturate** near 0 or 1, making it confident about classification.

**9. Visual Intuition**

* Imagine a seesaw balanced at zero.
* Logistic regression tilts the seesaw by the weighted 
* The sigmoid maps this tilt to a smooth curve — the steeper the tilt, the closer the probability to 0 or 1.

**10. Summary: Intuition Checklist**

| **Aspect** | **Intuition** |
| --- | --- |
| Goal | Model probability of class membership |
| Core transformation | Linear combination + sigmoid (logistic) |
| Output interpretation | Probability between 0 and 1 |
| Decision making | Threshold probability at 0.5 |
| Parameter learning | Maximize likelihood / minimize cross-entropy |
| Decision boundary | Linear boundary in feature space |

**How Logistic Regression Handles Different Data Types**

**1. Linear Data**

* **Behavior**: Logistic Regression performs **very well** on linearly separable data.
* **Explanation**: Since it models a **linear decision boundary**, it can easily find a hyperplane (line in 2D, plane in 3D) that separates the two classes.

**Example:**

If the data looks like this in 2D:

Class 0 Class 1

o x

o x

o x

Logistic Regression can learn a line like:



to separate them.

**2. Non-linear Data**

* **Behavior**: Logistic Regression **struggles** on non-linearly separable data.
* **Solution**: You need to **transform** your features (via feature engineering or using kernels) to **make them linearly separable in a higher dimension**.

**Example:**

XOR pattern (non-linearly separable):

vbnet

CopyEdit

Class 0: (0,0), (1,1)

Class 1: (0,1), (1,0)

A straight line cannot separate this data. But if you create a new feature like:

****

Now the data becomes linearly separable in the new space.

**3. Polynomial Data**

* **Behavior**: Base logistic regression can’t handle this **unless you add polynomial features**.
* **Solution**: Add higher-degree polynomial terms:



Then Logistic Regression learns a **non-linear decision boundary** in the original space, but it's still **linear in the transformed feature space**.

**4. High-dimensional or Sparse Data (e.g., Text Classification)**

* Logistic Regression works well here too, especially with **regularization (L1/L2)**.
* It’s often used for:
  + **Spam detection**
  + **Sentiment analysis**
  + **Document classification**

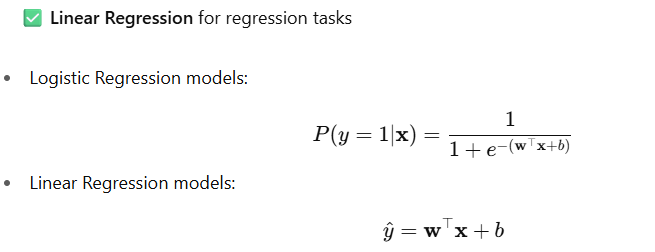
**🎯 Can Logistic Regression Solve Regression Problems?**

**❌ Short Answer: No, not directly.**

**✅ Explanation:**

* Logistic Regression is specifically designed to predict **probabilities** of class membership, not continuous numeric outputs.
* If your target variable is continuous (e.g., house price, age), you should use:

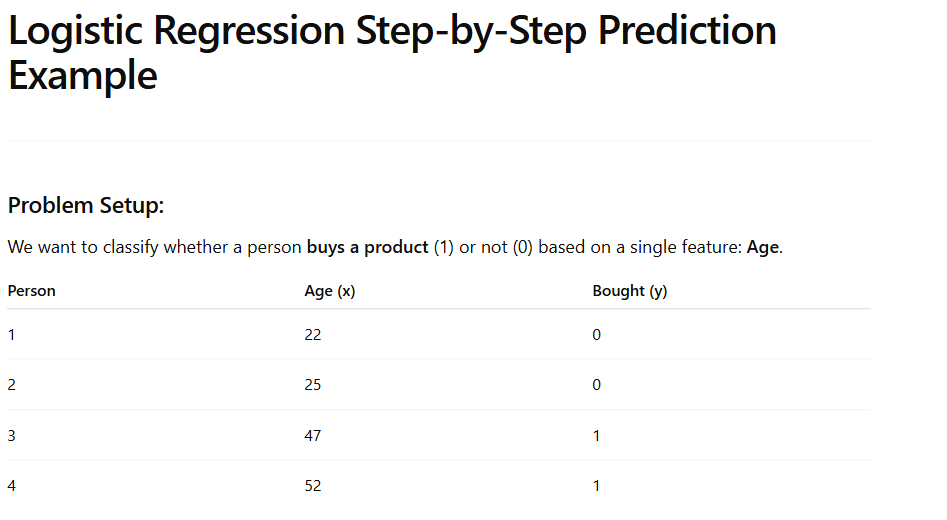
✅ **Linear Regression** for regression tasks

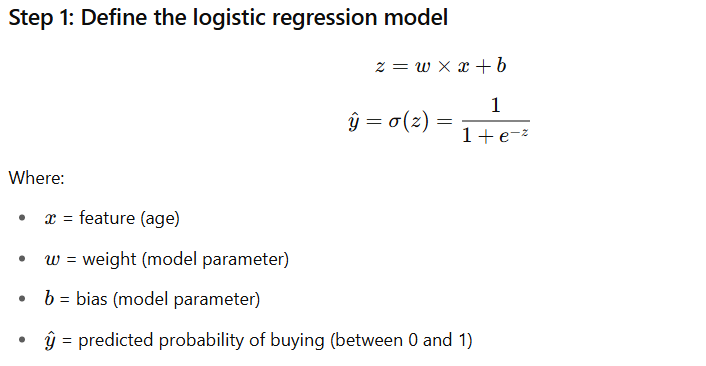


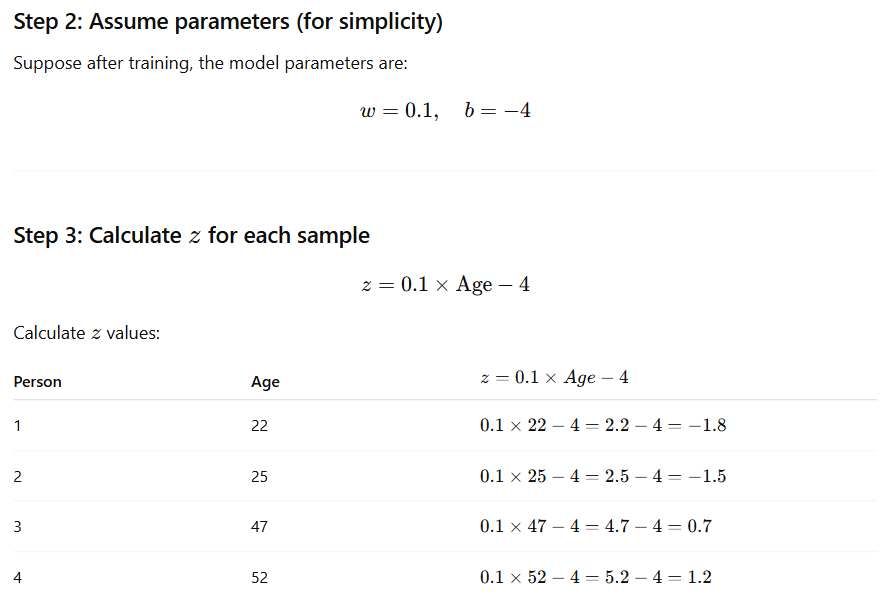
So although they look structurally similar, they solve **very different problems**.

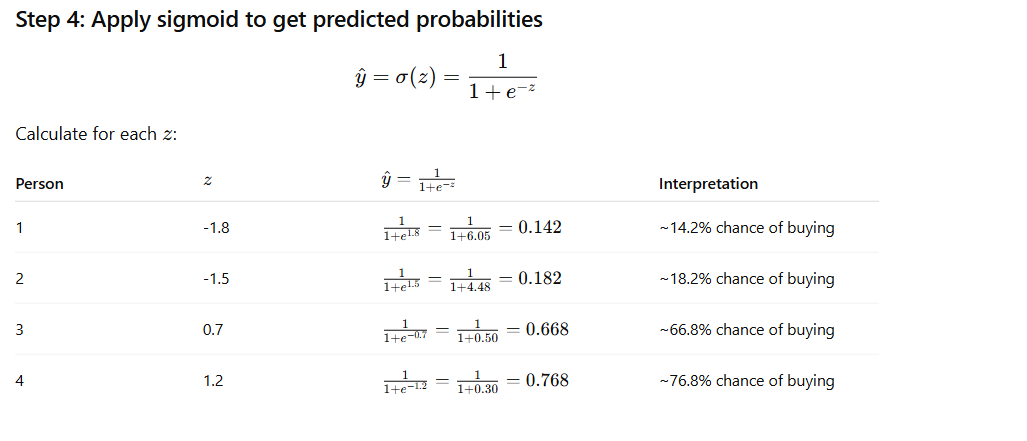
**🛠 Summary Table**

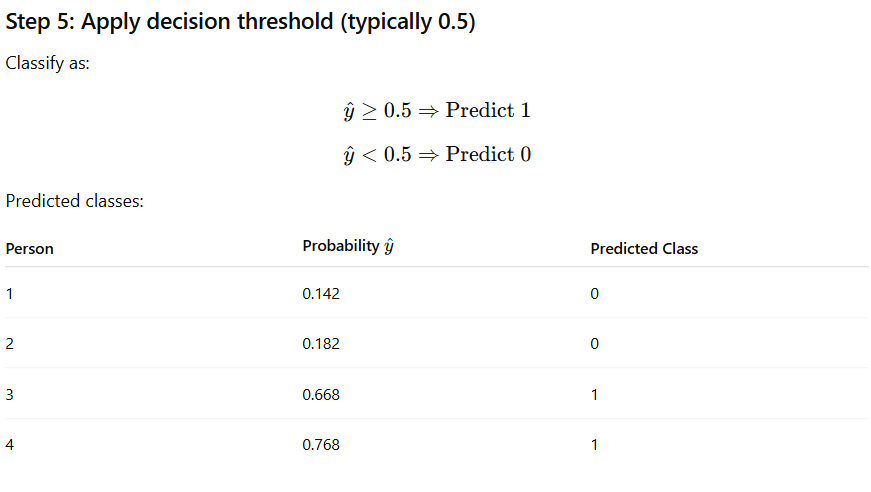
| **Data Type** | **Logistic Regression Performance** | **Requires Feature Transformation?** |
| --- | --- | --- |
| Linear | ✅ Excellent | ❌ No |
| Non-linear | ❌ Poor | ✅ Yes (e.g., kernel, feature map) |
| Polynomial | ❌ Initially Poor | ✅ Yes (add polynomial terms) |
| Sparse/Text | ✅ Excellent | ❌ Often No |
| Regression Target | ❌ Not Suitable | ❌ Use Linear Regression Instead |

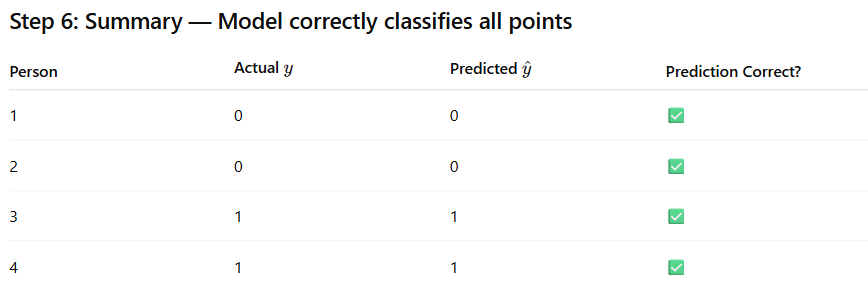


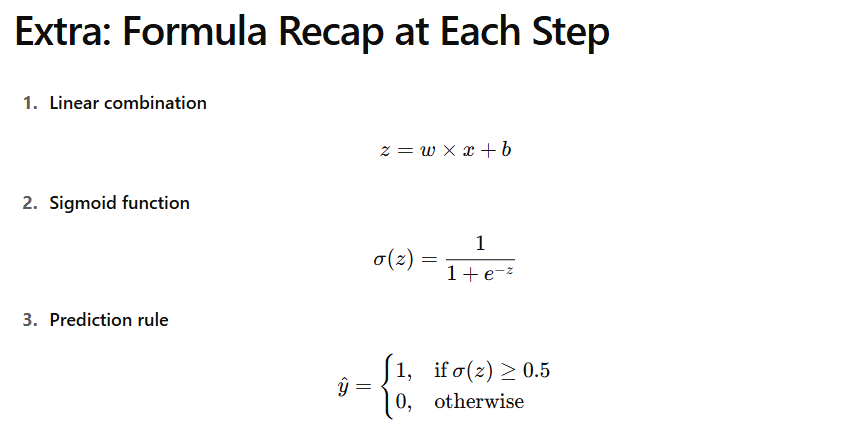


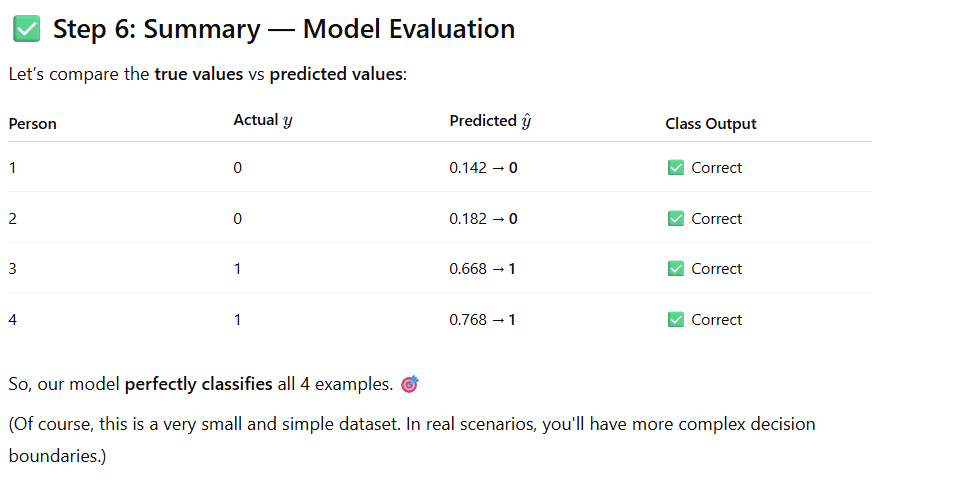


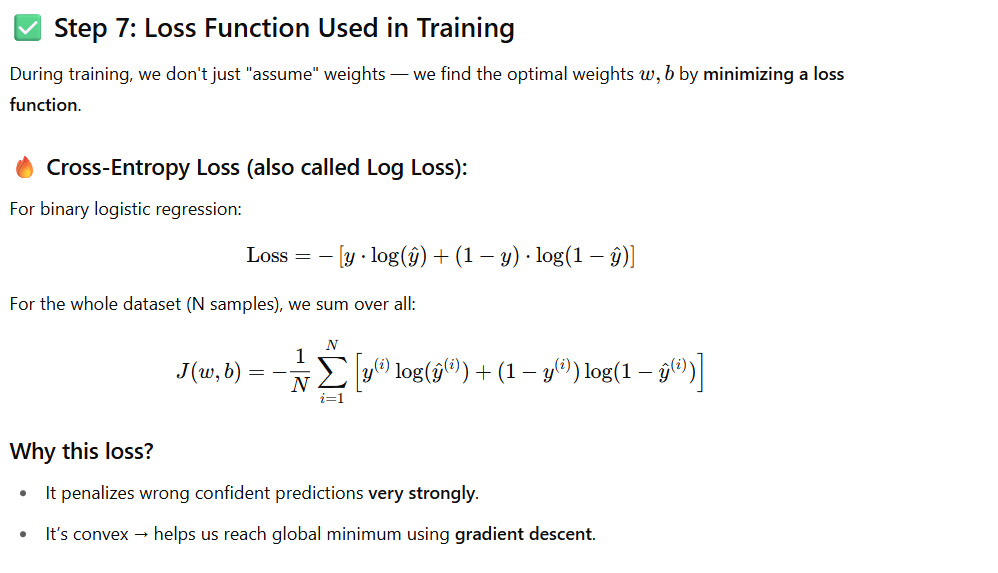


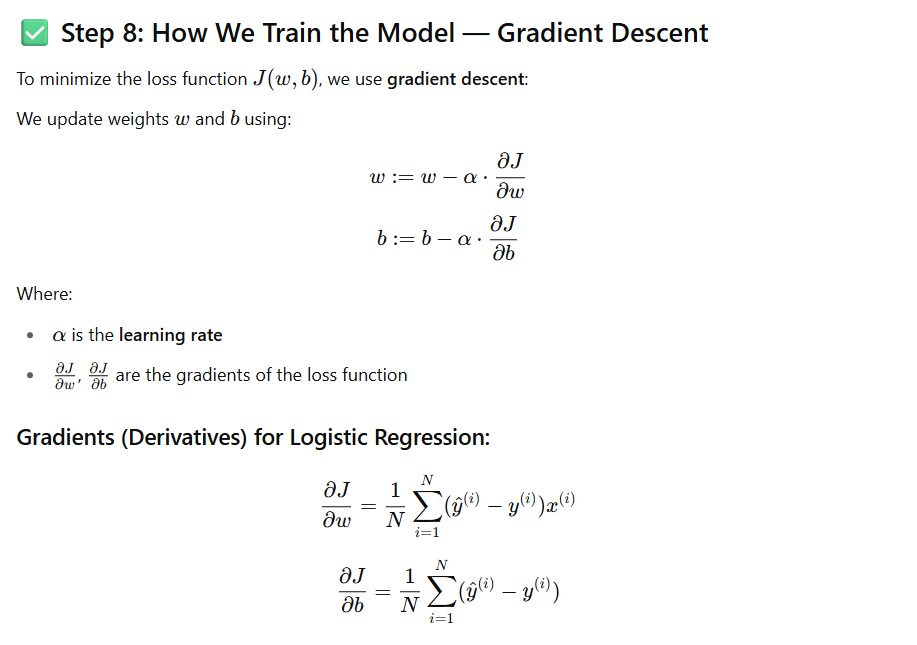


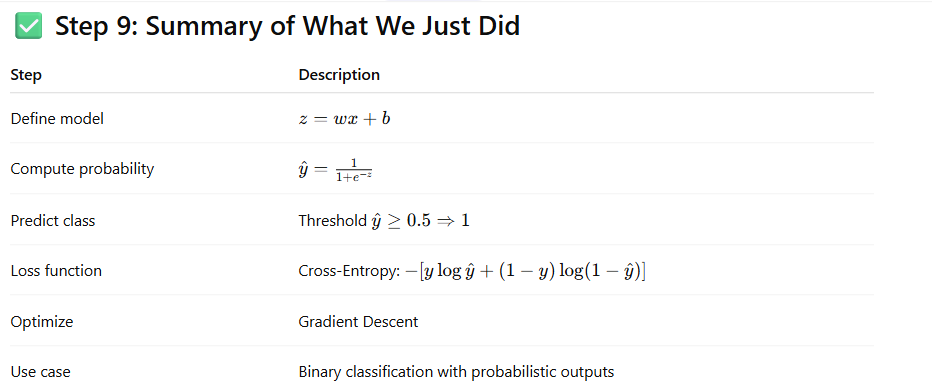












**Bonus Intuition**

**Imagine:**

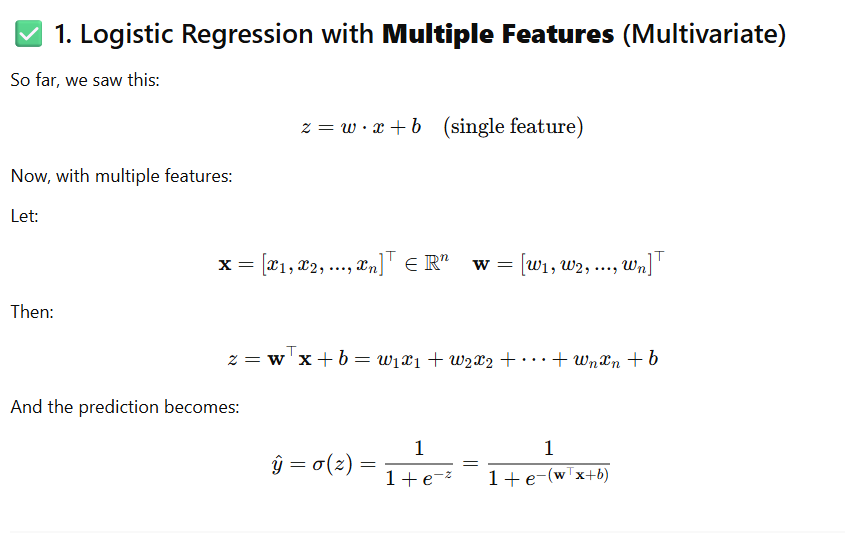
* Your sigmoid output is 0.95, but true label is 0 → model is **overconfident** and wrong → **high penalty** in loss.
* Your sigmoid output is 0.6 and true label is 1 → **correct**, but not confident → **moderate loss**.

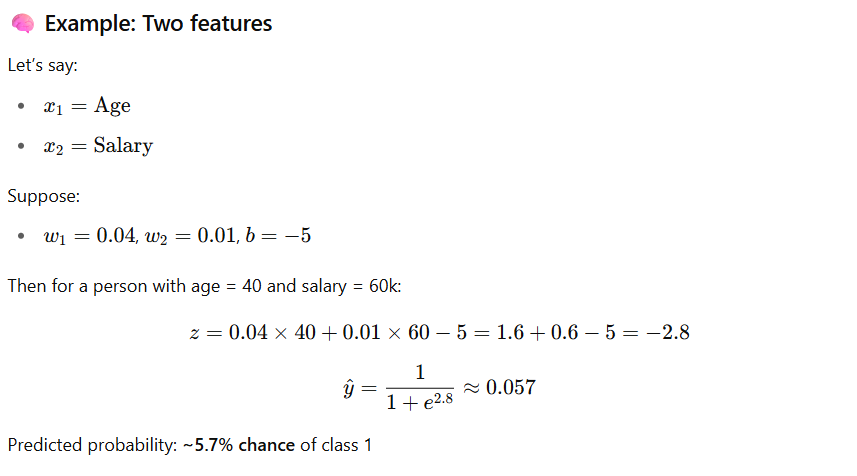
This encourages the model to **improve both accuracy and confidence** during training.

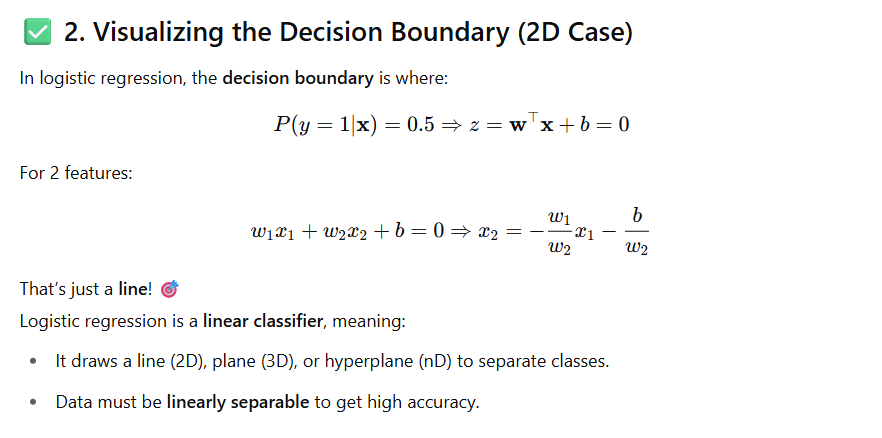
**🎯 Final Thoughts**

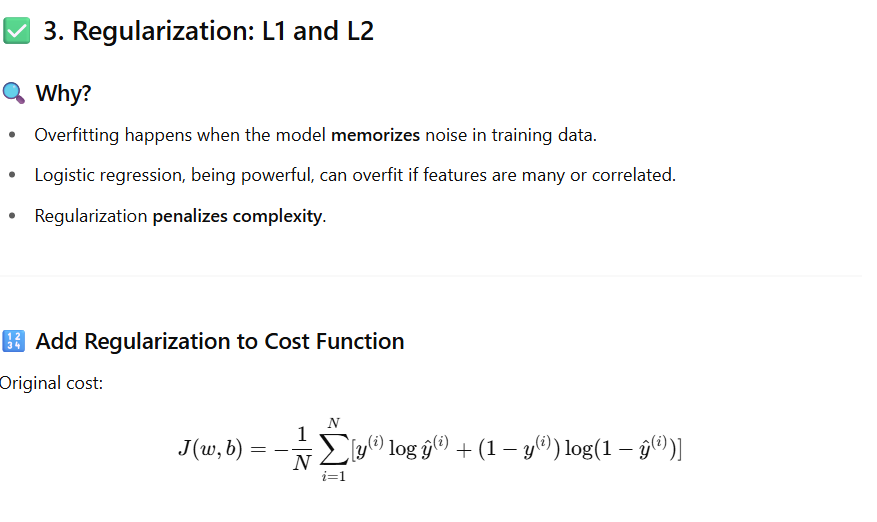
Logistic regression is powerful **not because it models non-linearity**, but because:

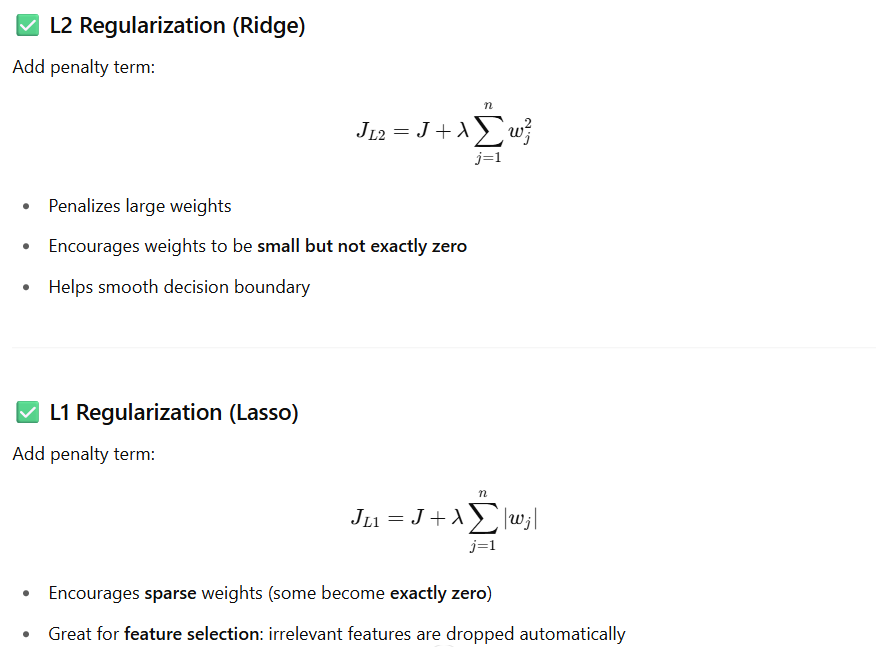
* It’s **interpretable**
* Works well for **linearly separable** classes
* Outputs **probabilities** you can calibrate
* Scales well to large datasets
* Forms the base for many advanced models (e.g., neural networks)











**1. Common Logistic Regression Hyperparameters (Overview)**

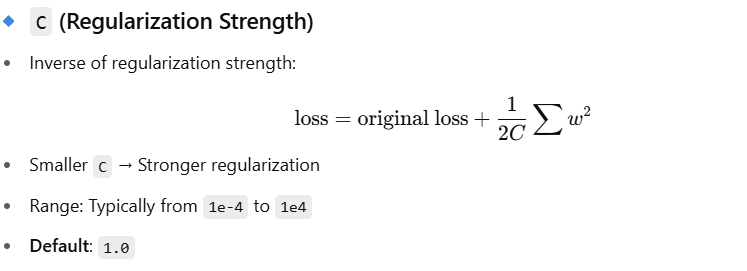
| **Hyperparameter** | **Description** |
| --- | --- |
| penalty | Type of regularization ('l1', 'l2', 'elasticnet', or 'none') |
| C | Inverse of regularization strength (smaller → more regularization) |
| solver | Optimization algorithm ('liblinear', 'saga', 'lbfgs', etc.) |
| max\_iter | Maximum number of iterations for convergence |
| fit\_intercept | Whether to include a bias term (intercept) |
| class\_weight | Handles class imbalance by assigning weights to classes |
| tol | Tolerance for stopping criteria in optimization |
| multi\_class | Strategy for multi-class classification ('ovr', 'multinomial') |
| l1\_ratio | Mixing ratio for L1 vs L2 (used only when penalty='elasticnet') |
| random\_state | Seed for randomness (important for reproducibility) |

**2. Explanation with Scikit-learn’s LogisticRegression (sklearn.linear\_model.LogisticRegression)**

**🔹 penalty**

* **Options**: 'l1', 'l2', 'elasticnet', 'none'
* **Default**: 'l2'
* Use:
  + 'l1': Feature selection (sparse models)
  + 'l2': Ridge regularization
  + 'elasticnet': Combination of L1 + L2
  + 'none': No regularization

🧠 Note: Not all solvers support all penalty types!



**solver**

| **Solver** | **Supports Penalties** | **Suitable For** |
| --- | --- | --- |
| 'liblinear' | 'l1', 'l2' | Small datasets, binary, OVR |
| 'lbfgs' | 'l2' | Large datasets, multiclass |
| 'saga' | 'l1', 'l2', 'elasticnet', 'none' | Large-scale, sparse |
| 'newton-cg' | 'l2', 'none' | Multiclass (better than lbfgs in some cases) |

**🔹 multi\_class**

* 'ovr' = One-vs-Rest (default for solvers except 'lbfgs')
* 'multinomial' = Softmax-based multiclass logistic regression
* 'auto' = Automatically chooses based on solver

**🔹 max\_iter**

* Max number of optimization iterations (default: 100)
* Increase this if solver doesn’t converge

**🔹 fit\_intercept**

* True: Learn bias term bbb
* False: Don’t learn intercept

**🔹 class\_weight**

* 'balanced' or custom dictionary like {0: 1, 1: 3}
* Useful for **imbalanced datasets**

**🔹 l1\_ratio (Elastic Net only)**

* Range: 0 (L2) to 1 (L1)
* Only used when penalty='elasticnet' and solver is 'saga'

**✅ 3. Suggested Configurations**

**🔸 Binary classification with regularization:**

python

CopyEdit

LogisticRegression(penalty='l2', C=1.0, solver='lbfgs')

**🔸 Sparse feature selection with L1:**

python

CopyEdit

LogisticRegression(penalty='l1', solver='liblinear', C=0.1)

**🔸 Large-scale, elastic net:**

python

CopyEdit

LogisticRegression(penalty='elasticnet', solver='saga', l1\_ratio=0.5)

**🔸 Imbalanced classes:**

python

CopyEdit

LogisticRegression(class\_weight='balanced')

**✅ 4. Tuning Tips**

* Always **scale your features** (StandardScaler or MinMaxScaler)
* Use **cross-validation** to tune C, penalty, and solver
* Monitor **convergence warnings** — increase max\_iter if needed
* Use **grid search or randomized search** for hyperparameter tuning

**✅ 5. Visualization of Effect of C**

| **C Value** | **Effect on Model** | **Interpretation** |
| --- | --- | --- |
| 0.001 | Very regularized | Simpler, low variance, high bias |
| 1.0 | Medium regularized | Balanced |
| 1000 | Weak regularization | Model fits training data tightly |