

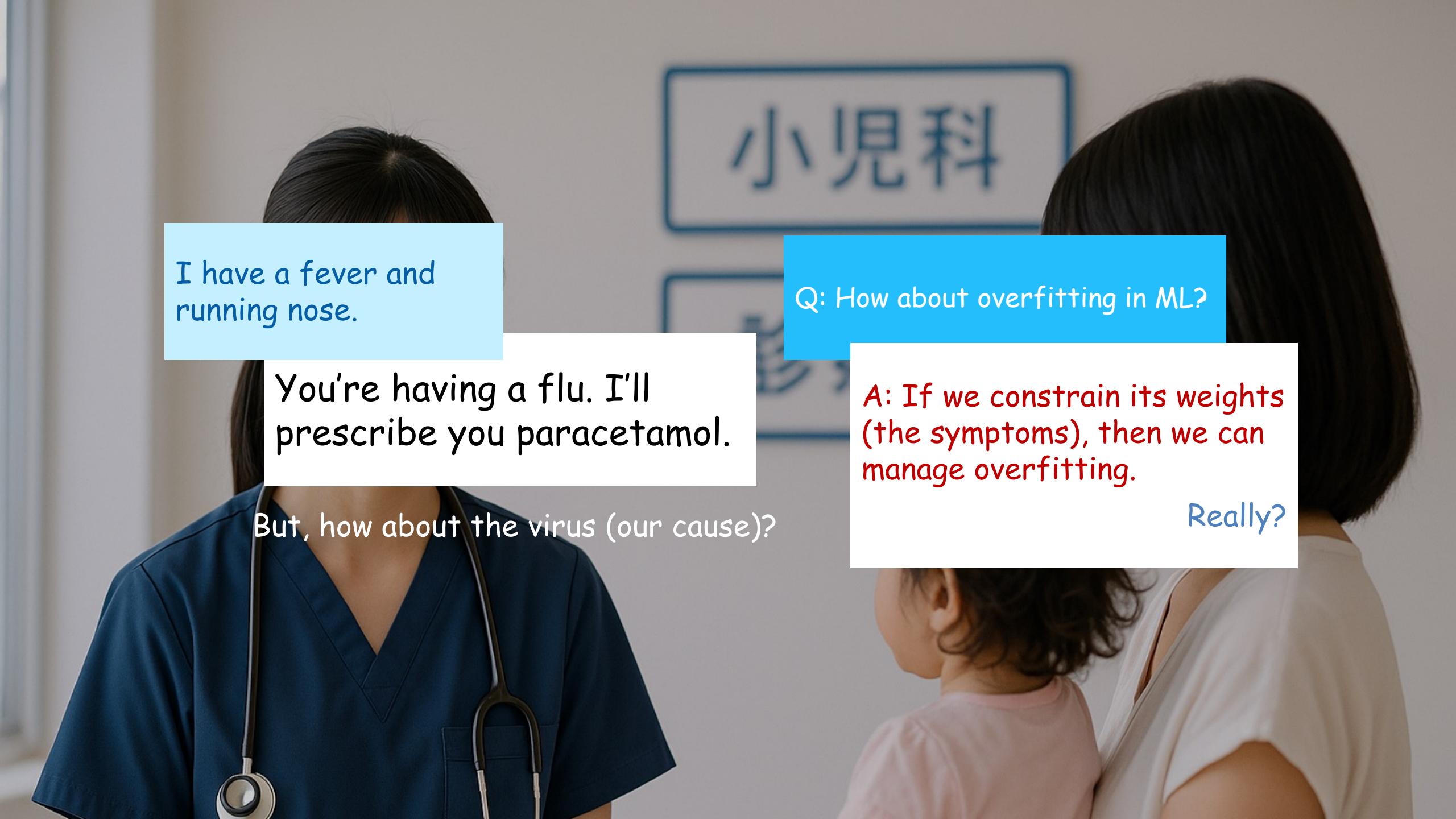
# Machine Learning

## Regularisation

Tarapong Sreenuch

8 February 2024

克明峻德，格物致知



I have a fever and running nose.

You're having a flu. I'll prescribe you paracetamol.

But, how about the virus (our cause)?

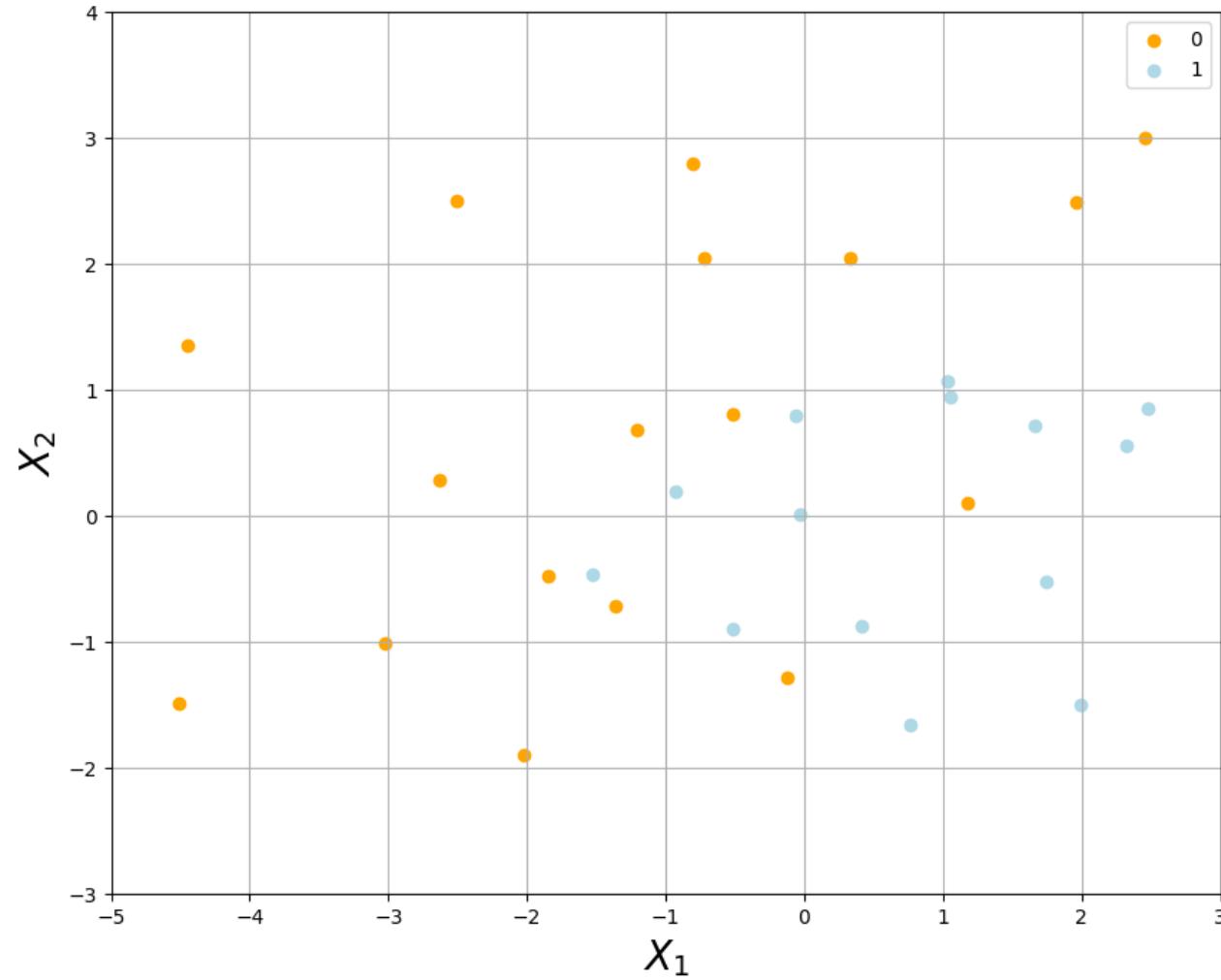
Q: How about overfitting in ML?

A: If we constrain its weights (the symptoms), then we can manage overfitting.

Really?

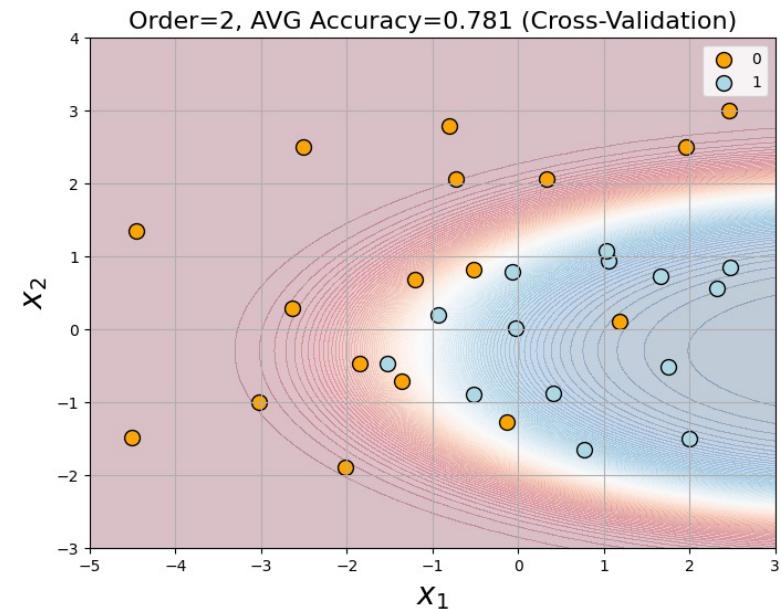
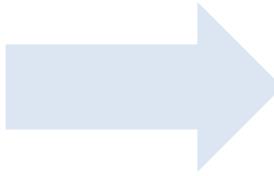
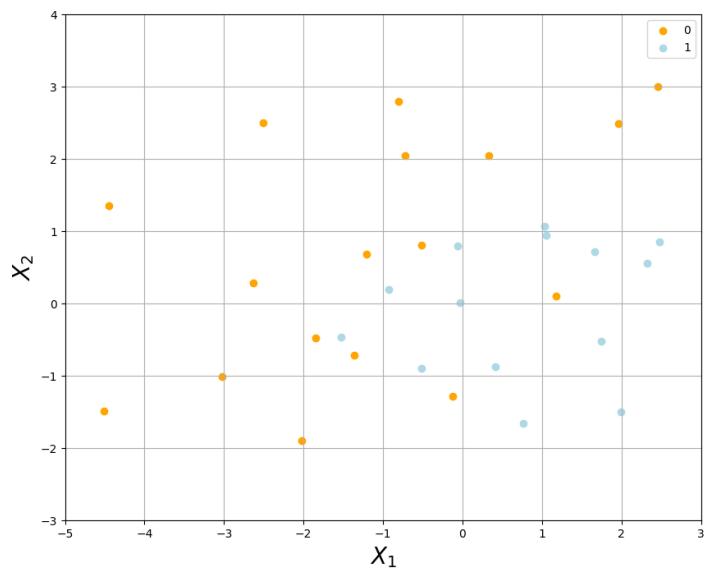
# Example Dataset

---



# Logistic Regression (2<sup>nd</sup> Order)

Feature	Weight
1	1.717
$x_1$	1.385
$x_2$	-0.579
$x_1^2$	-0.167
$x_2^2$	0.978

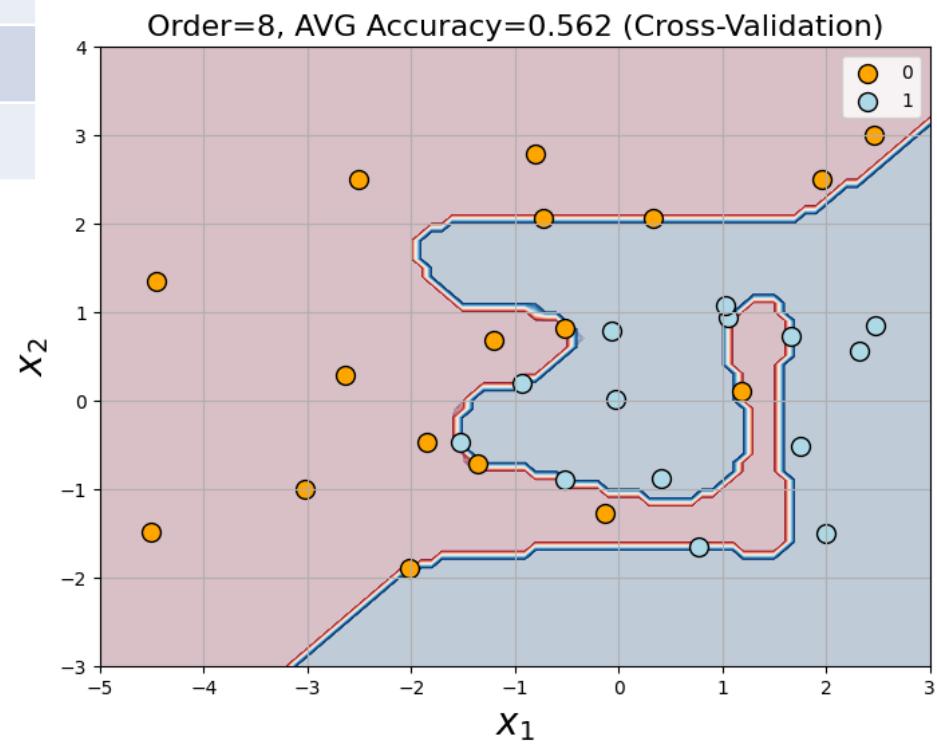


# Logistic Regression (8<sup>th</sup> Order)

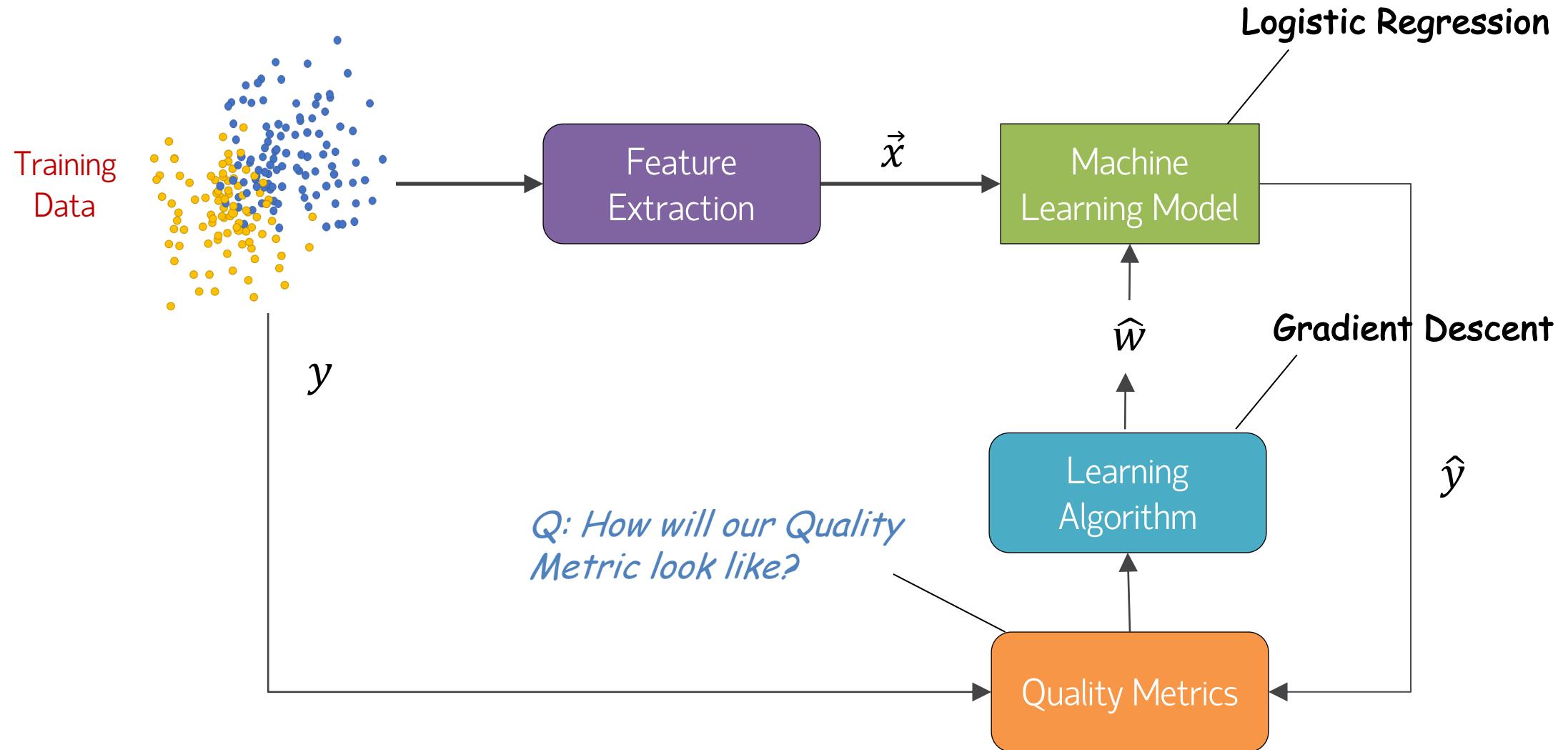
Feature	Weight
1	587.984
$x_1$	495.169
$x_2$	-492.094
$x_1^2$	-76.356
$x_2^2$	-409.916
$x_1^3$	-53.310
$x_2^3$	286.153
$x_1^4$	-356.961
$x_2^4$	-120.300
$x_1^5$	-457.228
$x_2^5$	540.862

Feature	Weight
$x_1^6$	74.364
$x_2^6$	233.336
$x_1^7$	166.729
$x_2^7$	-167.182
$x_1^8$	17.001
$x_2^8$	-31.508

*Overfitting → Large Weights*



# Workflow: Logistic Regression



## Desired Total Cost

---

*Want to Balance*

$$\text{Total Cost} = \text{Measure of Fit} + \text{Measure of Magnitude of Weights}$$
$$-L(\vec{w}) \qquad \qquad \qquad \|\vec{w}\|_2^2 = (w^{(1)})^2 + \dots + (w^{(M)})^2$$

*Negative Log Likelihood*

# Resulting Objective

---

Minimising Total Cost Function:

$$\min_{\vec{w}} -L(\vec{w}) + \lambda \|\vec{w}\|_2^2$$

Penalty Parameter, i.e. Balance of Fit and Magnitude

Ridge Regression (aka. L2 Regularisation)

# Gradient Vector

---

Total Cost Function:

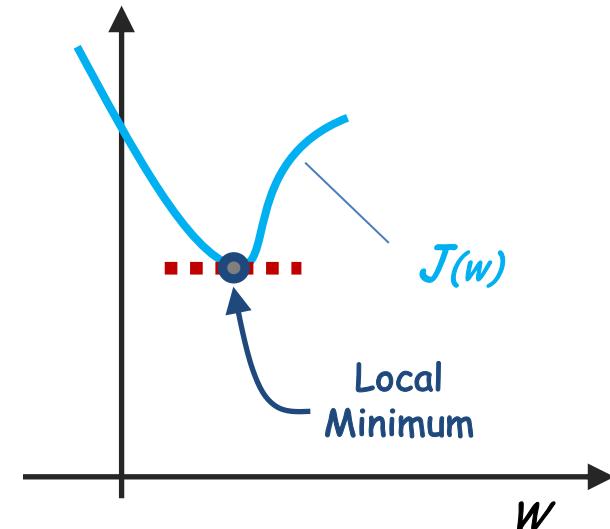
$$J(\vec{w}) = - \sum_{i=1}^N [y_i \log P(1|\vec{x}_i, \vec{w}) + (1 - y_i) \log(1 - P(1|\vec{x}_i, \vec{w}))] + \lambda \|\vec{w}\|_2^2$$

Derivative of Total Cost Function:

$$\frac{\partial J(\vec{w})}{\partial w^{(j)}} = - \sum_{i=1}^N (y_i - P(1|\vec{x}_i, \vec{w})) x_i^{(j)} + 2\lambda \vec{w}$$

Gradient Vector:

$$\nabla_{\vec{w}} J(\vec{w}) = -X^T(Y - P(1|X, \vec{w})) + 2\lambda \vec{w}$$



# Pseudocode for Logistic Regression with $L_2$ Regularisation

```
Function Logistic_Regress(new_point, data_points, labels, learning_rate, max_iterations, tolerance, lambda)
Begin
    // Step 1: Add bias term (column of ones) to data_points
    Add a column of ones to data_points: X

    // Step 2: Initialize weights (theta) to zeros
    Set weights = vector of zeros with the same length as the number of columns in X

    // Step 3: Train the logistic regression model using gradient descent with L2 regularization
    For iteration = 1 to max_iterations do
        // Step 3.1: Compute predictions for all data points in a single vectorized operation
        z = X * weights
        predictions = 1 / (1 + exp(-z))

        // Step 3.2: Calculate the gradient with L2 regularization
        errors = labels - predictions
        gradients = -(transpose(X) * errors / m) + (lambda * weights / m)

        // Step 3.3: Update weights
        weights = weights - learning_rate * gradients

        // Step 3.4: Check for convergence
        If the magnitude of all elements in gradients < tolerance then
            Break the loop

    // Step 4: Add bias term to new_point
    Add a 1 (bias term) to new_point: x // x is of shape (n+1, 1)

    // Step 5: Predict the probability for new_point
    z_new = transpose(weights) * x
    probability = 1 / (1 + exp(-z_new)) // Sigmoid function

    Return probability
End
```

# Python Code Snippet

```
def Logistic_Regress(new_point, data_points, labels, learning_rate=0.01, max_iterations=1000, tolerance=1e-6, lambda_=0.1):
    # Step 1: Add bias term (column of ones) to data_points
    X = np.hstack([np.ones((data_points.shape[0], 1)), data_points])

    # Step 2: Train the model using gradient descent with L2 regularization
    weights = gradient_descent_logistic(X, labels, learning_rate, max_iterations, tolerance, lambda_)

    # Step 3: Add bias term to new_point
    x = np.hstack([1, new_point])

    # Step 4: Predict the probability for new_point
    probability = expit(x @ weights)

    # Return the predicted probability
    return probability
```

# Python Code Snippet (cont.)

```
import numpy as np
from scipy.special import expit # Optimized sigmoid function

def compute_gradient(X, y, weights, lambda_):
    predictions = expit(X @ weights) # Predicted probabilities using sigmoid function
    errors = y - predictions # Error between actual and predicted
    gradients = -(X.T @ errors / X.shape[0]) + lambda_ * weights # Regularized gradient

    return gradients

def gradient_descent_logistic(X, y, learning_rate=0.01, max_iterations=1000, tolerance=1e-6, lambda_=0.1):
    # Initialize weights
    weights = np.zeros((X.shape[1], 1))

    for iteration in range(max_iterations):
        # Compute gradient with L2 regularization
        gradients = compute_gradient(X, y, weights, lambda_)

        # Update weights
        weights = weights - learning_rate * gradients

        # Check for convergence
        if np.linalg.norm(gradients) < tolerance:
            print(f"Converged after {iteration} iterations.")
            break

    return weights
```

# Usage Example

```
# Example usage
# Generate random data_points and labels for demonstration
# Set parameters for data generation
m, n = 100, 2 # Number of samples per class and number of features

np.random.seed(0)

# Generate data points for two classes
class_0 = np.hstack((1.5 + np.random.randn(m, 1), -1.5 + np.random.randn(m, 1)))
class_1 = np.hstack((-1.5 + np.random.randn(m, 1), 1.5 + np.random.randn(m, 1)))

# Combine the two classes into a single dataset
data_points = np.vstack((class_0, class_1))

# Generate labels: 0 for the first class, 1 for the second class
labels = np.vstack((np.zeros((m, 1)), np.ones((m, 1)))))

# Define a new data point
new_point = np.array([1, -1]) # Example new point without bias term

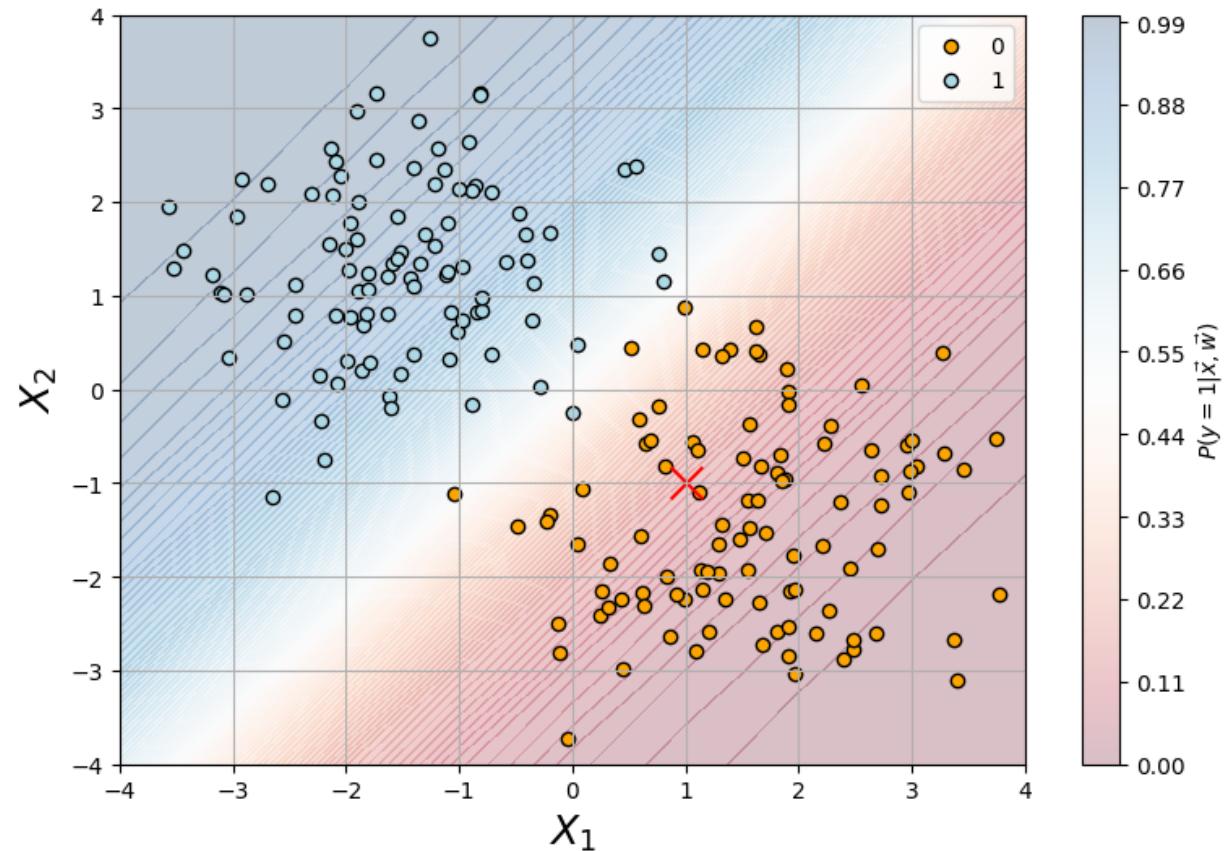
# Predict the probability for new_point with L2 regularization
learning_rate = 0.1
max_iterations = 1000
tolerance = 1e-6
Lambda_ = 0.1

probability = Logistic_Regress(new_point, data_points, labels, learning_rate, max_iterations, tolerance, lambda_)

print(f"Predicted probability for new point: {probability}")
```

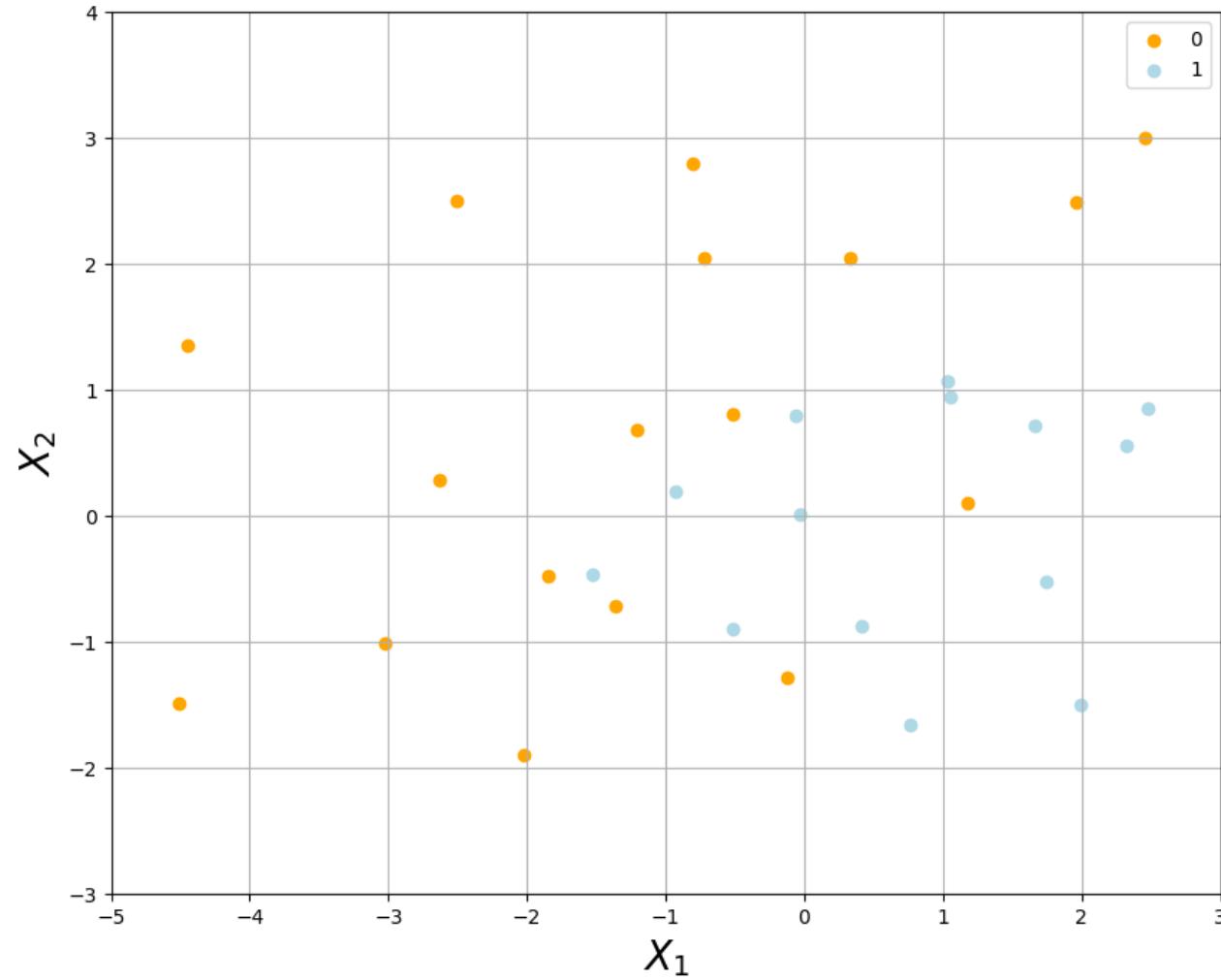
# Usage Example

Converged after 416 iterations.  
Predicted probability for new point: [0.1415866]



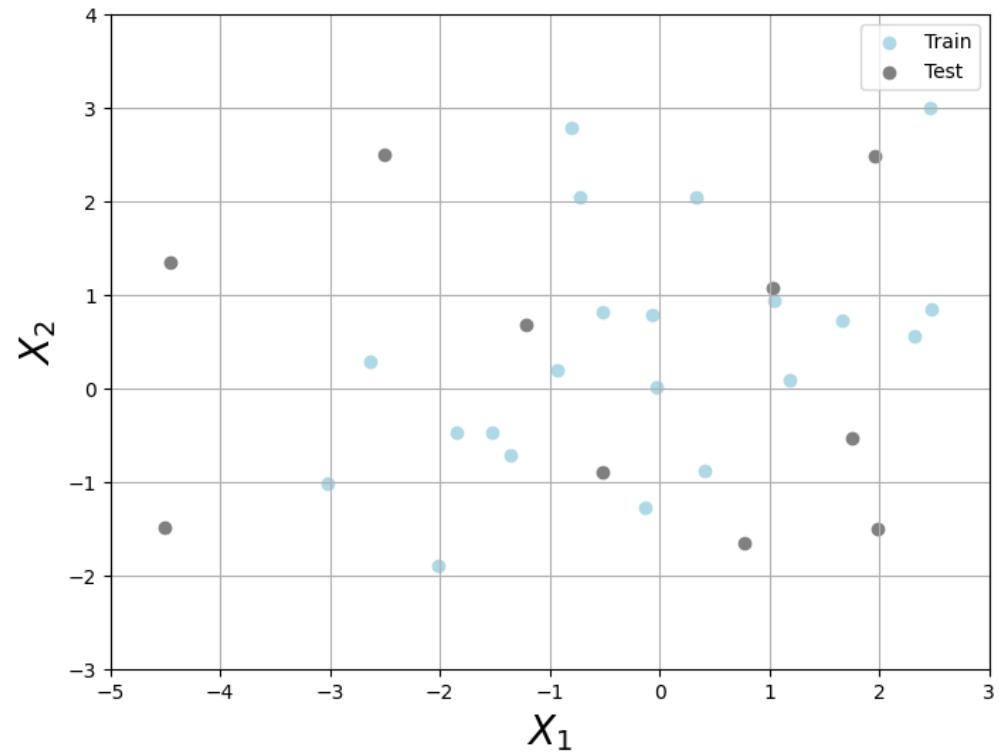
# Example Dataset

---



# Train, Validation and Test Datasets

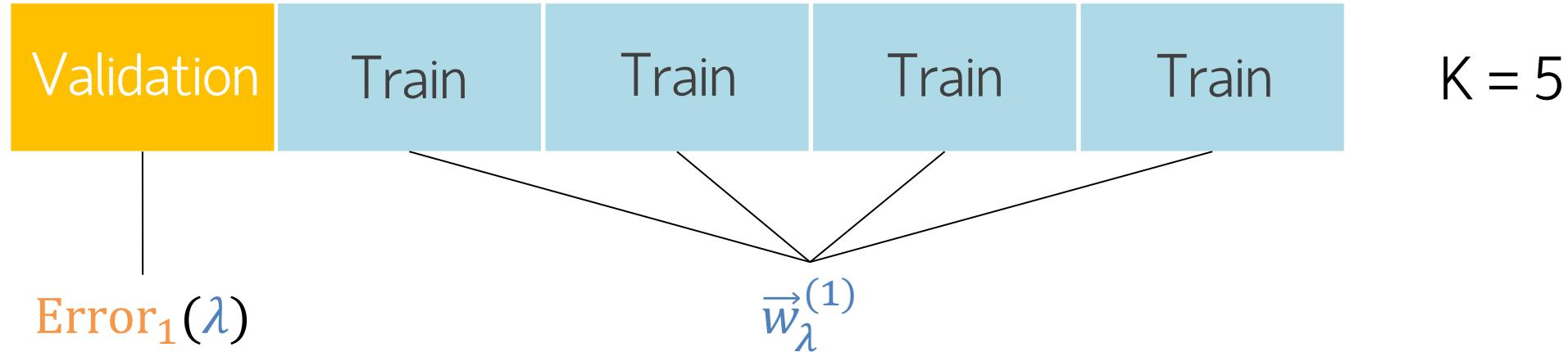
```
from sklearn.model_selection import train_test_split  
  
# First, split the data into train (70%) and test (30%)  
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.3, random_state=42, stratify=y)
```



- Train:Test is typically either 0.8:0.2 or 0.7:0.3.
- Test dataset is a proxy of unseen data, and it will only be used in the final evaluation.
- Train dataset is further divided into k folds (e.g., 5 or 10). For each fold, the model is trained on  $k-1$  folds and validated on the remaining fold.
- We use **K-Fold Cross-Validation** to fine-tune or optimize the ML model. Here, we find a hyper-parameter  $\lambda$  based on the average performance across folds.

# K-Fold Cross Validation

---

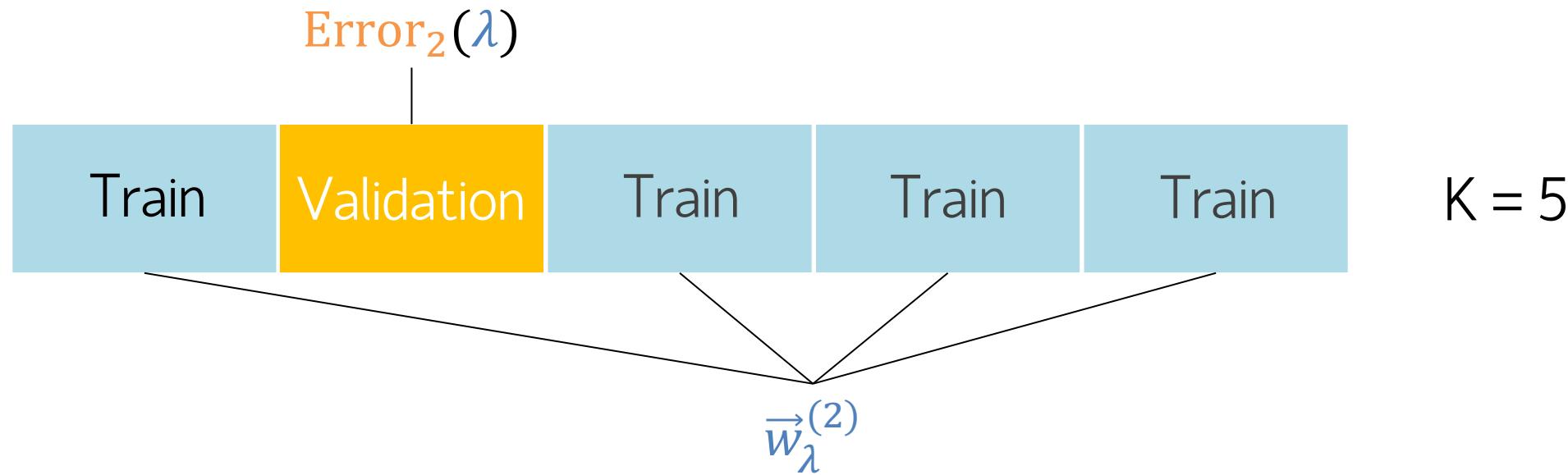


For  $k = 1, \dots, K$

Step 1: Estimate  $\vec{w}_\lambda^{(k)}$  on the training blocks.

Step 2: Compute error on Validation Block:  $\text{Error}_k(\lambda)$ .

# K-Fold Cross Validation

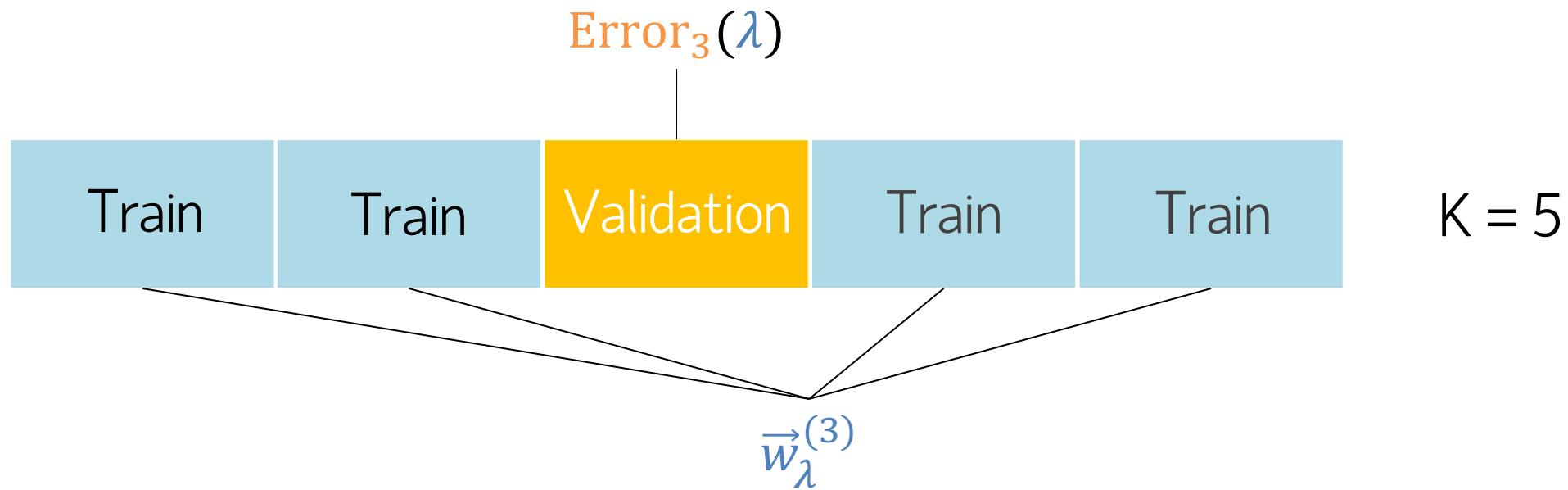


For  $k = 1, \dots, K$

Step 1: Estimate  $\vec{w}_\lambda^{(k)}$  on the training blocks.

Step 2: Compute error on Validation Block: Error<sub>k</sub>(λ).

# K-Fold Cross Validation

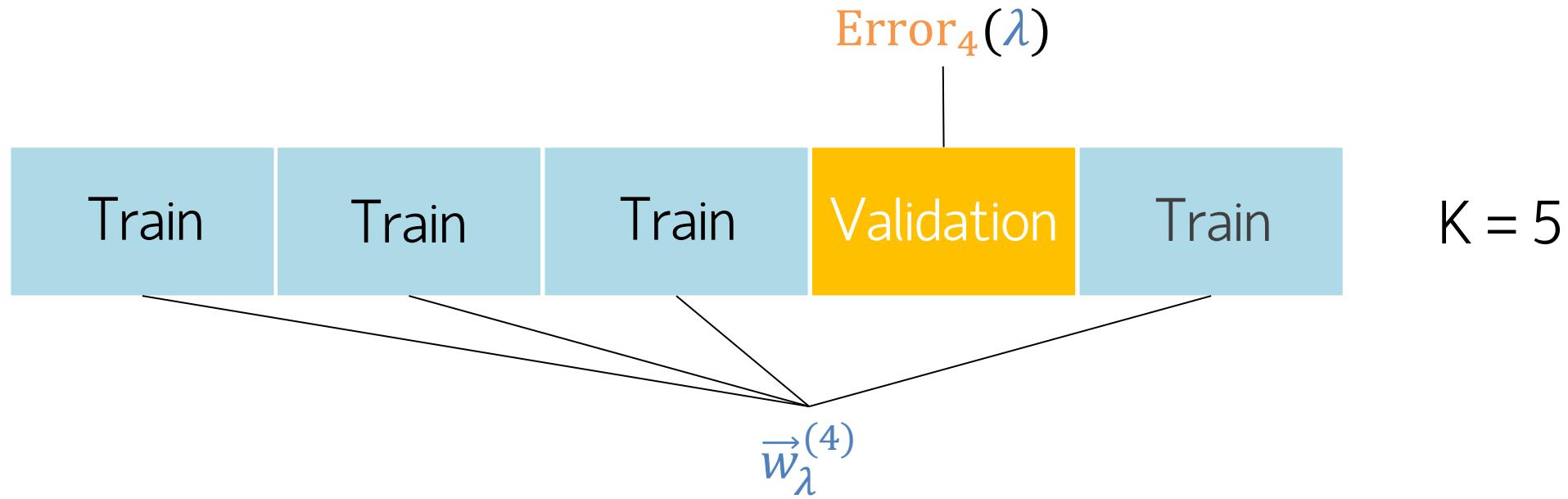


For  $k = 1, \dots, K$

Step 1: Estimate  $\vec{w}_\lambda^{(k)}$  on the training blocks.

Step 2: Compute error on Validation Block: Error<sub>k</sub>(λ).

# K-Fold Cross Validation



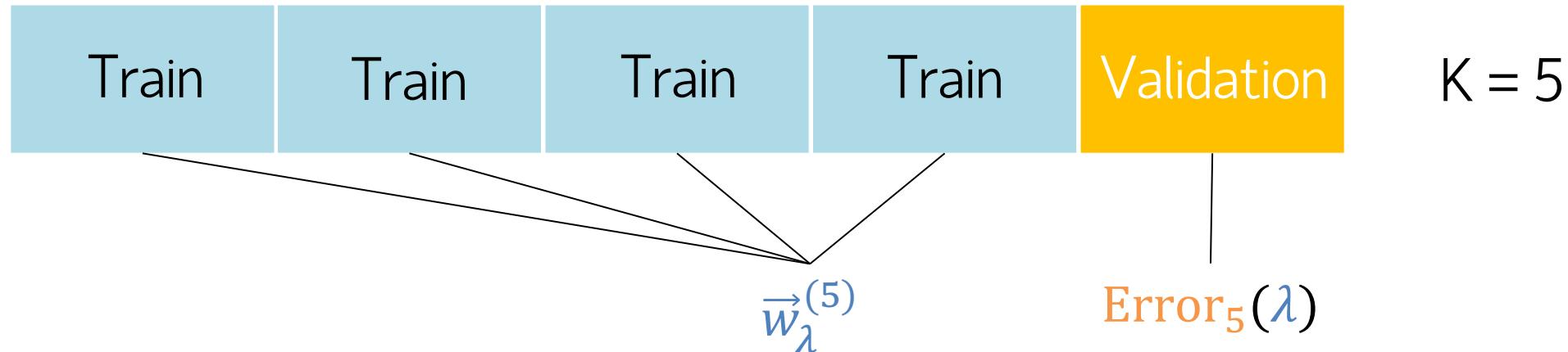
For  $k = 1, \dots, K$

Step 1: Estimate  $\vec{w}_\lambda^{(k)}$  on the training blocks.

Step 2: Compute error on Validation Block:  $\text{Error}_k(\lambda)$ .

# K-Fold Cross Validation

---



For  $k = 1, \dots, K$

Step 1: Estimate  $\vec{w}_\lambda^{(k)}$  on the training blocks.

Step 2: Compute error on Validation Block:  $\text{Error}_k(\lambda)$ .

Compute Average Error:  $\text{CV}(\lambda) = \frac{1}{K} \sum_{k=1}^K \text{Error}_k(\lambda)$ .

# scikit-learn: Logistic Regression with $L_2$ Regularisation

```
# Step 1: Standardize the features using only the training data
scaler = StandardScaler()
X_train = scaler.fit_transform(X_train)
X_test = scaler.transform(X_test)

# Step 2: Create an instance of LogisticRegression
lambda = 10000
model = LogisticRegression(penalty='l2', C=1/lambda, random_state=204)

# Step 3: Define cross-validation strategy
k = 5
kf = KFold(n_splits=k, shuffle=True, random_state=42)

# Step 4: Perform cross-validation on the training data
cv_scores = cross_val_score(model, X_train, y_train, cv=kf, scoring='accuracy')
print("Mean Cross-Validation Accuracy: {:.2f}".format(np.mean(cv_scores)))
```

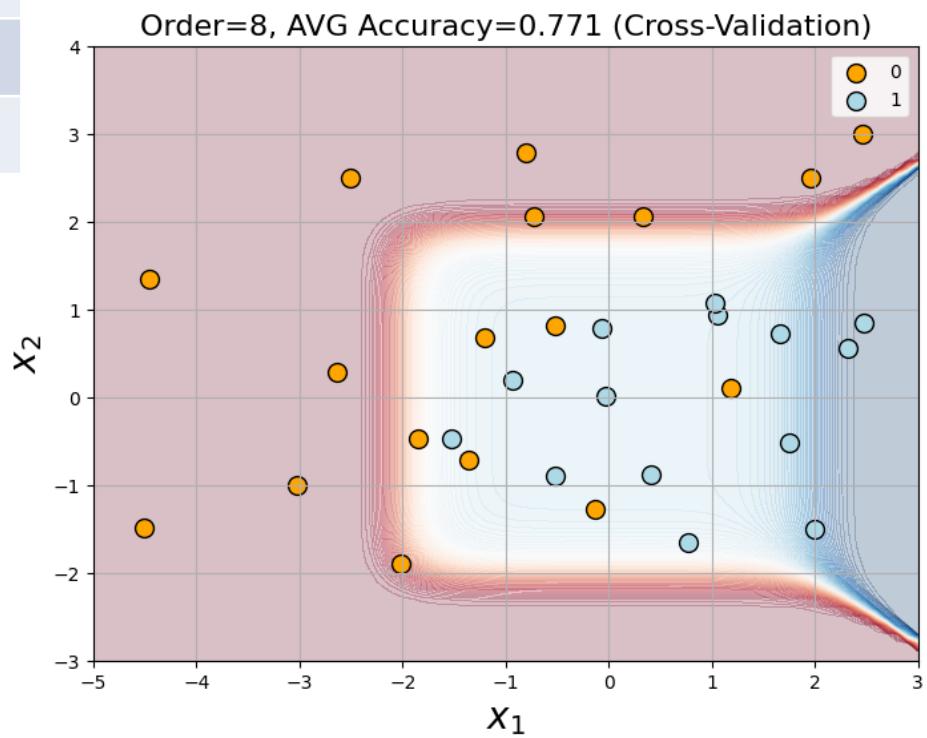
Mean Cross-Validation Accuracy: 0.77



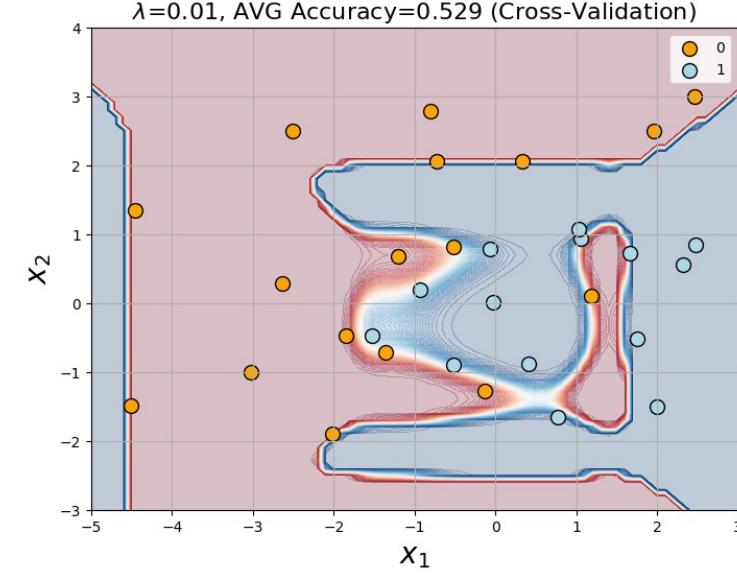
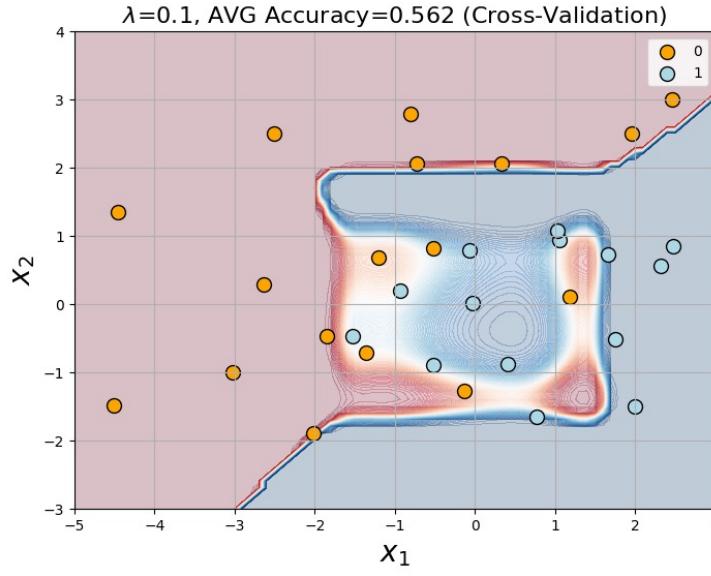
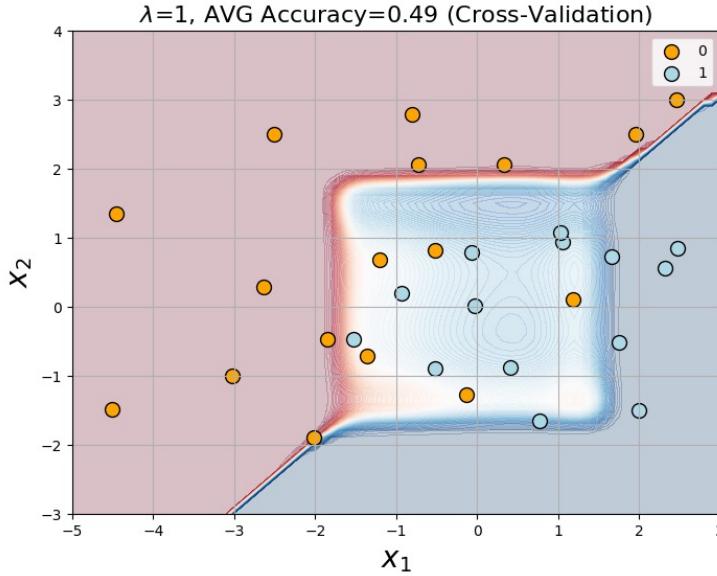
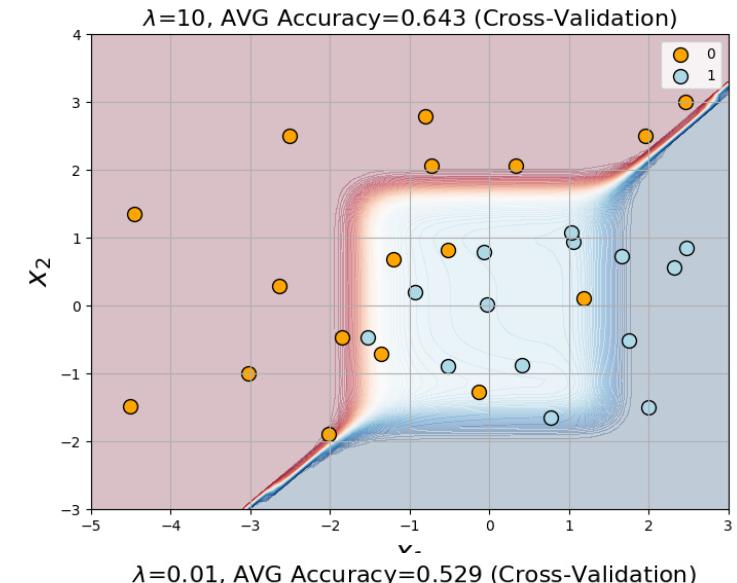
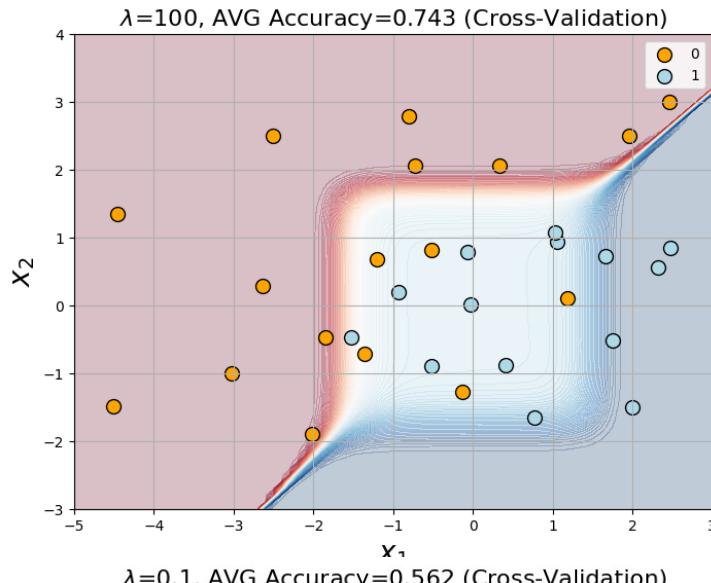
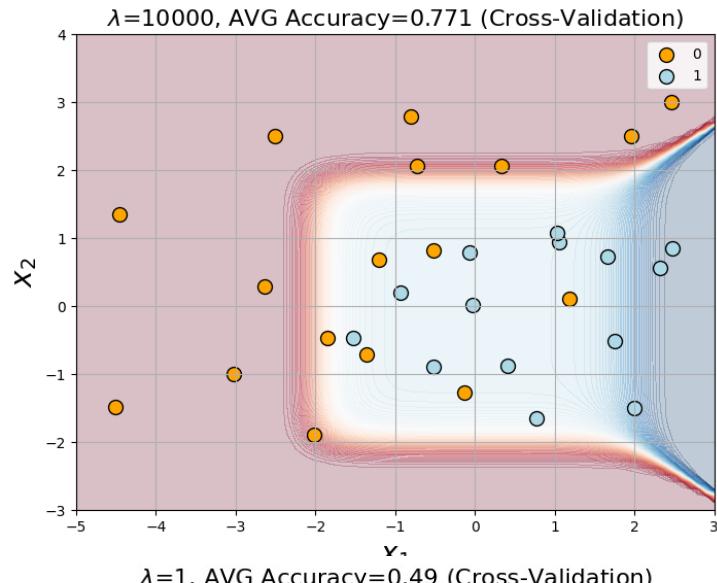
# Regularised Decision Boundary (8<sup>th</sup> Order)

Feature	Coefficient
1	0.609
$x_1$	3.661e-4
$x_2$	-3.319e-5
$x_1^2$	-6.429e-5
$x_2^2$	-2.815e-5
$x_1^3$	8.919e-4
$x_2^3$	-1.843e-4
$x_1^4$	-1.316e-4
$x_2^4$	-2.531e-4
$x_1^5$	2.826e-3
$x_2^5$	-8.357e-4

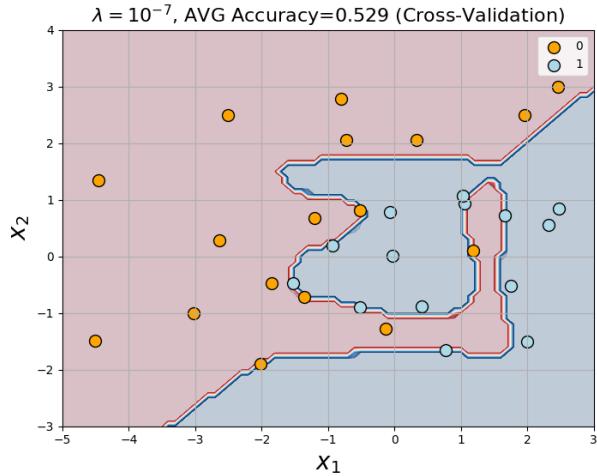
Feature	Coefficient
$x_1^6$	-2.408e-4
$x_2^6$	-1.386e-3
$x_1^7$	9.870e-3
$x_2^7$	-3.375e-3
$x_1^8$	-3.631e-4
$x_2^8$	-6.624e-3



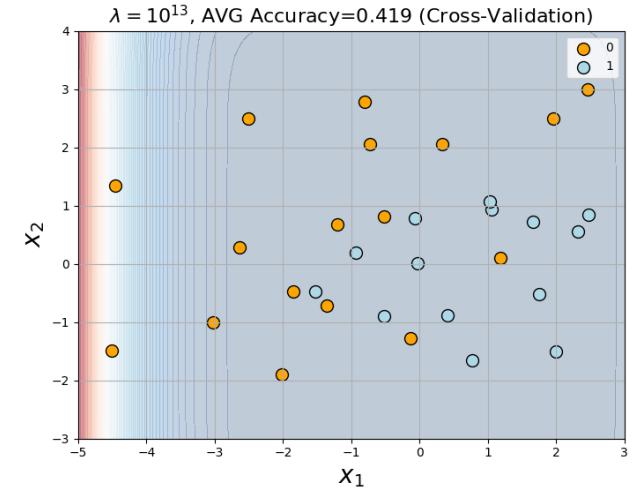
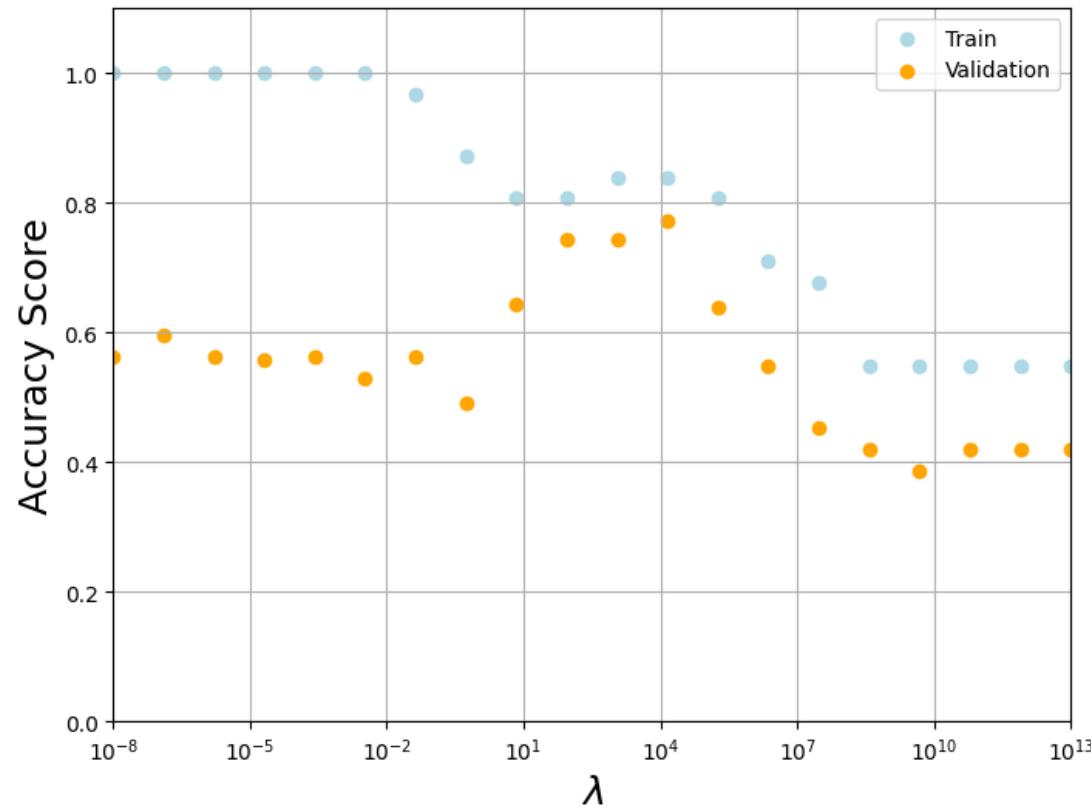
# Model Complexity



# Bias-Variance Trade-Offs



- High Variance, Low Bias
- Complex Decision Boundary
- High Train Accuracy
- Low Validation Accuracy

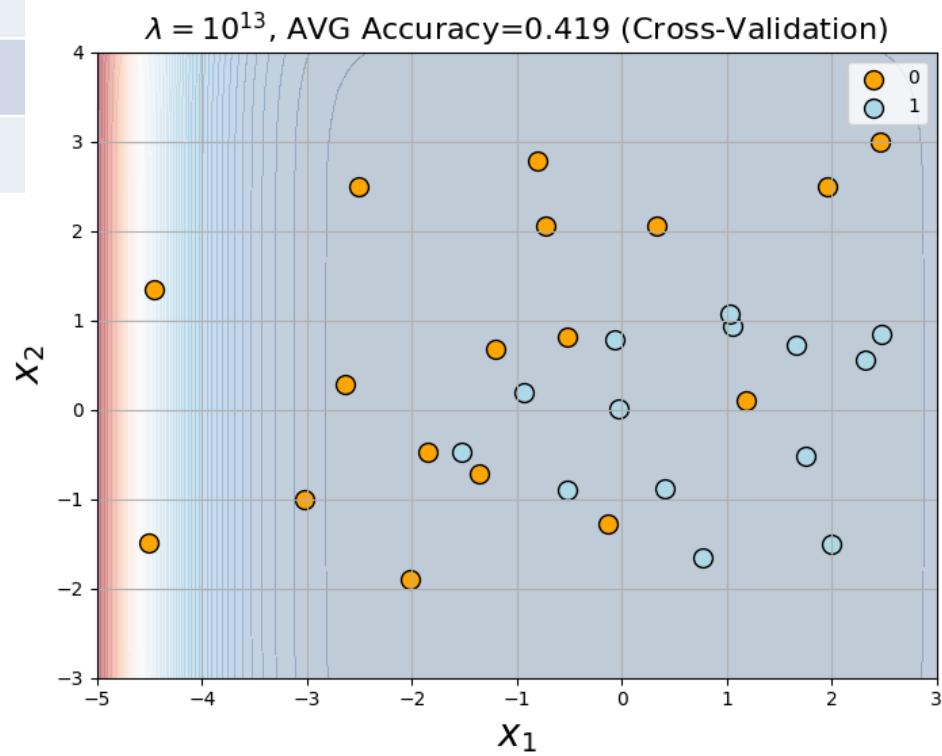


- High Bias, Low Variance
- Simple Decision Boundary
- Low Train Accuracy
- Low Validation Accuracy

# Regularised Decision Boundary ( $\lambda = 10^{13}$ )

Feature	Coefficient
1	-0.194
$x_1$	1.463e-12
$x_2$	-5.507e-13
$x_1^2$	-2.369e-12
$x_2^2$	-1.633e-12
$x_1^3$	1.310e-11
$x_2^3$	-4.208e-12
$x_1^4$	-4.208e-11
$x_2^4$	-1.171e-11
$x_1^5$	1.935e-10
$x_2^5$	-3.018e-11

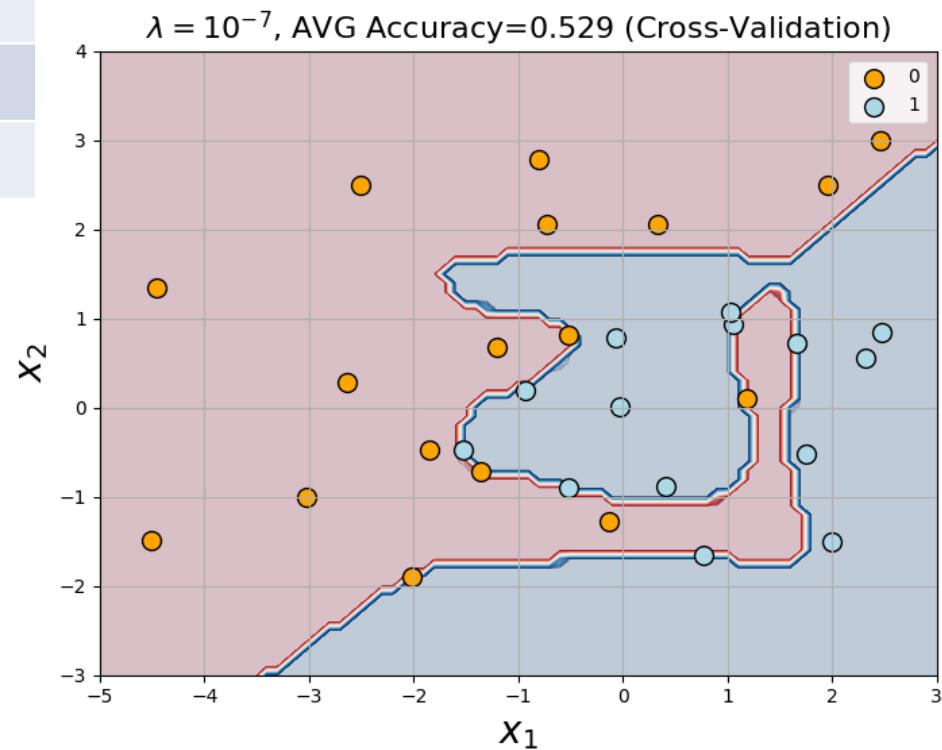
Feature	Coefficient
$x_1^6$	-7.800e-10
$x_2^6$	-8.390e-11
$x_1^7$	3.483e-9
$x_2^7$	-2.244e-10
$x_1^8$	1.508e-8
$x_2^8$	-6.308e-10



# Regularised Decision Boundary ( $\lambda = 10^{-7}$ )

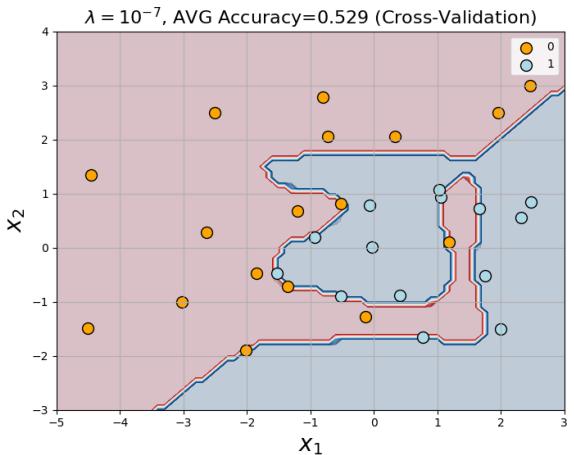
Feature	Coefficient
1	432.548
$x_1$	333.613
$x_2$	-352.095
$x_1^2$	-61.796
$x_2^2$	-256.131
$x_1^3$	-36.690
$x_2^3$	219.2734
$x_1^4$	-246.131
$x_2^4$	-106.567
$x_1^5$	-310.860
$x_2^5$	405.405

Feature	Coefficient
$x_1^6$	44.591
$x_2^6$	107.260
$x_1^7$	110.216
$x_2^7$	-150.206
$x_1^8$	11.905
$x_2^8$	15.687

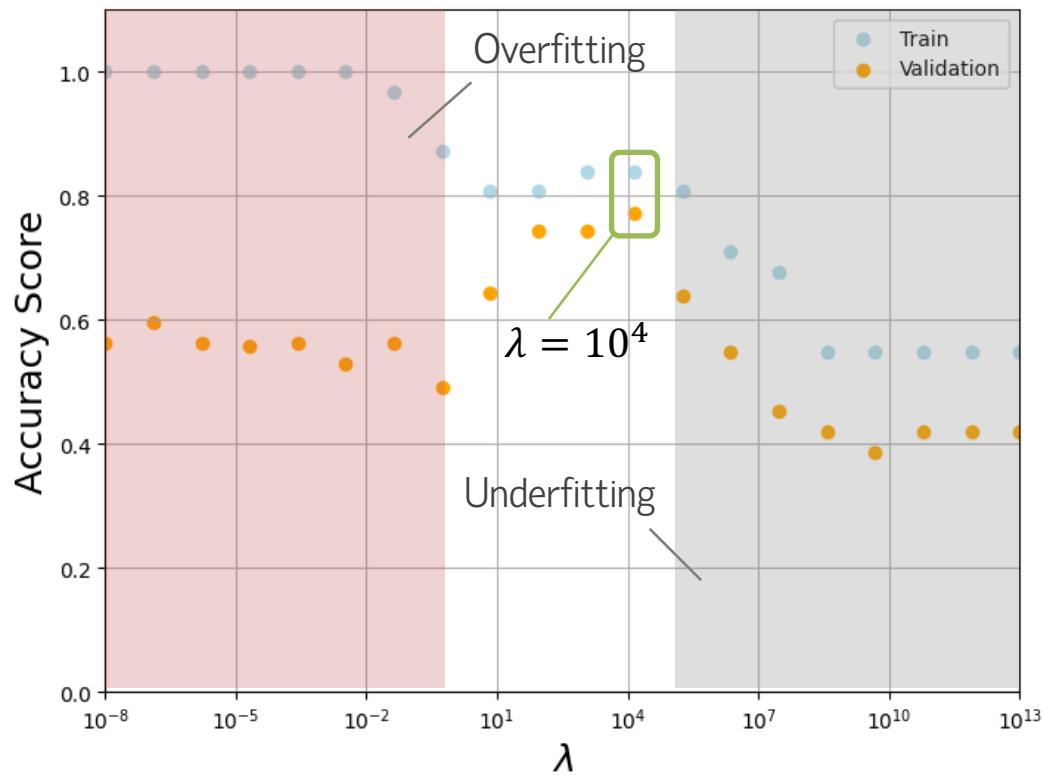


# Overfitting vs Underfitting

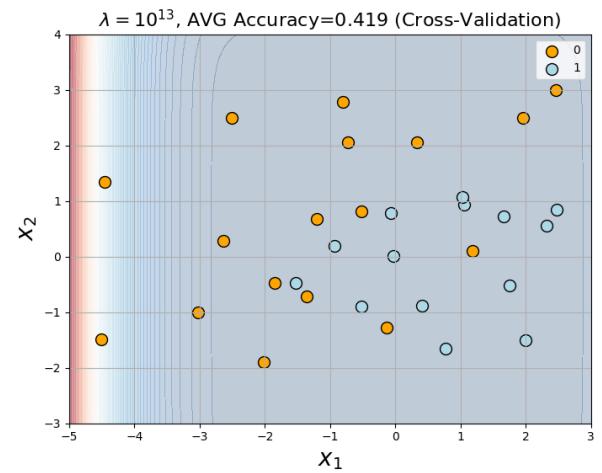
- **Overfitting:** With minimal penalty on weights, the model captures noise in the training data, leading to high variance and poor generalization on validation data.
- **Underfitting:** A strong penalty on weights forces them to shrink, overly simplifying the model. This results in high bias, causing it to miss important patterns in both training and validation data.



- High Variance, Low Bias
- Complex Decision Boundary
- High Train Accuracy
- Low Validation Accuracy



- **Optimal:** The ideal  $\lambda$  imposes just enough weight penalty to prevent overfitting while preserving the model's ability to capture true patterns, leading to accurate predictions on both training and unseen data.



- High Bias, Low Variance
- Simple Decision Boundary
- Low Train Accuracy
- Low Validation Accuracy

# MyLogisticRegressor (Revised)

```
import numpy as np
from scipy.special import expit # Sigmoid function
from sklearn.base import BaseEstimator, ClassifierMixin

class MyLogisticRegressor(BaseEstimator, ClassifierMixin):
    def __init__(self, learning_rate=0.01, max_iter=1000, tol=1e-6, penalty='l2', lambda_=0.1):
        self.learning_rate = learning_rate
        self.max_iter = max_iter
        self.tol = tol
        self.penalty = penalty
        self.lambda_ = lambda_
        self.weights_ = None
```

# MyLogisticRegressor (cont.)

```
def _add_bias(self, X):
    """Add a column of ones to X to account for the bias term."""
    return np.hstack([np.ones((X.shape[0], 1)), X])

def _compute_gradient(self, X, y, weights):
    """Compute the gradient of the cost function."""
    predictions = expit(X @ weights) # Predicted probabilities using sigmoid function
    errors = y - predictions # Error between actual and predicted
    gradients = -X.T @ errors / X.shape[0] # Average gradient

    # Add L2 regularization if penalty is set to 'l2'
    if self.penalty == 'l2':
        gradients = gradient + self.lambda_ * weights

    return gradients

def _gradient_descent(self, X, y):
    """Perform gradient descent to optimize weights."""
    weights = np.zeros((X.shape[1], 1))

    for iteration in range(self.max_iter):
        gradients = self._compute_gradient(X, y, weights)
        weights += self.learning_rate * gradients

        # Check for convergence
        if np.linalg.norm(gradients) < self.tol:
            print(f"Converged after {iteration} iterations.")
            break

    return weights
```

# MyLogisticRegressor (cont.)

```
def fit(self, X, y):
    """Fit the logistic regression model to the training data."""
    X = self._add_bias(X)
    y = y.reshape(-1, 1) # Ensure y is a column vector
    self.weights_ = self._gradient_descent(X, y)
    return self

def predict_proba(self, X):
    """Predict probability estimates for samples in X."""
    X = self._add_bias(X)
    probabilities = expit(X @ self.weights_)
    return np.hstack([1 - probabilities, probabilities]) # Return probabilities for classes [0, 1]

def predict(self, X):
    """Predict class labels for samples in X."""
    proba = self.predict_proba(X)
    return (proba[:, 1] ≥ 0.5).astype(int)
```

# Usage Example

```
# Example usage
if __name__ == "__main__":
# Generate some example data
np.random.seed(0)
m, n = 100, 2
X_class0 = np.random.randn(m, n) - 1
X_class1 = np.random.randn(m, n) + 1
X = np.vstack([X_class0, X_class1])
y = np.hstack([np.zeros(m), np.ones(m)])

# Initialize and fit MyLogisticRegressor
model = MyLogisticRegressor(learning_rate=0.1, max_iter=1000, tol=1e-6, penalty='l2', lambda_=0.1)
model.fit(X, y)

# Predict class probabilities and labels
print("Predicted probabilities:\n", model.predict_proba(X)[:5]) # Show probabilities for first 5 samples
print("Predicted labels:\n", model.predict(X)[:5]) # Show labels for first 5 samples
```

Converged after 426 iterations.

Predicted probabilities:

```
[[0.45729389 0.54270611] [0.25744809 0.74255191] [0.7140739 0.2859261 ] [0.73288984 0.26711016] [0.8095836 0.1904164 ]]
```

Predicted labels:

```
[1 1 0 0 0]
```

# Summary

---

- Regularisation helps prevent overfitting by penalising large weights, encouraging models to capture meaningful patterns rather than noise. This leads to simpler, more generalised models that perform well on new data.
- The cost function in regularised logistic regression combines the negative log-likelihood with an added penalty term,  $\lambda \|\vec{w}\|_2^2$ . This term balances minimising error with controlling weight magnitudes to reduce model complexity.
- The regularisation parameter,  $\lambda$ , governs the strength of the penalty. A smaller  $\lambda$  may result in high variance and overfitting, while a larger  $\lambda$  increases bias and risks underfitting. Tuning  $\lambda$  helps strike an optimal balance between variance and bias.
- Regularisation implicitly controls model complexity and performs feature selection by suppressing coefficients. Features with low-magnitude weights are essentially excluded from the model, allowing it to focus on the most impactful features and reducing overfitting.