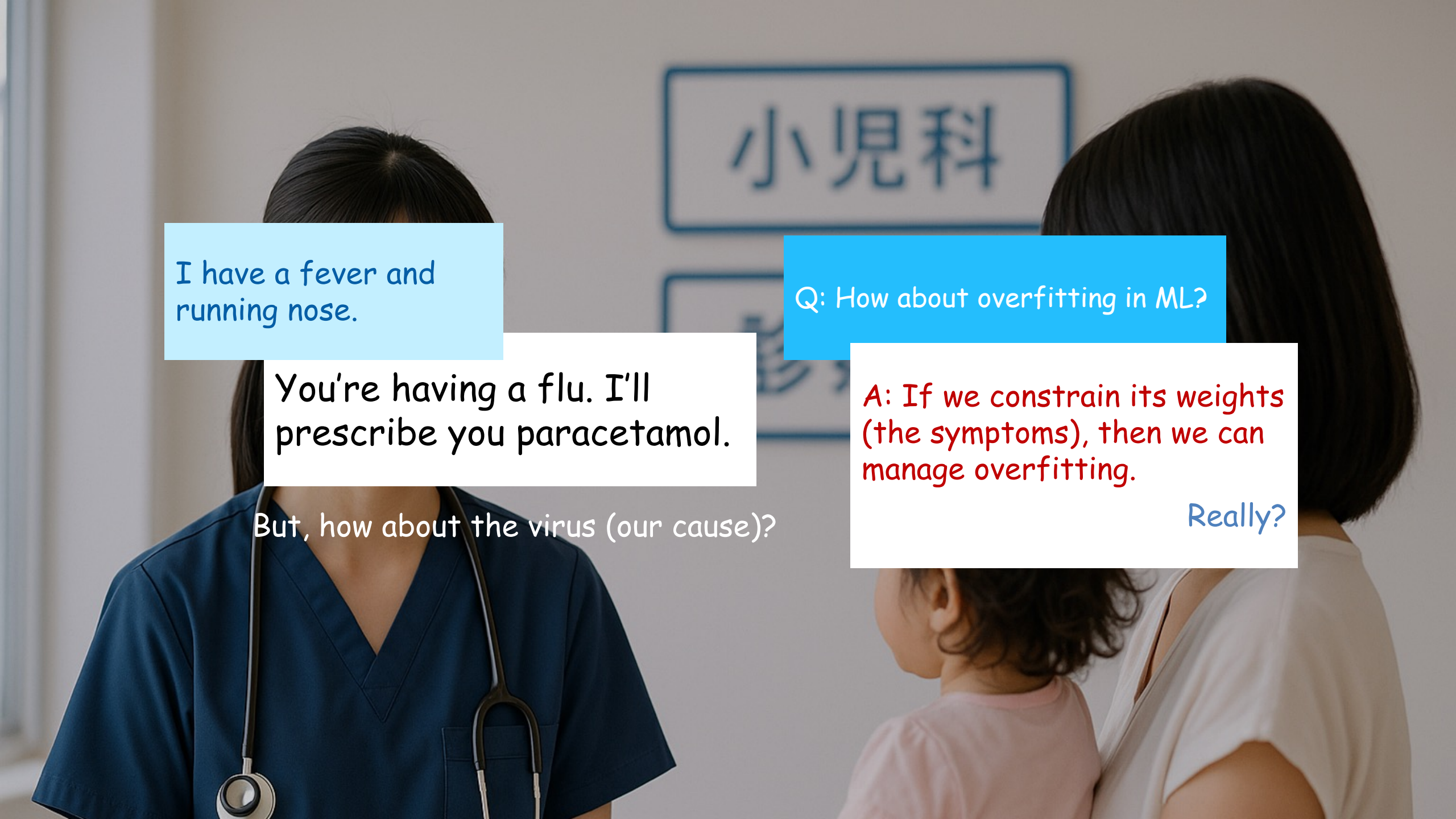
Machine Learning

Regularisation

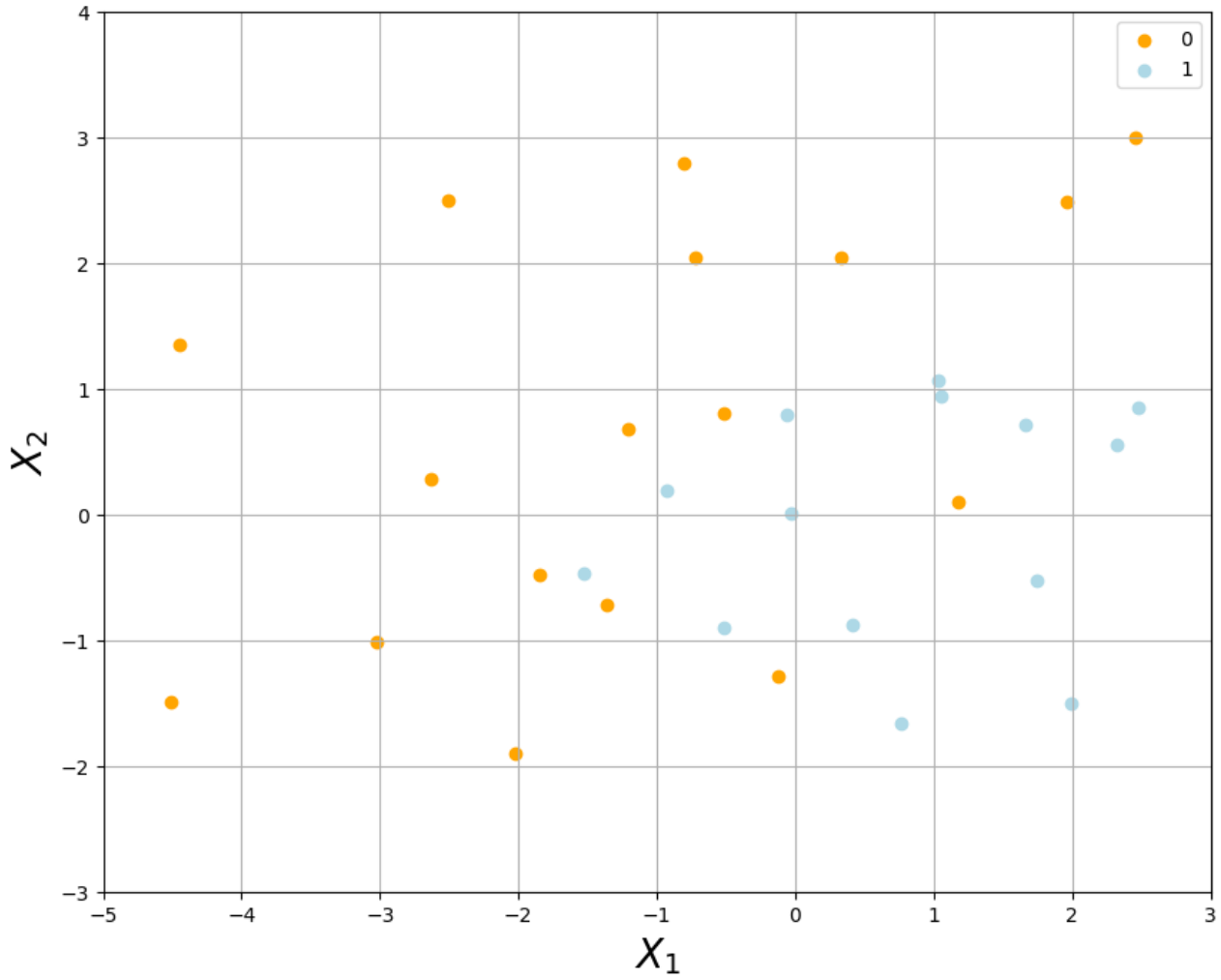
Tarapong Sreenuch

8 February 2024

克明峻德，格物致知

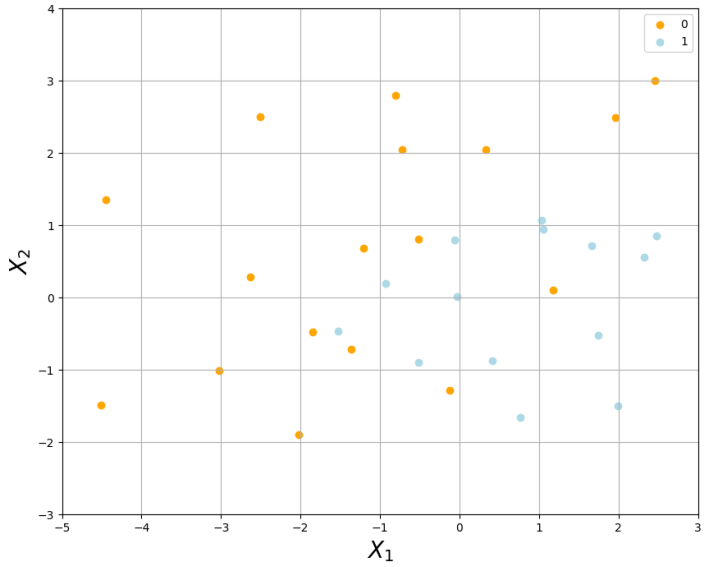

小児科
I have a fever and running nose.
You're having a flu. I'll prescribe you paracetamol.
But, how about the virus (our cause)?
Q: How about overfitting in ML?
A: If we constrain its weights (the symptoms), then we can manage overfitting.
Really?

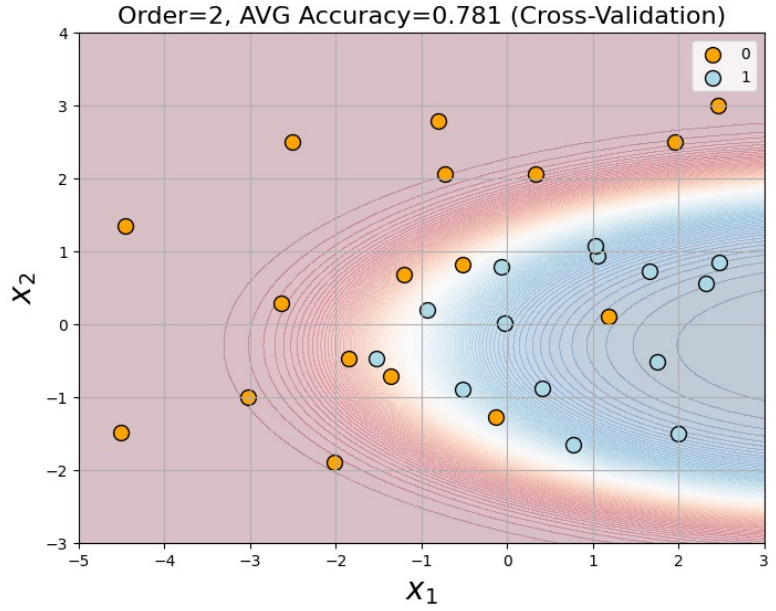
Example Dataset



Logistic Regression (2nd Order)

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



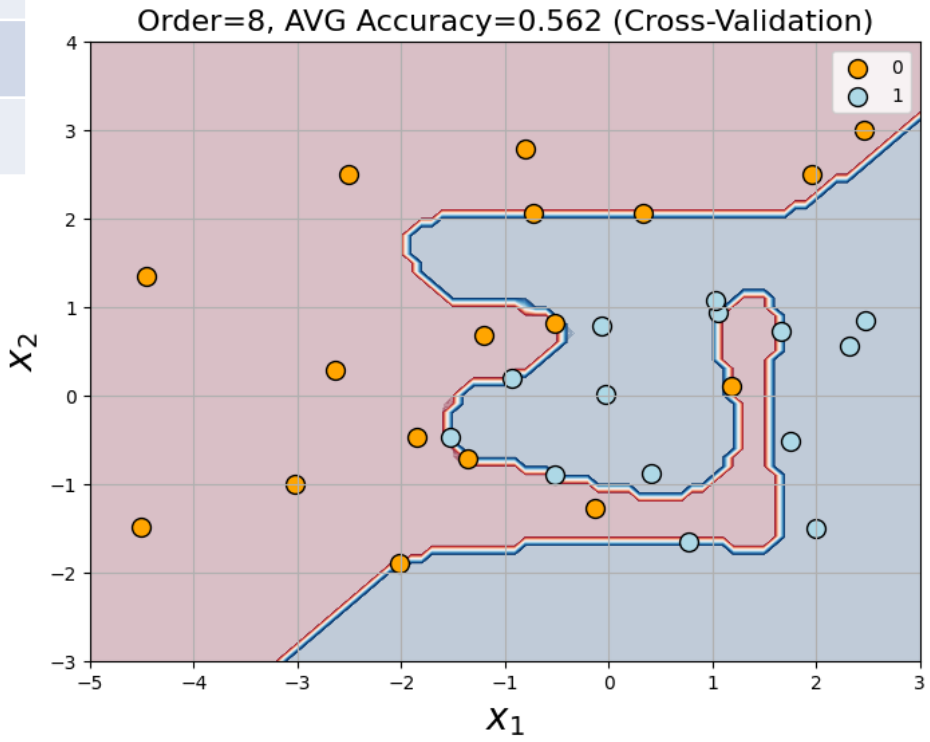


Logistic Regression (8^{th} Order)

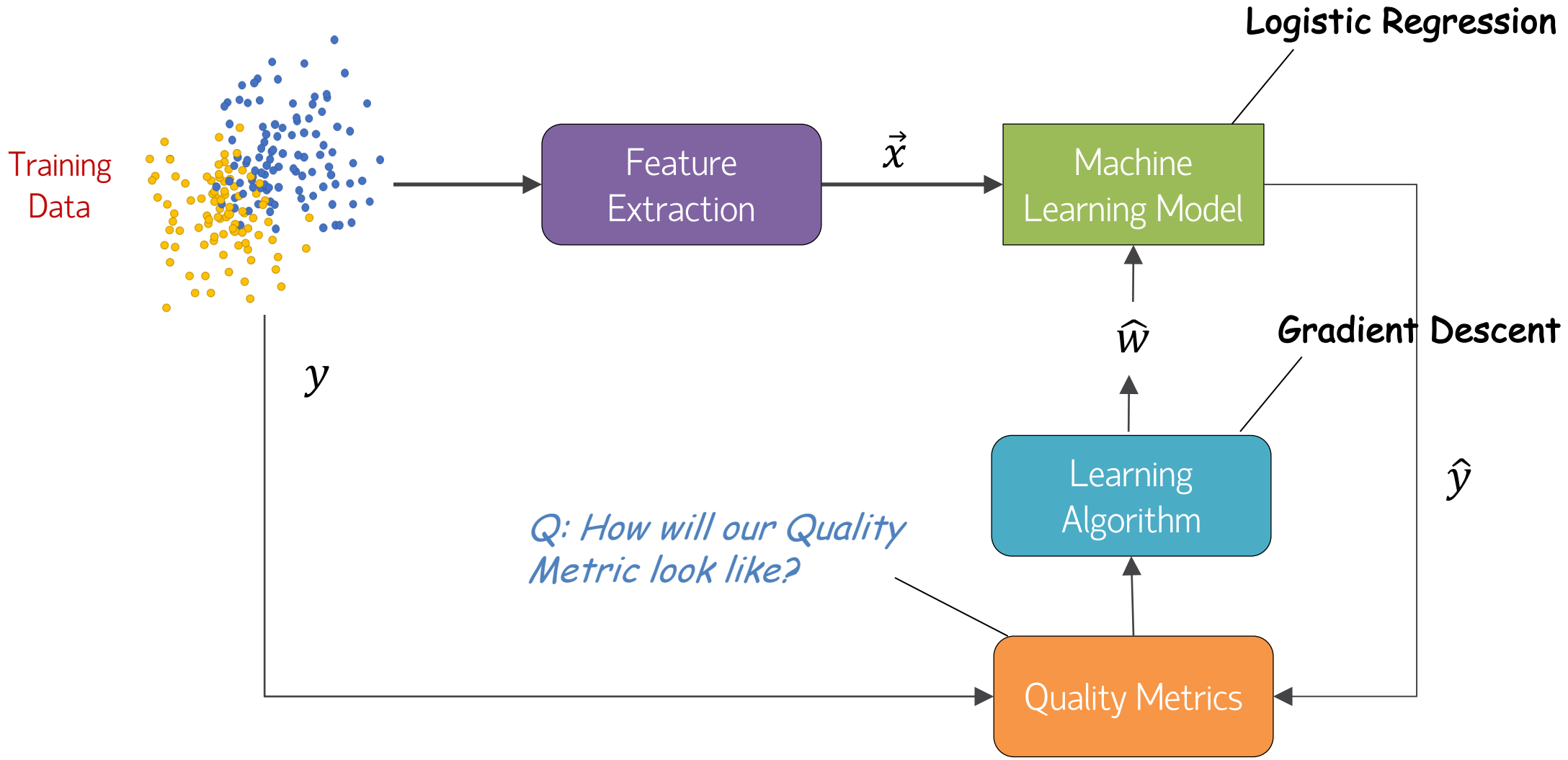
Feature	Weight
1	587.984
x_1	495.169
x_2	-492.094
x_1^2	-76.356
x_2^2	-409.916
x_1^3	-53.310
x_2^3	286.153
x_1^4	-356.961
x_2^4	-120.300
x_1^5	-457.228
x_2^5	540.862

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

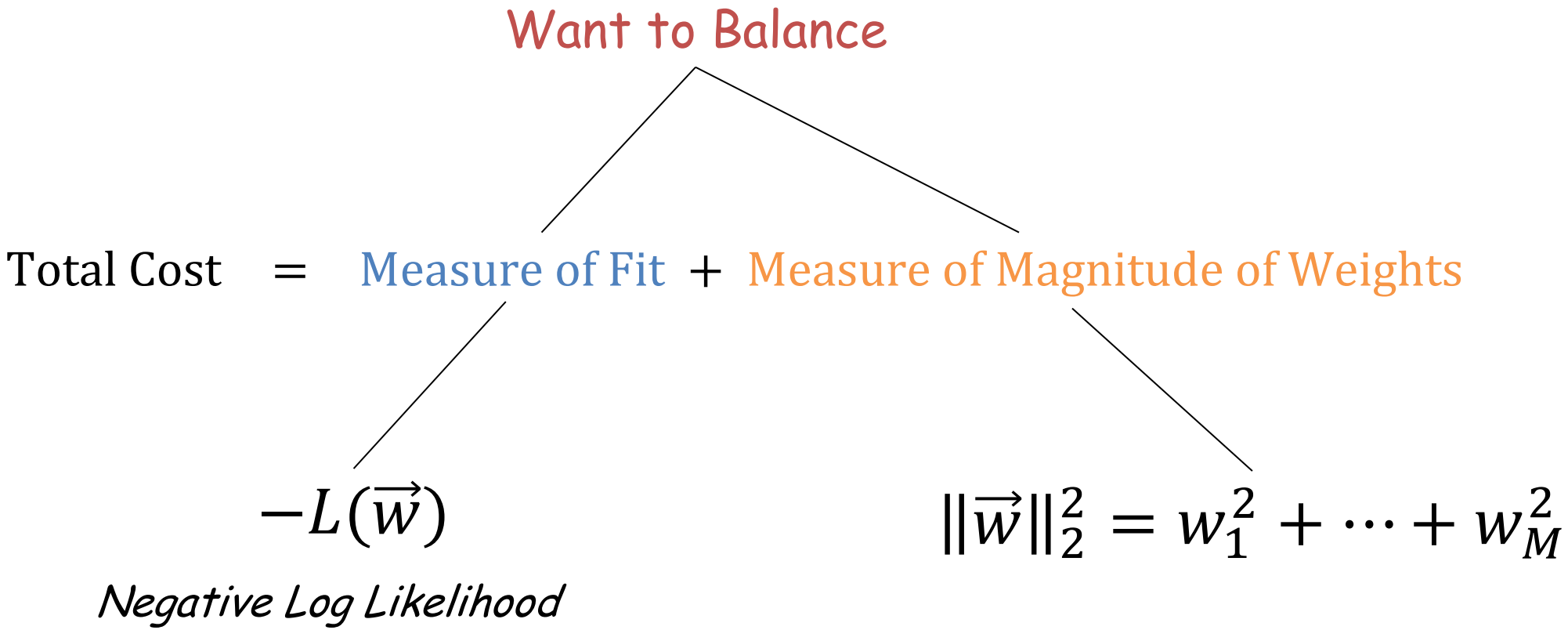
Overfitting → Large Weights



Workflow: Logistic Regression



Desired Total Cost



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)

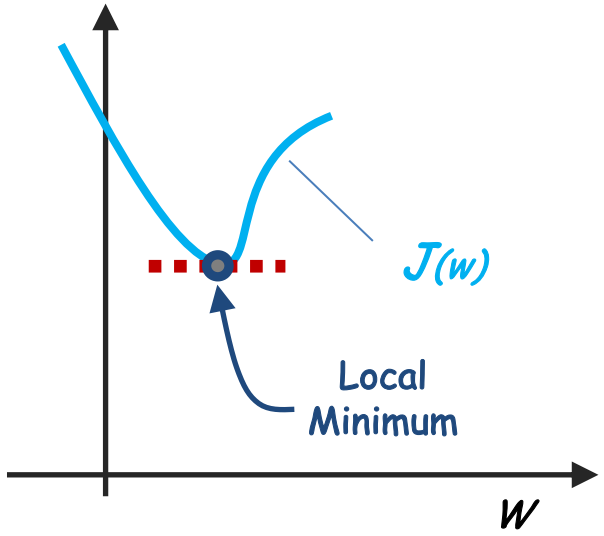
Gradient Vector

Total Cost Function:

$$J(\vec{w}) \quad = \quad -\sum_{i=1}^{N}\left[y^{(i)}\log P(1|\vec{x}^{(i)},\vec{w}) + (1-y^{(i)})\log(1-P\left(1\middle|\vec{x}^{(i)},\vec{w}\right))\right] + \lambda\|\vec{w}\|_2^2$$

Derivative of Total Cost Function:

$$\frac{\partial J(\vec{w})}{\partial w_j} \quad = \quad -\sum_{i=1}^{N}\left(y_i - P(1|\vec{x}^{(i)},\vec{w})\right)x_j^{(i)} + 2\lambda\vec{w}$$



Gradient Vector:

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

Pseudocode for L_2 Regularisation

```
# L2 Regularisation — Plug-in Gradient Term

# Inputs
#  base_grad(w)  ← gradient of the base loss (no regularisation)
#  w0            ← initial parameters
#  n             ← learning rate
#  max_iter      ← maximum iterations
#  tol           ← stop when ||g|| ≤ tol
#  λ ≥ 0         ← L2 strength (intercept w[0] is never penalised)

# ----- helper (exclude intercept) -----
Function REG_VECTOR(w)
  r ← copy(w)
  r[0] ← 0
  Return r
End

# L2 loss piece:   λ · ||REG_VECTOR(w)||²
# L2 grad piece:   2 · λ · REG_VECTOR(w)

# ----- optimise (vanilla GD + L2) -----
w ← w0

FOR t = 1 TO max_iter DO
  g_base ← base_grad(w)              # gradient of base loss
  g_reg  ← 2 · λ · REG_VECTOR(w)         # L2 gradient (no intercept)
  g      ← g_base + g_reg            # total gradient

  IF norm(g) ≤ tol THEN BREAK
  w ← w - n · g
END FOR

Return w
```

Logistic Regression with L_2 Regularisation

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

class MyLogisticRegression(BaseEstimator, ClassifierMixin):
    def __init__(self, eta=1e-3, max_iter=2000, tol=1e-6, lambda_=0.0):
        self.eta, self.max_iter, self.tol = eta, max_iter, tol
        self.lambda_ = lambda_              # λ: L2 strength (0.0 = no regularization)

    def fit(self, X, y):
        Phi = np.c_[np.ones(X.shape[0]), X]   # [1 | X]
        N, D = Phi.shape

        # Gradient of:  sum NLL  +  λ*||w'||^2   (w'[0]=0 ⇒ intercept not penalized)
        def grad(w):
            p = expit(Phi @ w)
            reg = w.copy(); reg[0] = 0.0
            return (Phi.T @ (p - y)) + self.lambda_ * reg

        w0 = np.zeros(D)
        w, n_iter, gn = gradient_descent(
            grad, w0, eta=self.eta, max_iter=self.max_iter, tol=self.tol
        )

        self.weights_, self.n_iter_, self.grad_norm_ = w, n_iter, gn
        return self

    def predict_proba(self, X):
        Xs = np.c_[np.ones(X.shape[0]), X]
        p1 = expit(Xs @ self.weights_)
        return np.column_stack([1 - p1, p1])

    def predict(self, X):
        return (self.predict_proba(X)[:, 1] ≥ 0.5).astype(int)
```

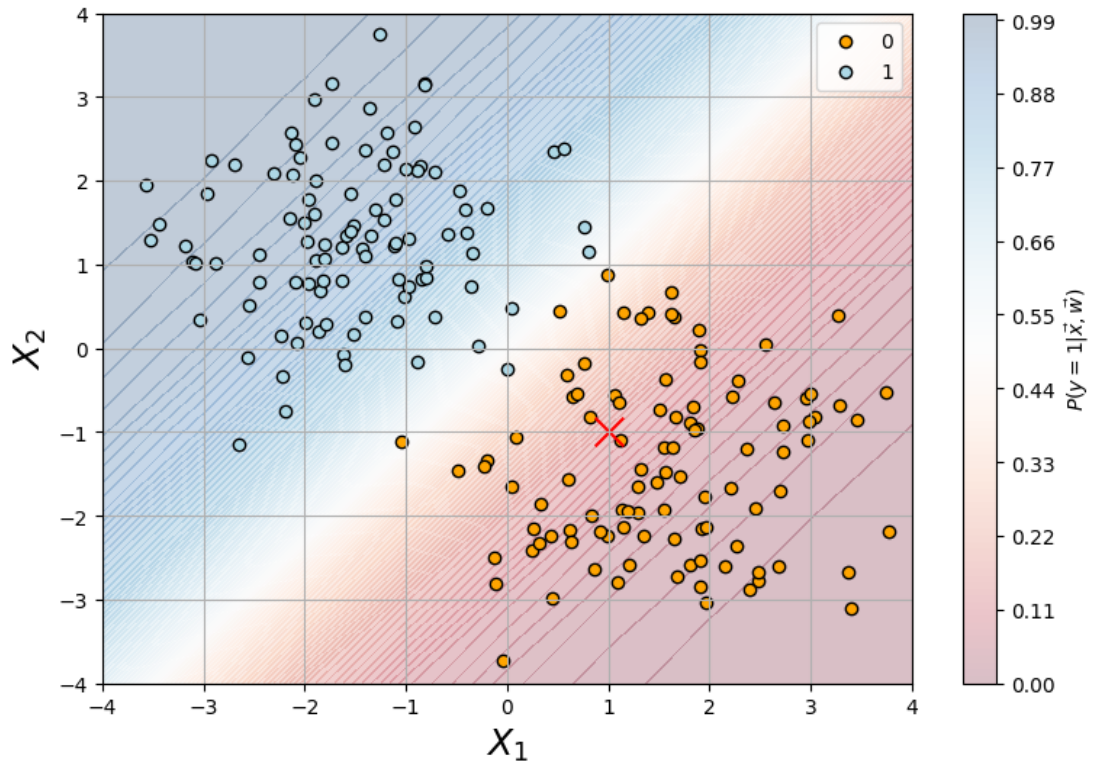
Gradient Descent Function

```
from typing import Callable, Tuple
import numpy as np

def gradient_descent(
    grad: Callable[[np.ndarray], np.ndarray],
    w0: np.ndarray,
    eta: float = 0.05,
    max_iter: int = 1000,
    tol: float = 1e-6,
) → Tuple[np.ndarray, int, float]:
    w = np.asarray(w0, dtype=float).ravel()
    n_iter = 0

    for t in range(max_iter):
        g = grad(w)
        gn = float(np.linalg.norm(g))
        if gn ⩽ tol:
            n_iter = t
            break
        w = w - eta * g
        n_iter = t + 1
    else:
        # loop exhausted without break → recompute final grad norm
        gn = float(np.linalg.norm(grad(w)))

    return w, n_iter, gn
```

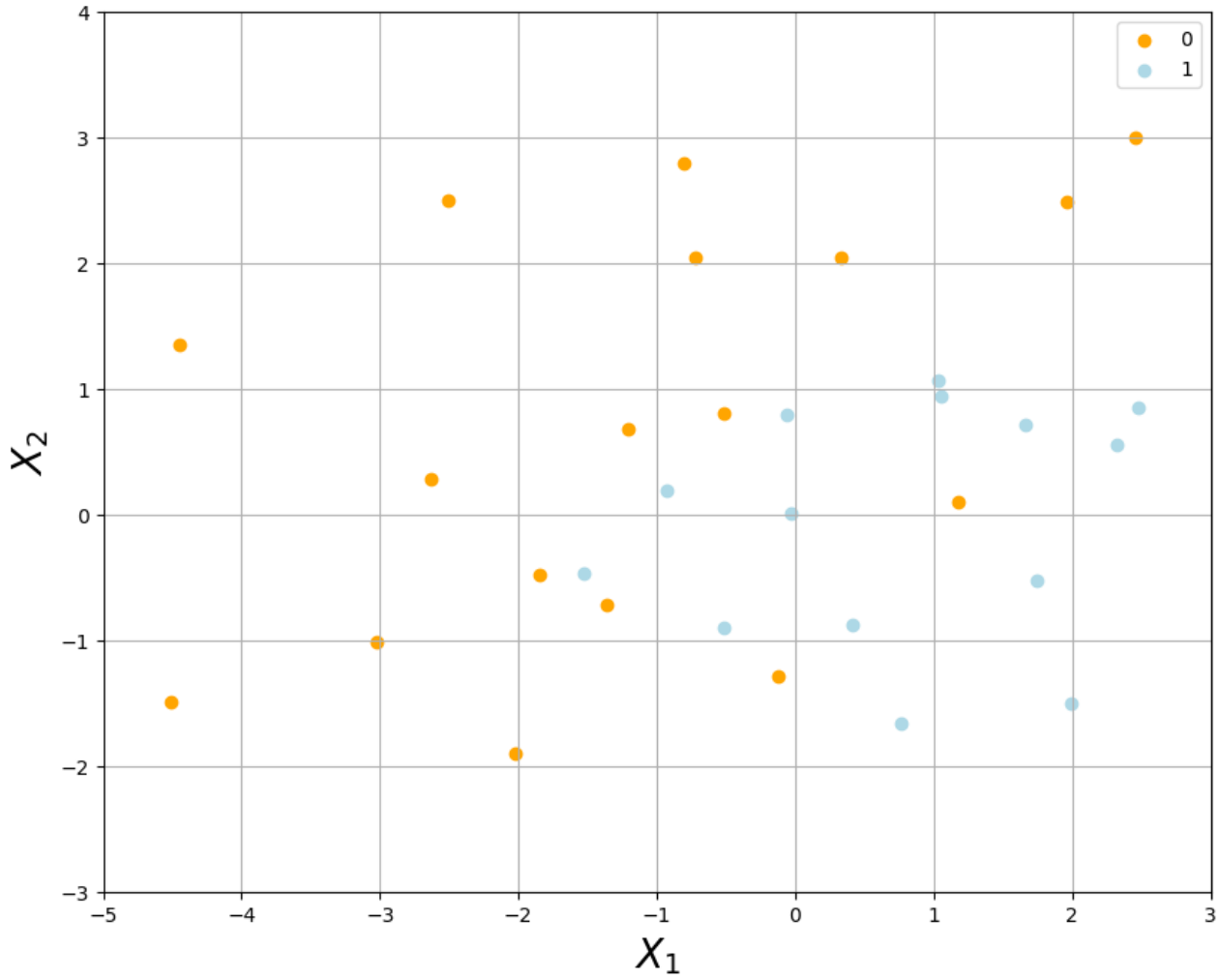
Usage Example

```
import numpy as np
# from your_module import MyLogisticRegression  # class already defined

# ----- data -----
m, n = 100, 2
np.random.seed(0)
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)))
X = np.vstack((class_0, class_1))                          # (2m, 2)
y = np.concatenate([np.zeros(m), np.ones(m)])              # (2m,) in {0,1}

# test point (no bias term; model adds it)
X_test = np.array([[1.0, -1.0]])

# ----- train with L2 only (use default eta/max_iter/tol) -----
lambda_ = 0.1
model = MyLogisticRegression(lambda_=lambda_)  # default GD param
model.fit(X, y)

# ----- predict -----
prob = model.predict_proba(X_test)[0, 1]

print(f"Predicted probability for new point: {prob:.4f}"
```

```
Predicted probability for new point: [0.1415866]
```

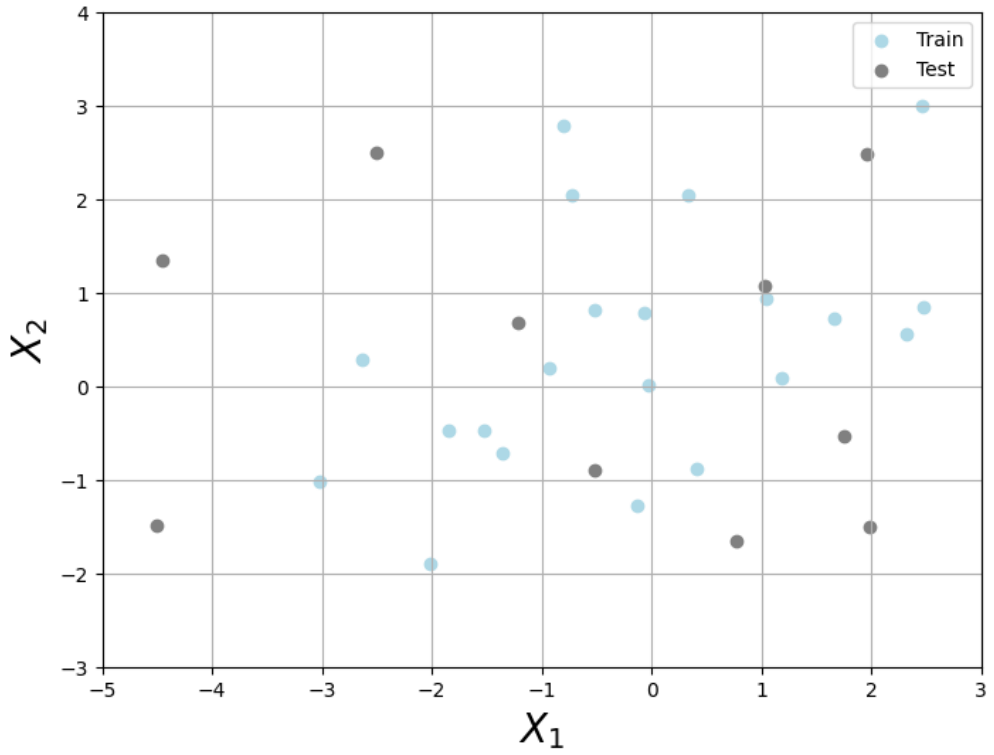


Example Dataset



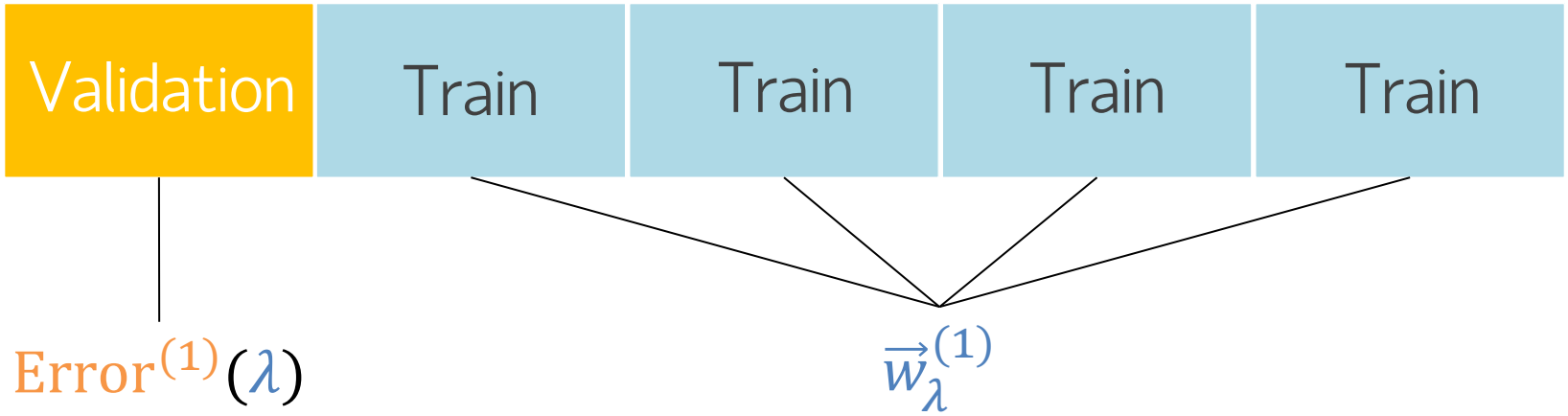
Train, Validation and Test Datasets

```
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 typicaly 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 λ based on the average performance across folds.

K-Fold Cross Validation

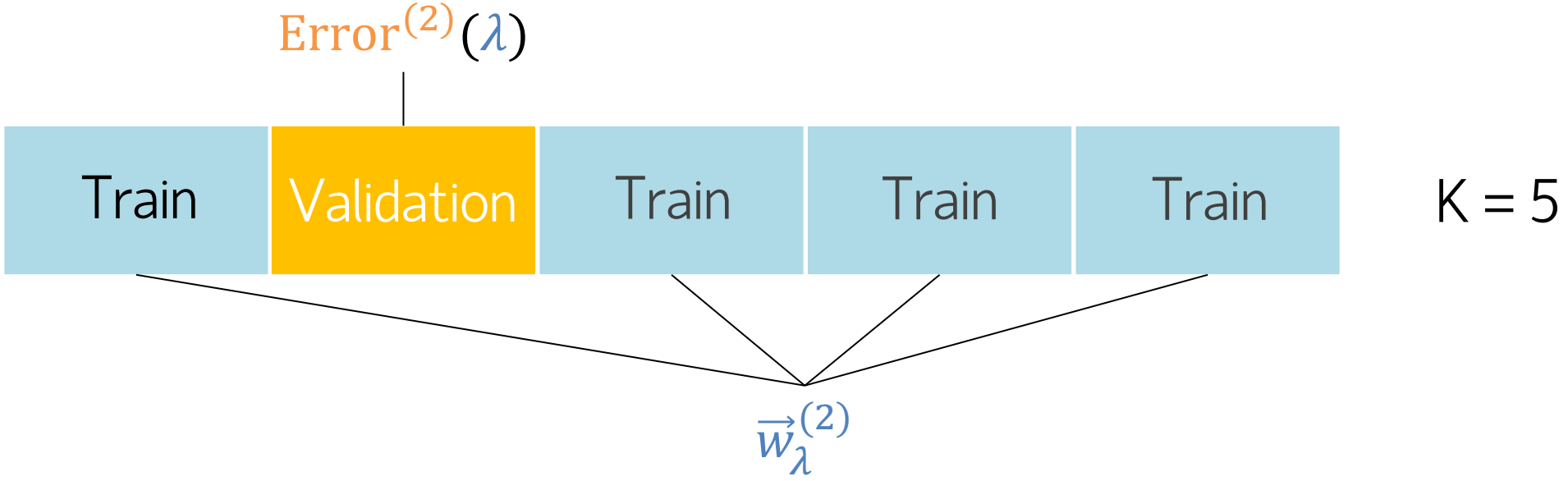


For k = 1, …, K

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

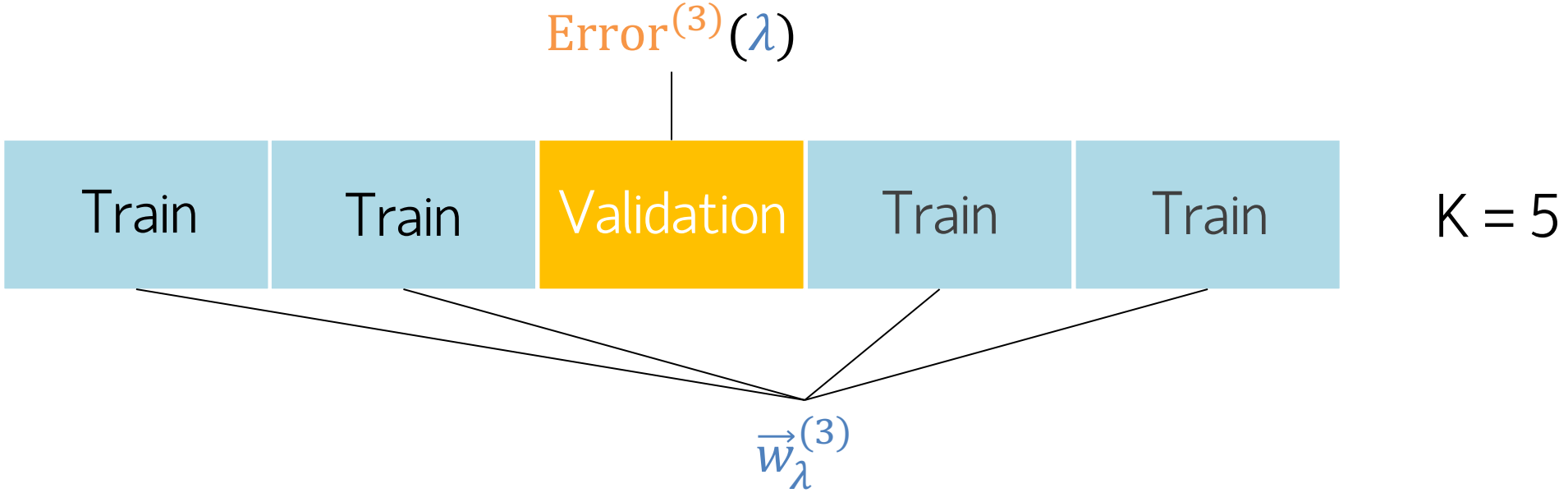


For k = 1, ..., K

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

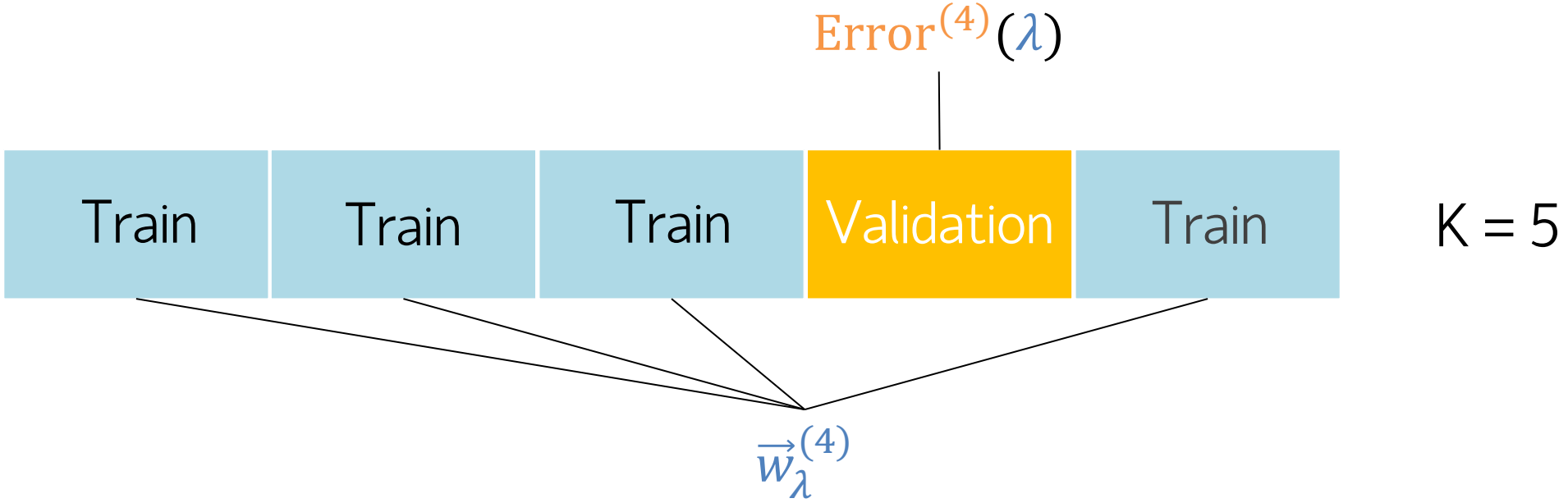


For k = 1, ..., K

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

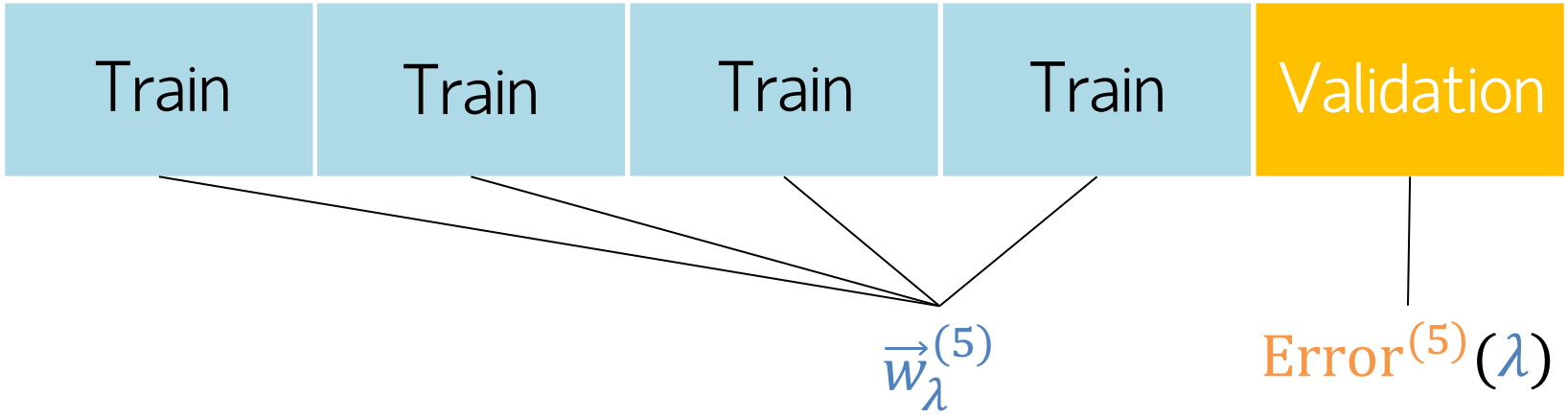


For k = 1, ..., K

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, …, K

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

Step 2: Compute error on Validation Block: $\text{Error}^{(1)}(\lambda)$.

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

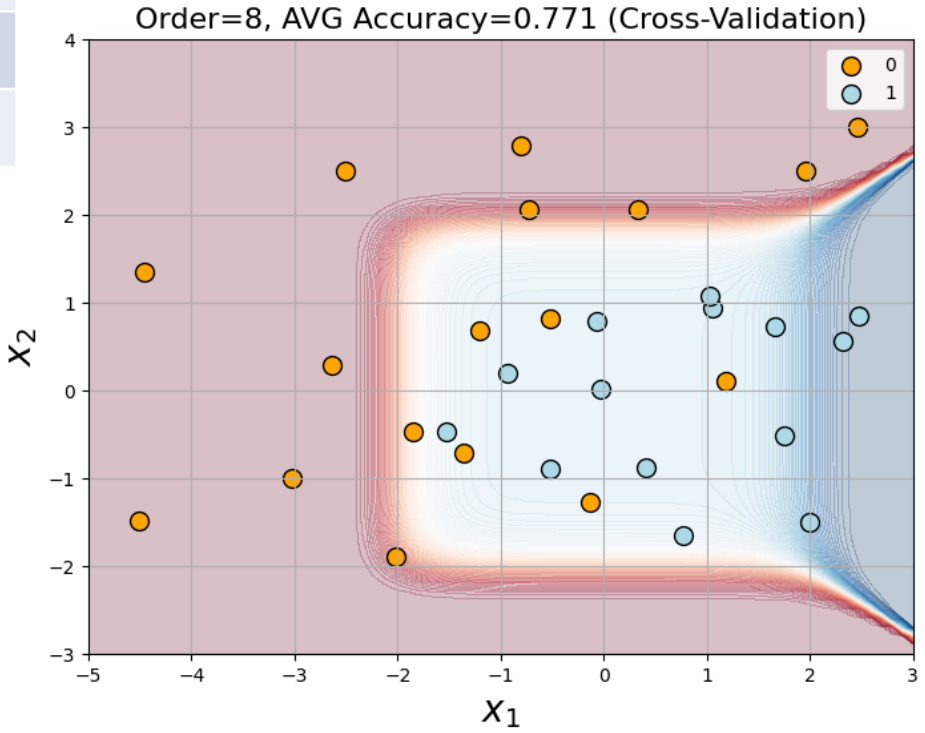
```
import numpy as np
from sklearn.linear_model import LogisticRegression
from sklearn.model_selection import KFold, cross_val_score

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

# Step 2: Define cross-validation strategy
k = 5
kf = KFold(n_splits=k, shuffle=True, random_state=42)

# Step 3: 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)))
# optional: print per-fold scores
# print("Per-fold:", cv_scores)
```


```
Mean Cross-Validation Accuracy: 0.77
```

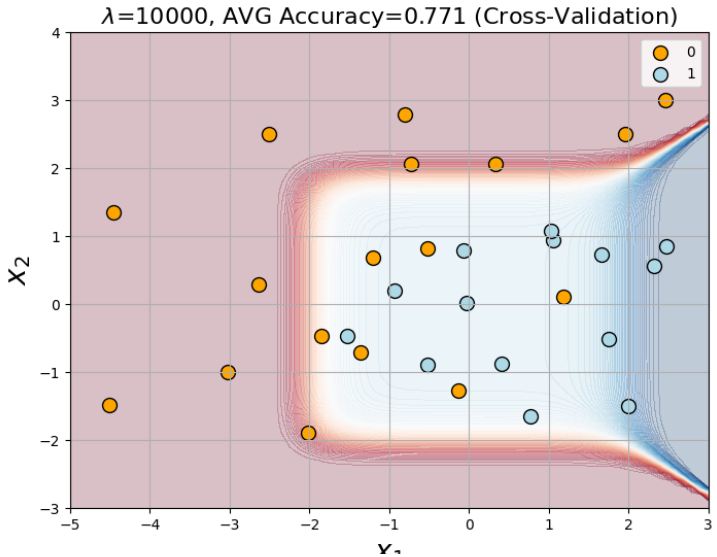
Regularised Decision Boundary (8^{th} Order)

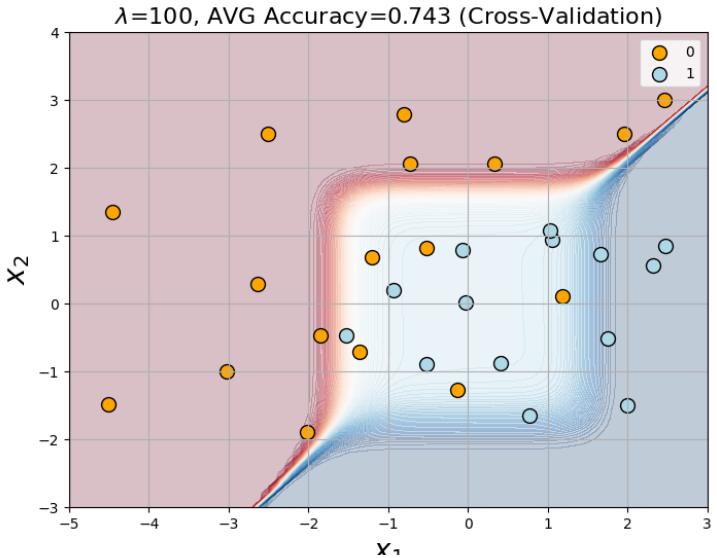
Feature	Coefficient
1	0.609
x_1	3.661e-4
x_2	-3.319e-5
x_1^2	-6.429e-5
x_2^2	-2.815e-5
x_1^3	8.919e-4
x_2^3	-1.843e-4
x_1^4	-1.316e-4
x_2^4	-2.531e-4
x_1^5	2.826e-3
x_2^5	-8.357e-4

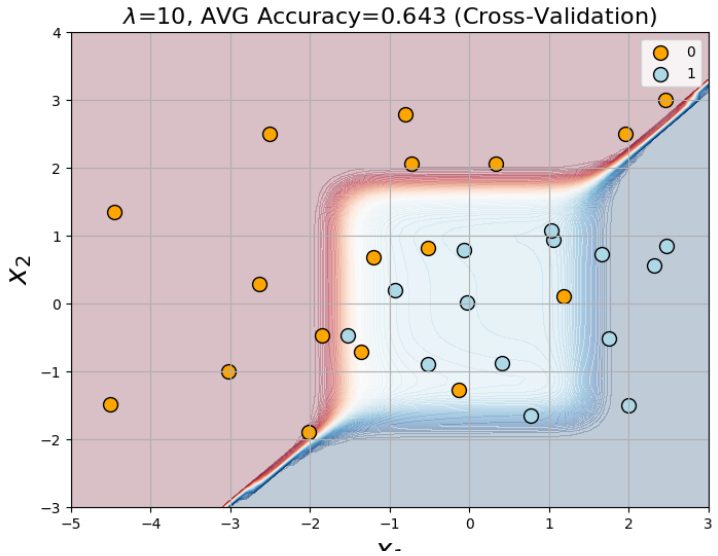
Feature	Coefficient
x_1^6	-2.408e-4
x_2^6	-1.386e-3
x_1^7	9.870e-3
x_2^7	-3.375e-3
x_1^8	-3.631e-4
x_2^8	-6.624e-3

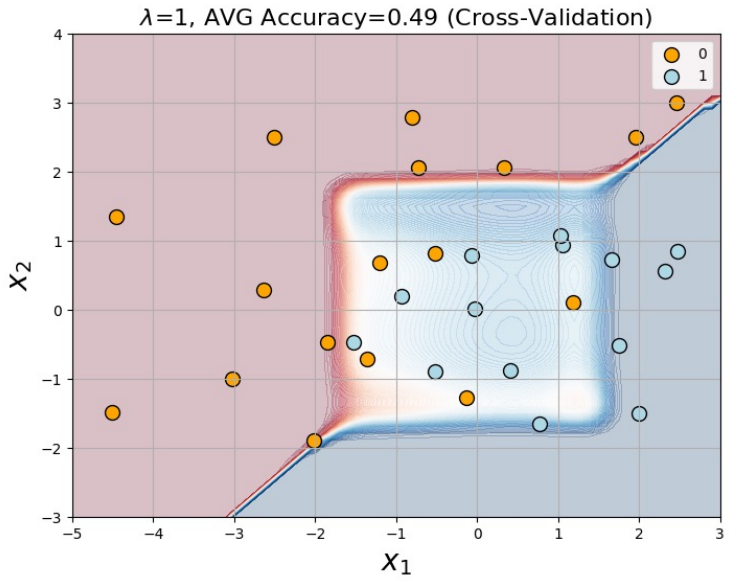


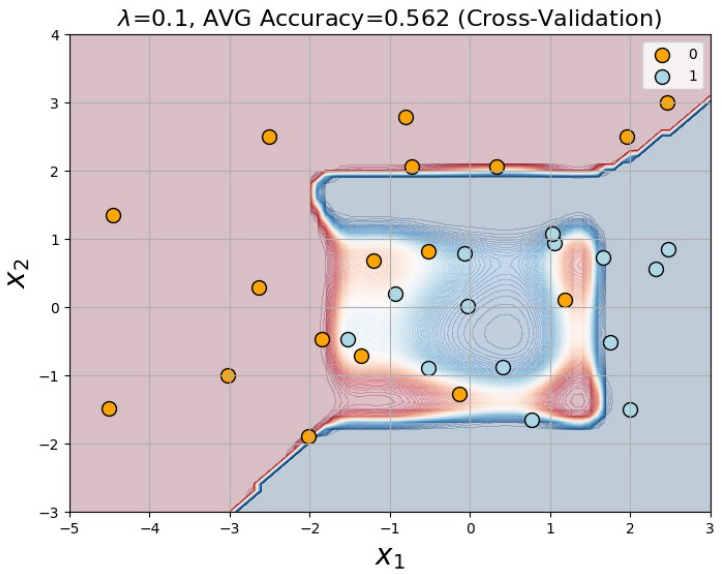
Model Complexity

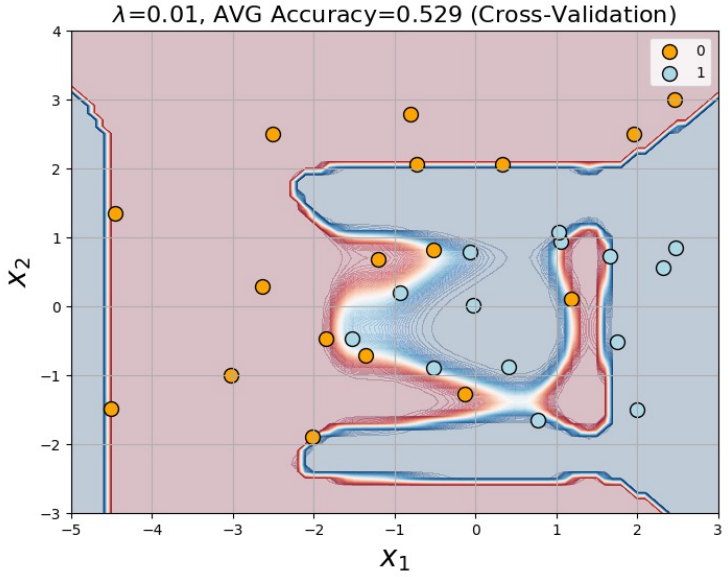




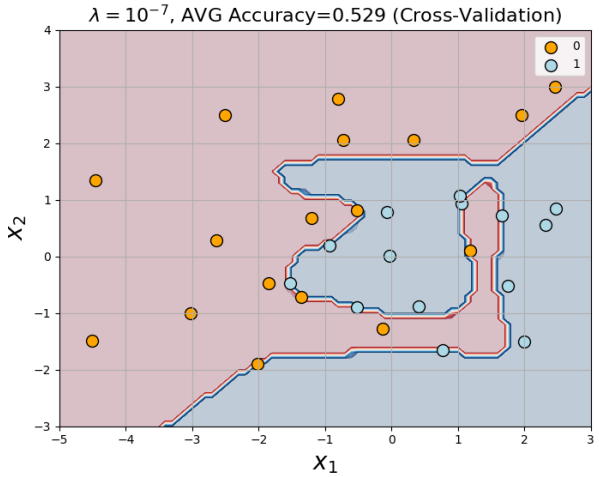




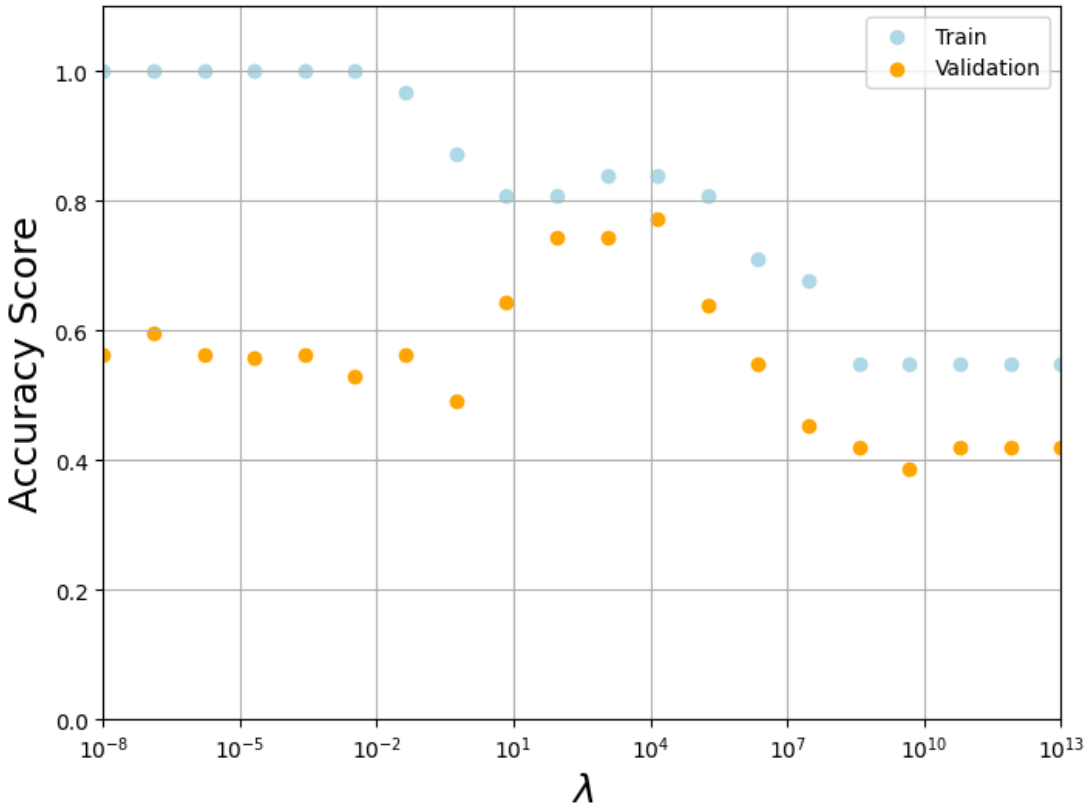


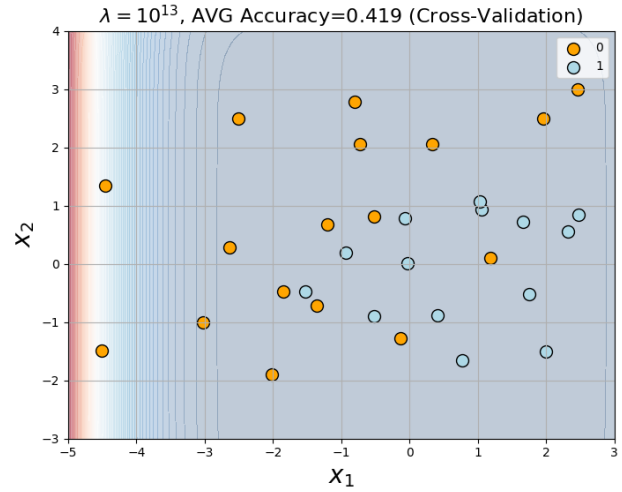


Bias-Variance Trade-Offs



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



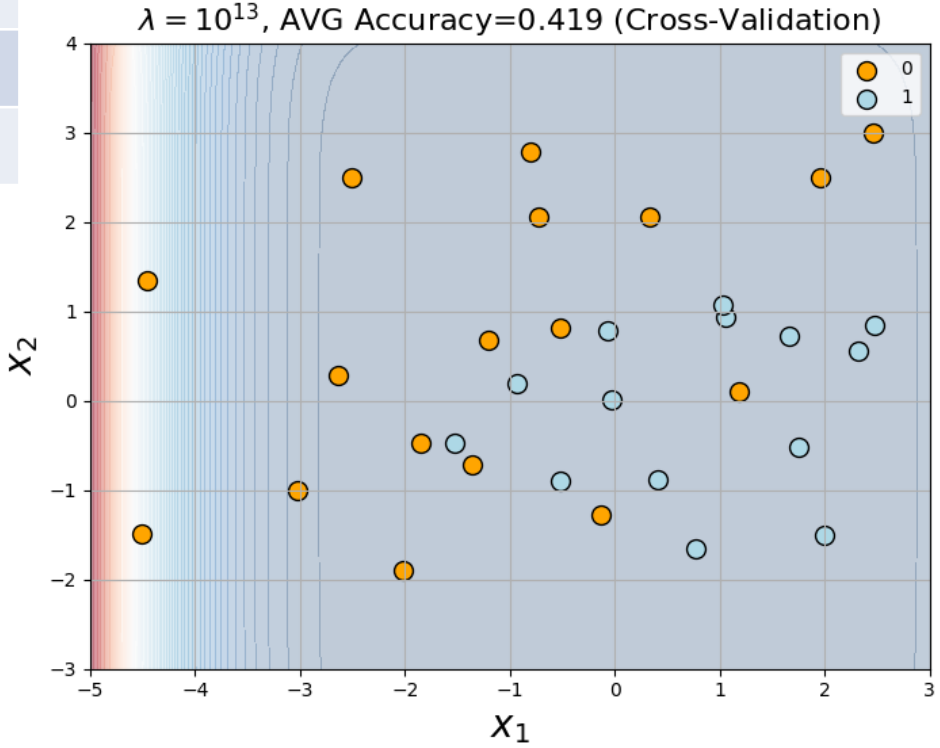


- High Bias, Low Variance
- Simple Decision Boundary
- Low Train Accuracy
- Low Validation Accuracy

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

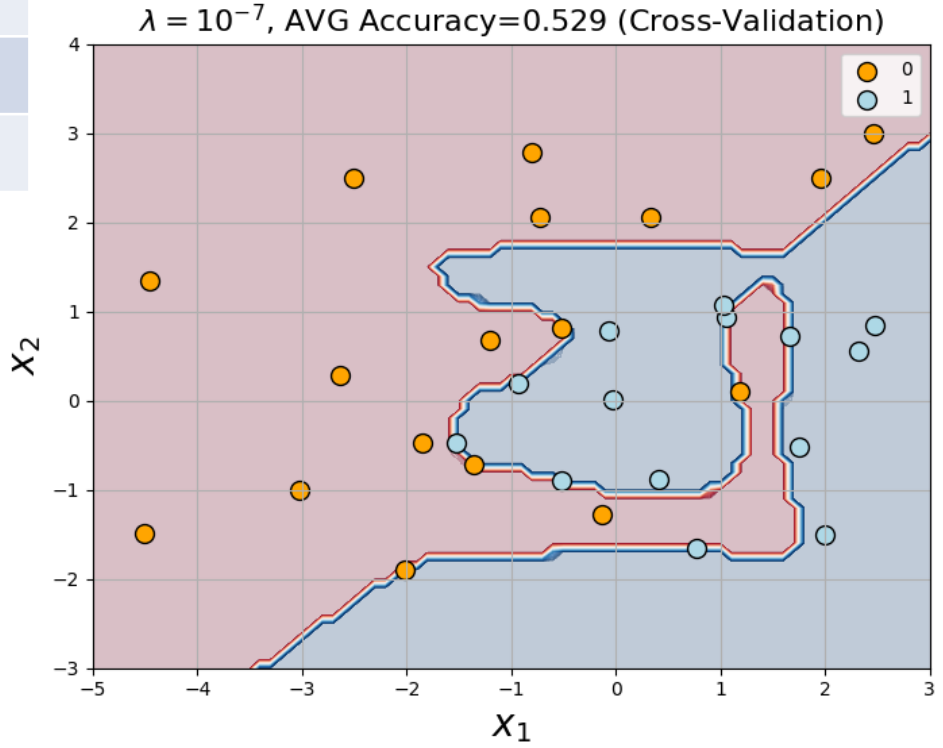
Feature	Coefficient
x_1^6	-7.800e-10
x_2^6	-8.390e-11
x_1^7	3.483e-9
x_2^7	-2.244e-10
x_1^8	1.508e-8
x_2^8	-6.308e-10



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

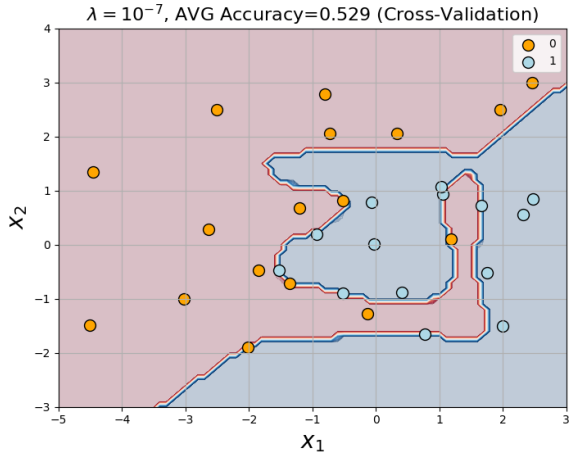
Feature	Coefficient
1	432.548
x_1	333.613
x_2	-352.095
x_1^2	-61.796
x_2^2	-256.131
x_1^3	-36.690
x_2^3	219.2734
x_1^4	-246.131
x_2^4	-106.567
x_1^5	-310.860
x_2^5	405.405

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

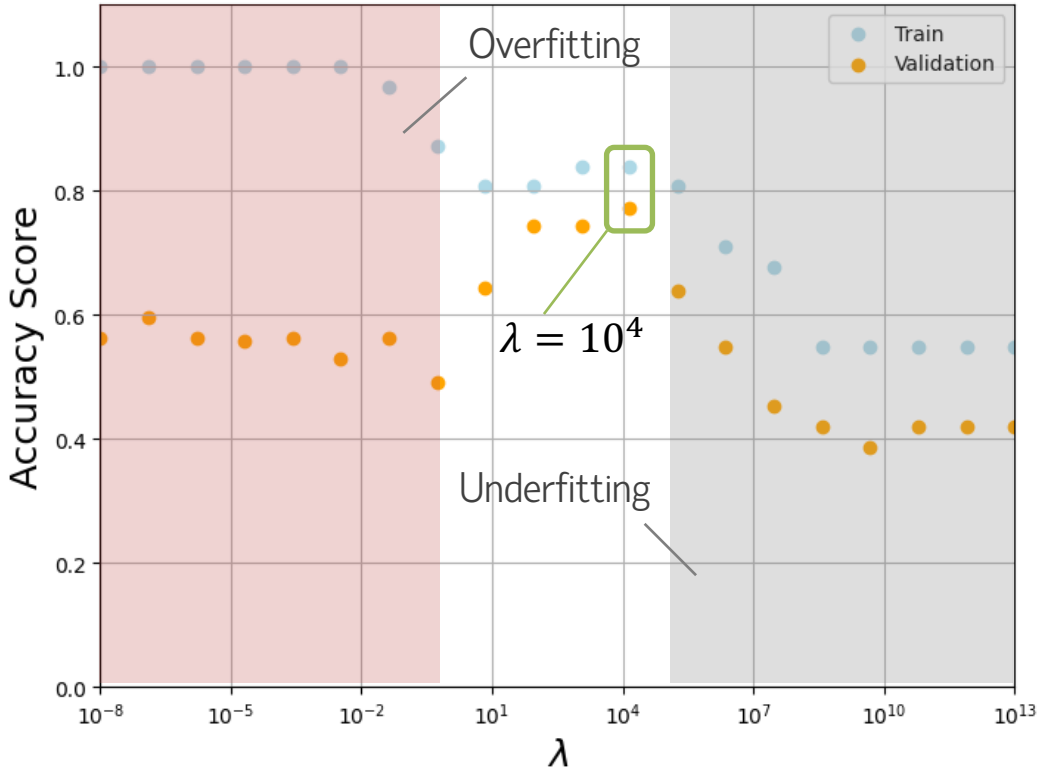


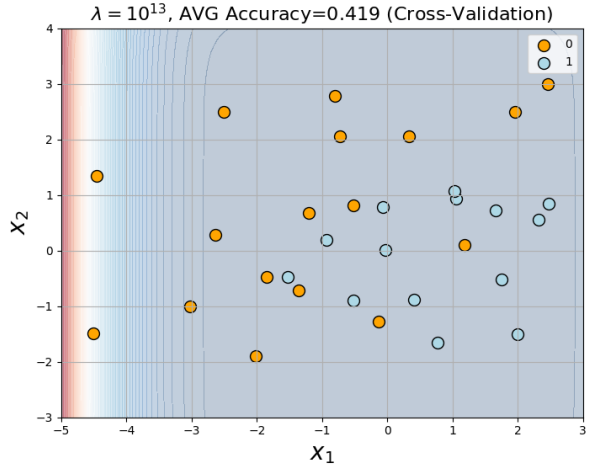
Overfitting vs Underfitting

- **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





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

- **Optimal:** The ideal λ 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.

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, λ, governs the strength of the penalty. A smaller λ may result in high variance and overfitting, while a larger λ increases bias and risks underfitting. Tuning λ 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.