Machine Learning

Logistic Regression

Tarapong Sreenuch

8 February 2024

克明峻德，格物致知

Service records show year-3 car owners going quiet. The clocks tick past the warranty period.

The budget can't cover everyone. We focus on the likely leavers and send a "stay" bundle voucher.

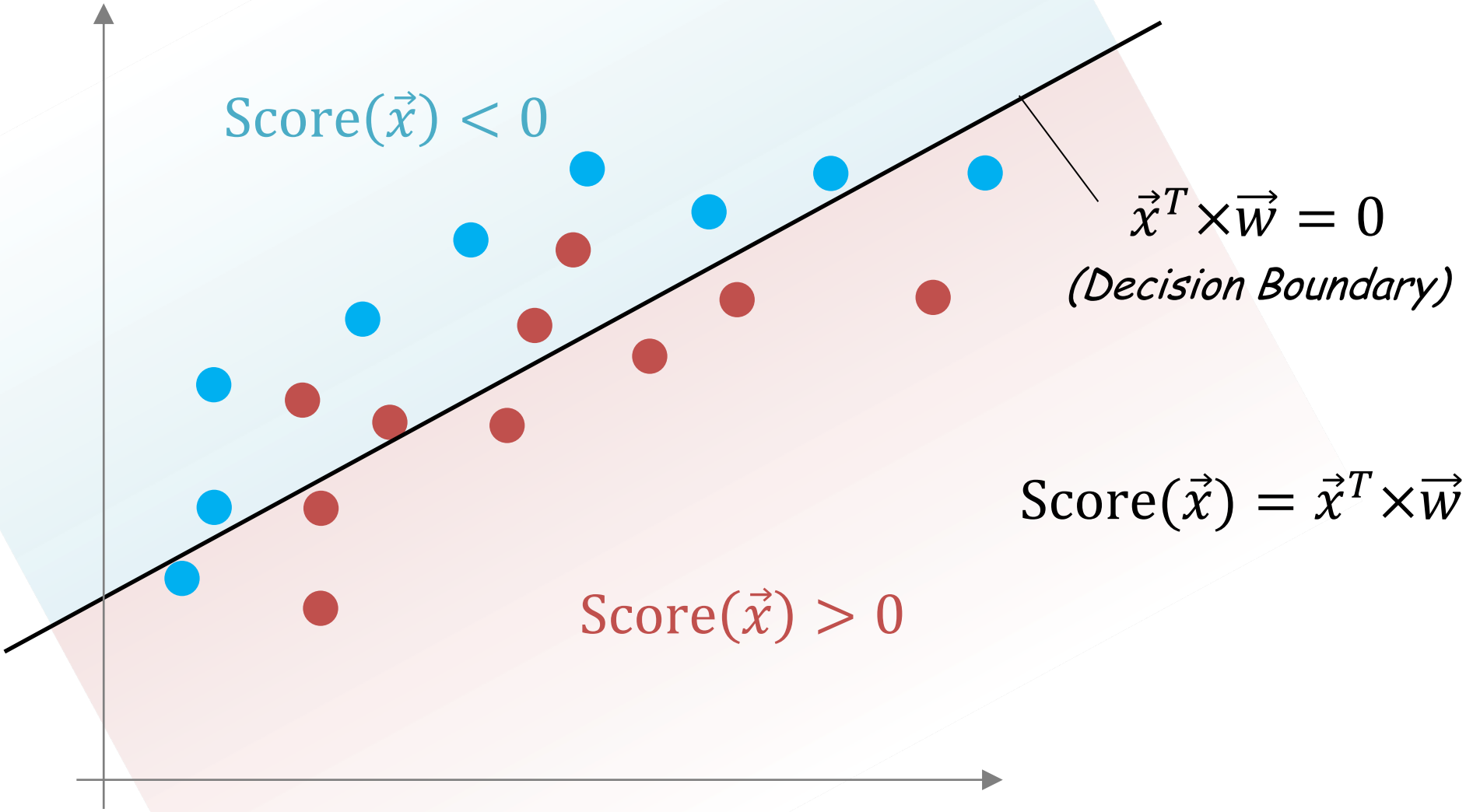
We maximise our ROI with we have.

Q: How is this like for an ML classifier?

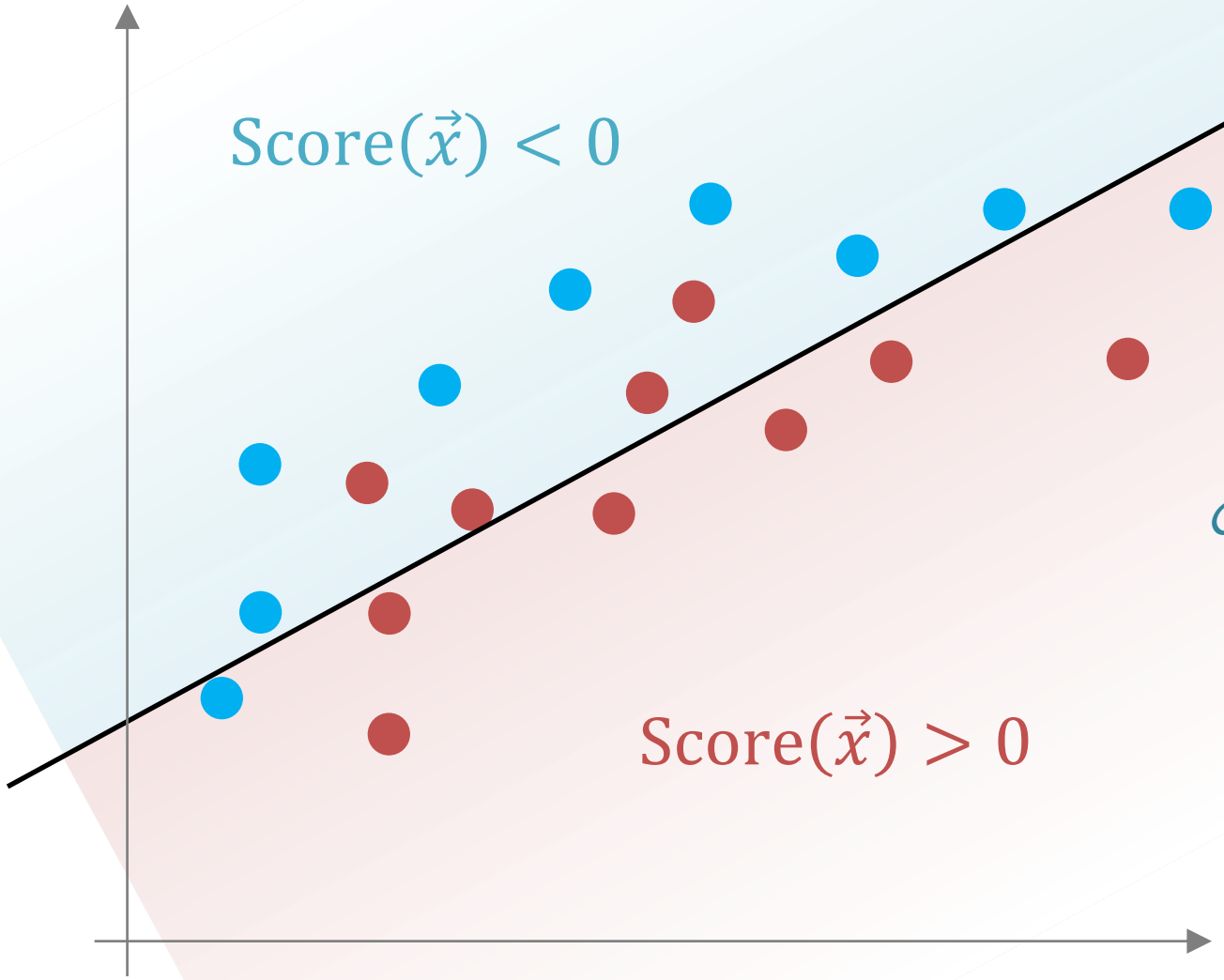
A: Logistic Regression outputs a likelihood, instead of a hard yes-or-no decision.

Not all mistakes cost the same.

Linear Classifier



Recap: Accuracy



$$\vec{x}^T \times \vec{w} = 0$$

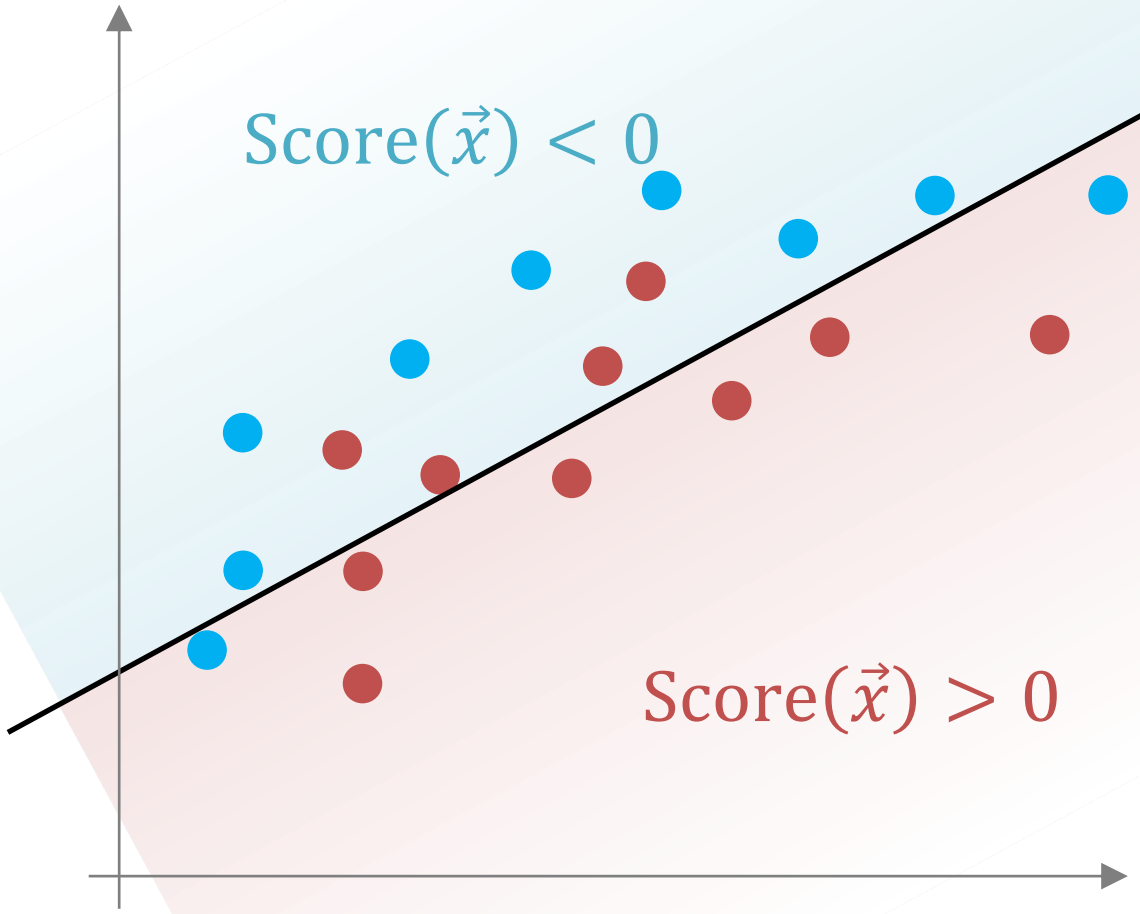
(Decision Boundary)

Q: How many did we classify it correctly?

$$\text{Accuracy} = \frac{6+7}{19}$$

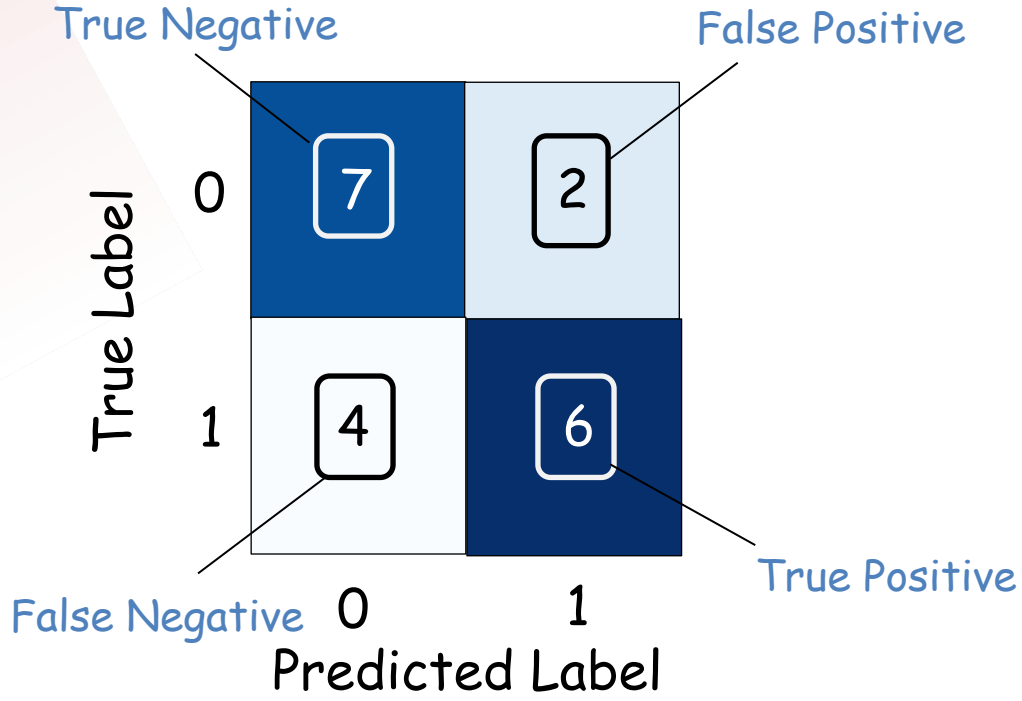
$$\approx 0.68$$

Recap: Confusion Matrix



$\vec{x}^T \times \vec{w} = 0$

(Decision Boundary)



Prediction Confidence

The ramen & everything else were awesome!

Definitely Positive

$$P\left(y = 1 \,\middle|\, \vec{x} = \text{"The ramen \& everything else were awesome!"}\right) = 0.99$$

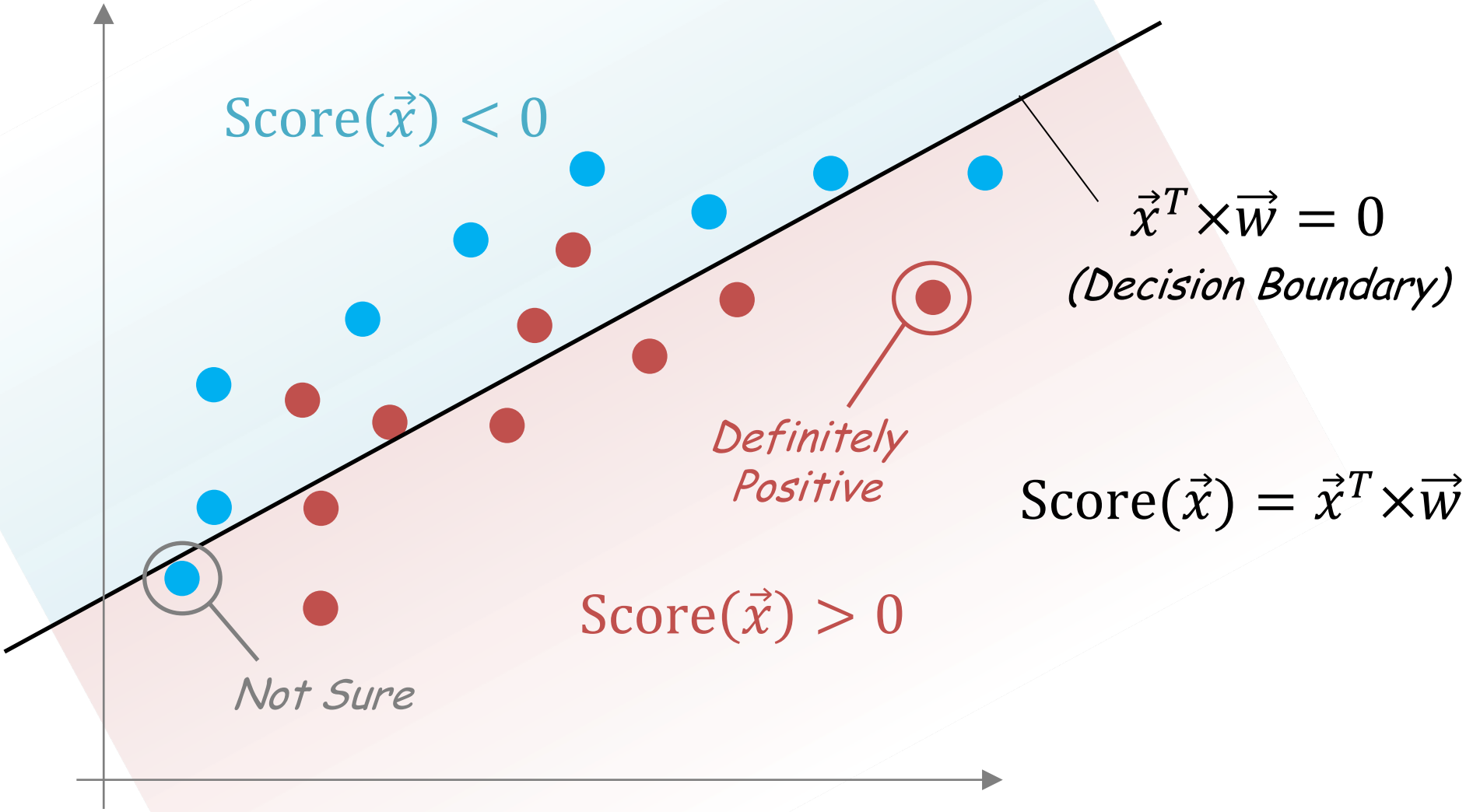
The ramen was good, the service was OK.

Not Sure

$$P\left(y = 1 \,\middle|\, \vec{x} = \text{"The ramen was good, the service was OK."}\right) = 0.55$$

We read the conditional probability $P(y|\vec{x})$ as "the probability of y given $\vec{x}$".

How Confidence?



Scoring Interpretation

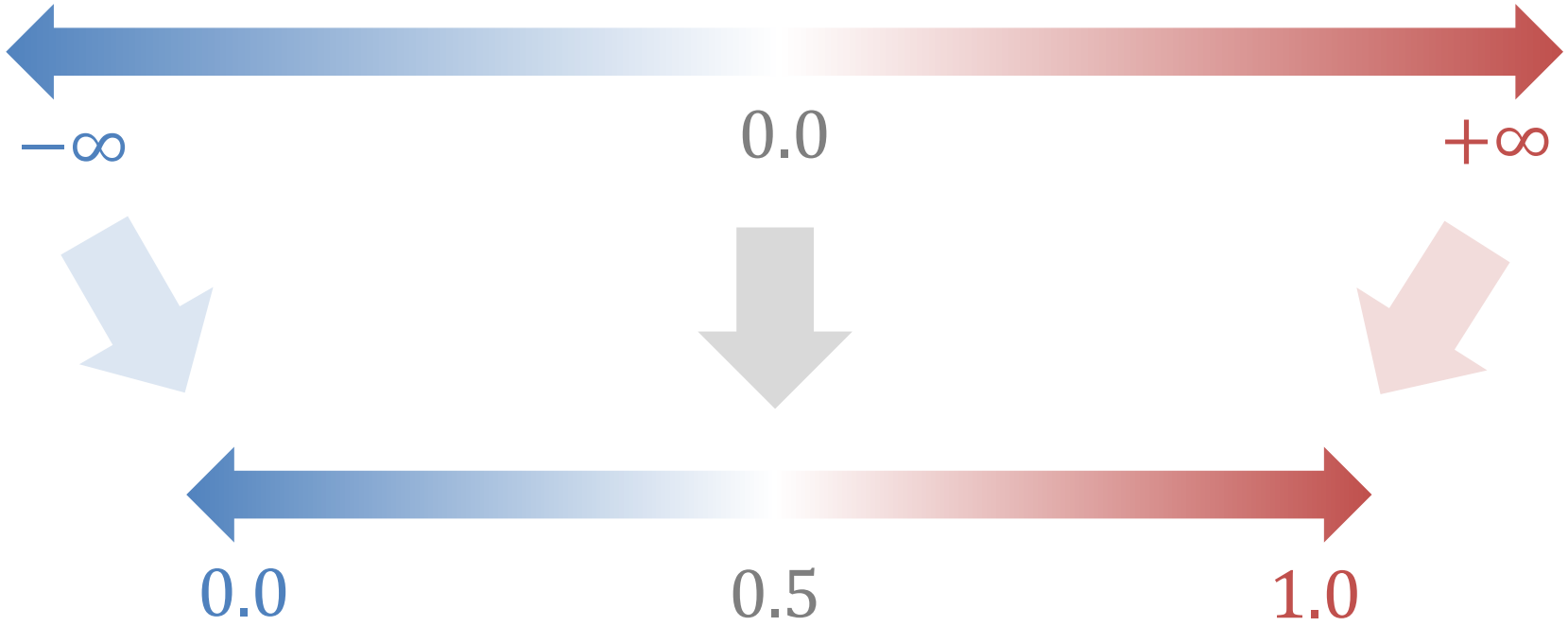
$$\text{Score}(\vec{x}_i) = \vec{x}_i^T \times \vec{w}$$

$-\infty$ $\quad \hat{y}_i = 0 \quad$ 0.0 $\quad \hat{y}_i = 1 \quad$ $+\infty$

Very Sure	*Not Sure If*	*Very Sure*
$\hat{y}_i = 0$	$\hat{y}_i = 0 \; or \; 1$	$\hat{y}_i = 1$
$P(y = 1 \mid \vec{x}_i, \vec{w}) = 0$	$P(y = 1 \mid \vec{x}_i, \vec{w}) = 0.5$	$P(y = 1 \mid \vec{x}_i, \vec{w}) = 1$

Scoring Interpretation

$$\text{Score}(\vec{x}_i) = \vec{x}_i^T \times \vec{w}$$



$$P(y = 1 \mid \vec{x}_i)$$

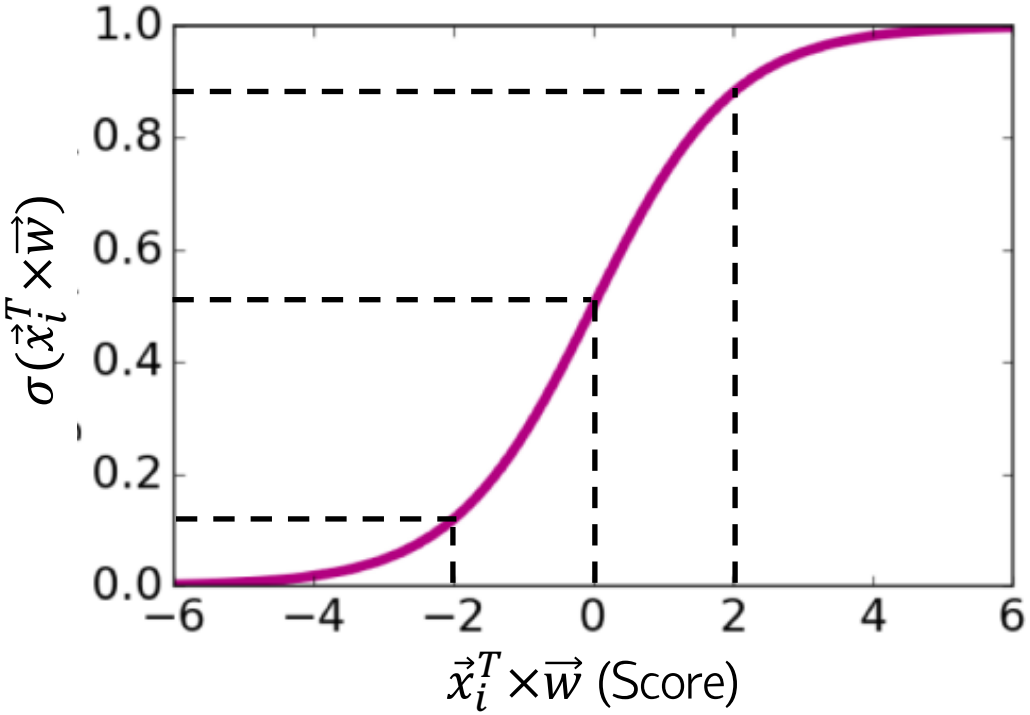
Q: How do we map (-∞, +∞) to [0.0, 1.0]?

Sigmoid (or Logistic) Function

Sigmoid Function:

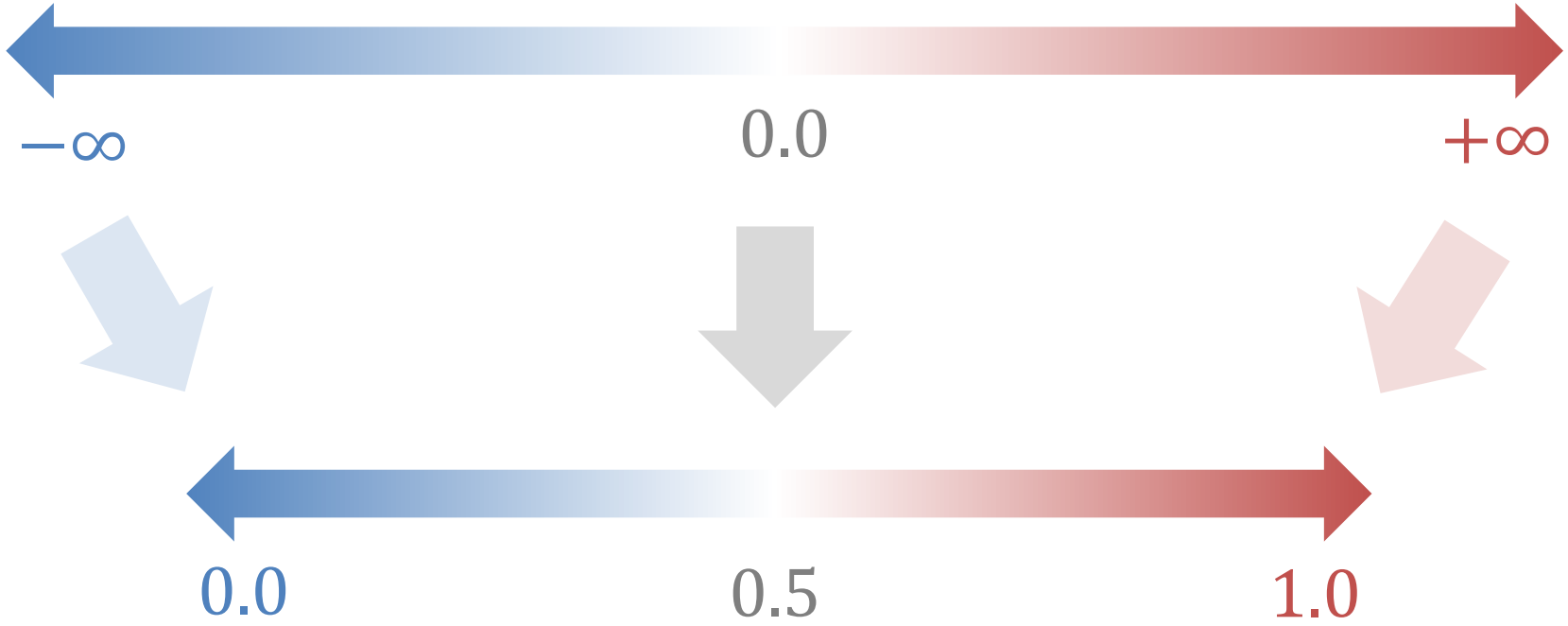
$$\sigma(\vec{x}_i^T \times \vec{w}) = \frac{1}{1 + e^{-\vec{x}_i^T \times \vec{w}}}$$

$\vec{x}_i^T \times \vec{w}$ (Score)	$\sigma(\vec{x}_i^T \times \vec{w})$
$-\infty$	0.0
-2.0	0.12
0.0	0.5
$+2.0$	0.88
$+\infty$	1.0



Logistic Regression Model

$$\text{Score}(\vec{x}_i) = \vec{x}_i^T \times \vec{w}$$



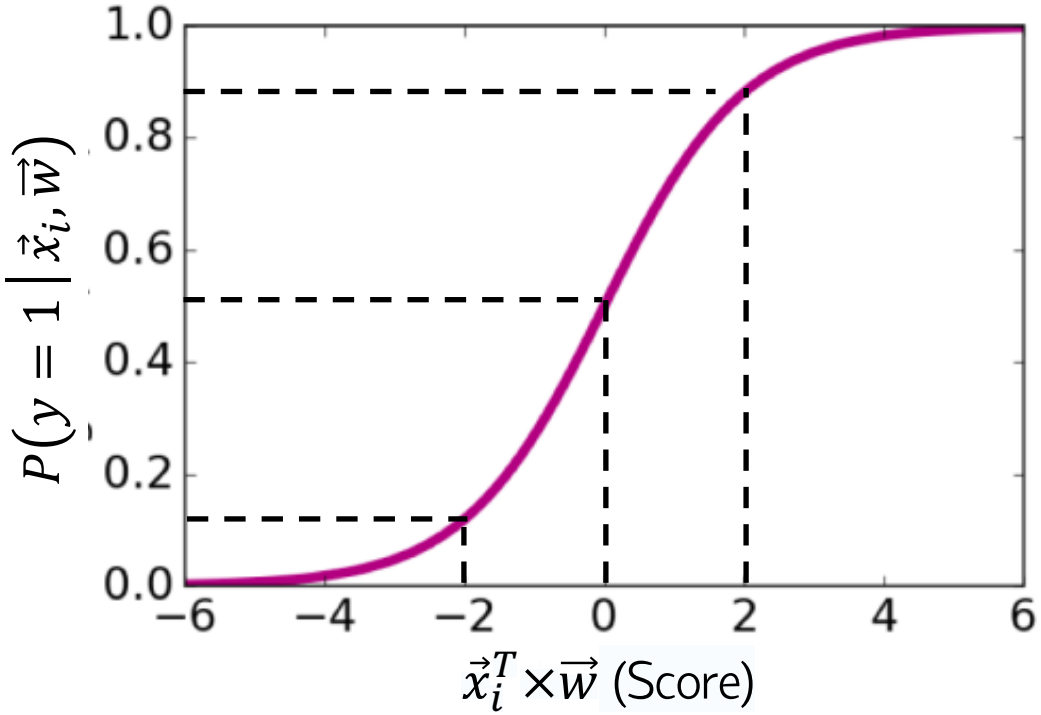
$$P\left(y = 1 \,\middle|\, \vec{x}_i, \vec{w}\right) = \frac{1}{1+e^{-\vec{x}_i^T \times \vec{w}}}$$

Understand Logistic Regression

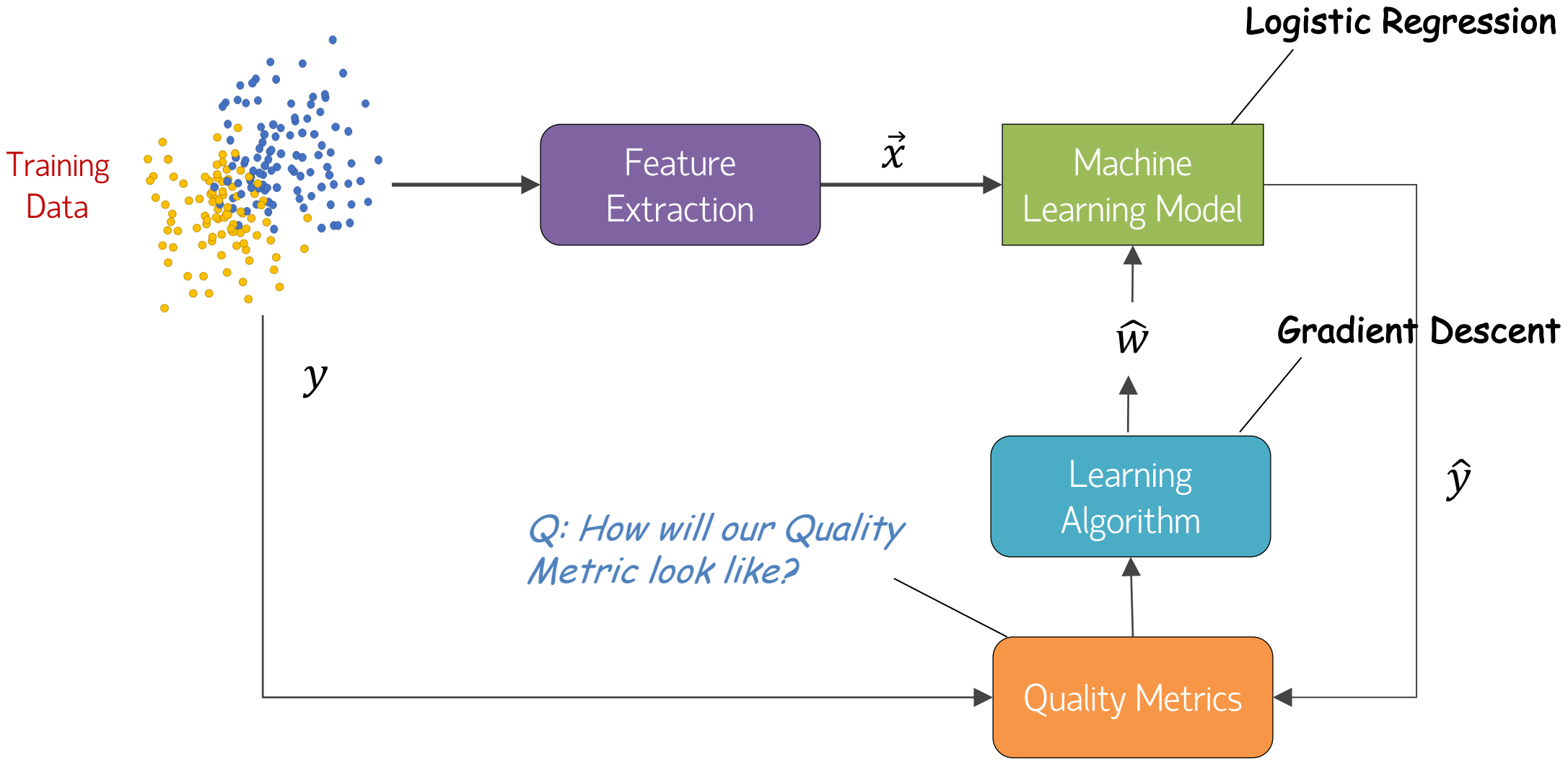
Probabilistic Score:

$$P\left(y=1 \mid \vec{x}_i, \vec{w}\right) = \frac{1}{1+e^{-\vec{x}_i^T \times \vec{w}}}$$

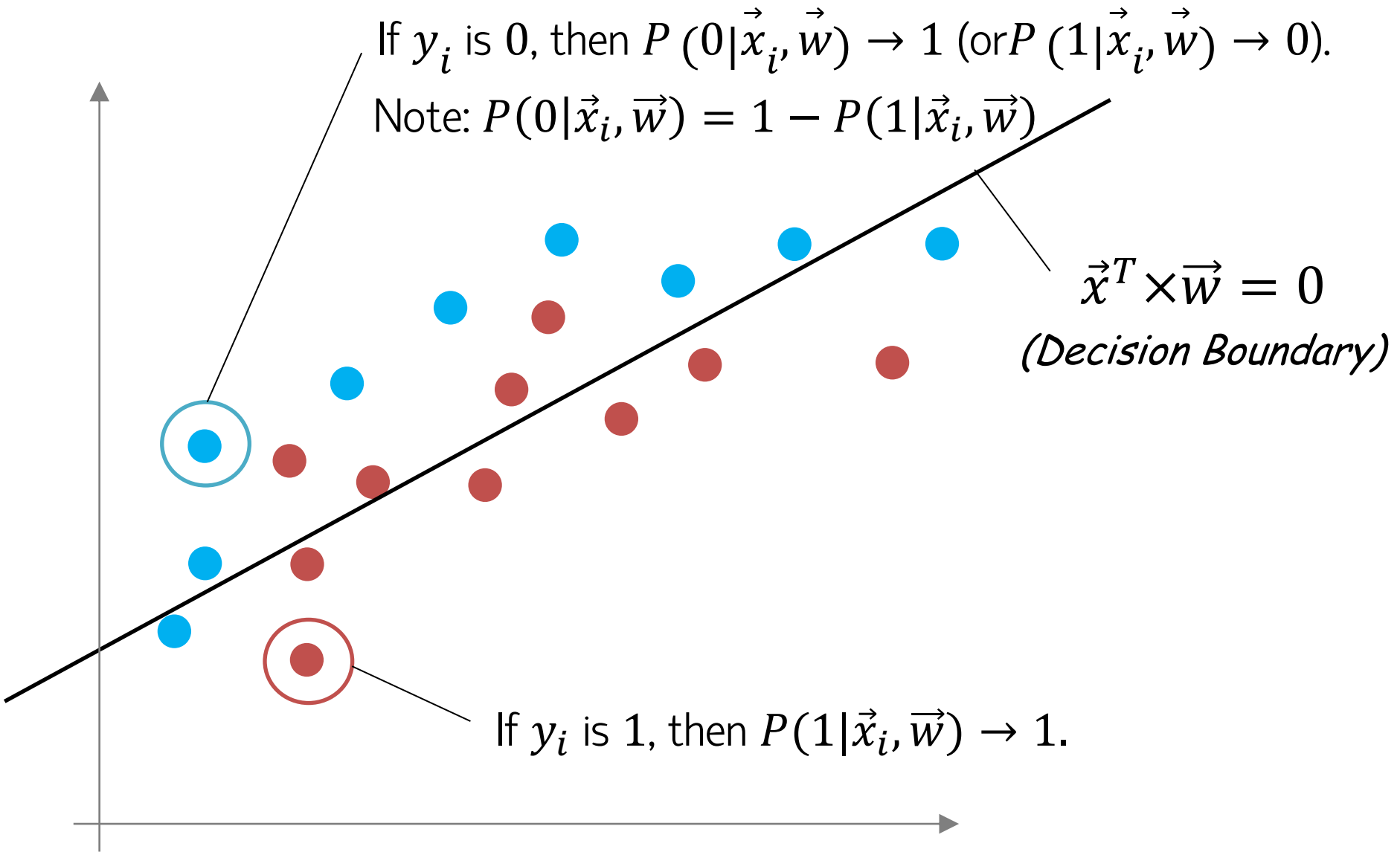
$\vec{x}_i^T \times \vec{w}$ (Score)	$P\left(y=1 \mid \vec{x}_i, \vec{w}\right)$
$-\infty$	0.0
-2.0	0.12
0.0	0.5
$+2.0$	0.88
$+\infty$	1.0



Workflow: Logistic Regression



What We Are Optimising



Recall: Joint Probability

Q: What is the probability (or likelihood) of rolling a six on both dice?

A: $\frac{1}{6} \times \frac{1}{6} = \frac{1}{36}$

$$P(A,B) = P(A) \times P(B)$$

Probability of Rolling a Six on Dice A

Probability of Rolling a Six on Dice B

Quality Metric: Likelihood Function

Likelihood Function:

If y_i is 1, then $P(1|\vec{x}_i, \vec{w}) \rightarrow 1$.

$$L(\vec{w}) = \prod_{i=1}^{N} P(1|\vec{x}_i, \vec{w})^{y_i} \times [1 - P(1|\vec{x}_i, \vec{w})]^{(1-y_i)}$$

$= 1$ if $y_i = 0$

$= 1$ if $y_i = 1$

If y_i is 0, then $P(1|\vec{x}_i, \vec{w}) \rightarrow 0$ (or $P(0|\vec{x}_i, \vec{w}) \rightarrow 1$).

Note: $P(0|\vec{x}_i, \vec{w}) = 1 - P(1|\vec{x}_i, \vec{w})$

Likelihood Function as an On/Off Switch

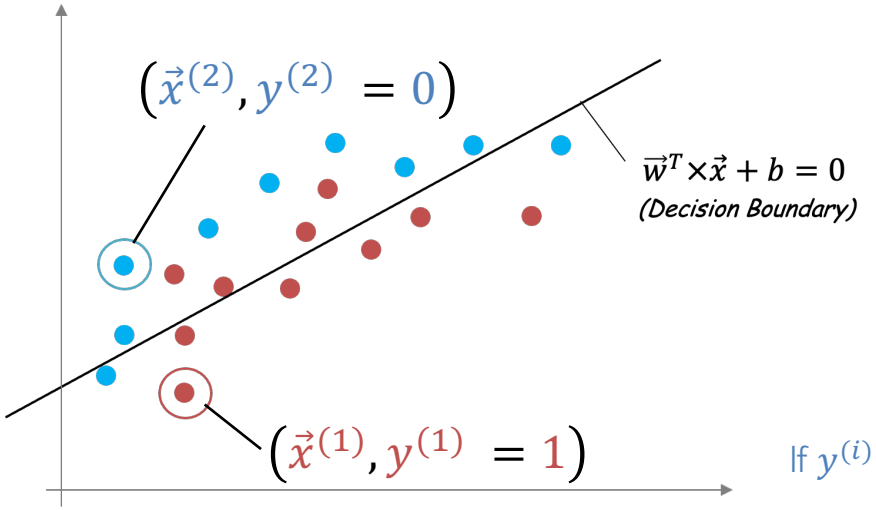
Likelihood Function:

0 (OFF) if $y^{(i)} = 0$ // 0 (OFF) if $y^{(i)} = 1$ //

$$L(\vec{w}, b) = -\sum_{i=1}^{N} \left[y^{(i)} \log P(1|\vec{x}^{(i)}, \vec{w}, b) + (1 - y^{(i)}) \log(1 - P(1|\vec{x}^{(i)}, \vec{w}, b)) \right]$$

Related Properties:

$P(y = 1\|\vec{x}, \vec{w}, b) \in (0,1)$	$P(y = 1\|\vec{x}, \vec{w}, b) \to 1^-$
$\log P(y = 1\|\vec{x}, \vec{w}, b) \in (-\infty, 0)$	$\log P(y = 1\|\vec{x}, \vec{w}, b) \to 0^-$
$-\log P(y = 1\|\vec{x}, \vec{w}, b) \in (0, +\infty)$	$-\log P(y = 1\|\vec{x}, \vec{w}, b) \to 0^+$



If $y^{(i)}$ is 1, then $-\log P(1|\vec{x}^{(i)}, \vec{w}) \to 0^+$.

$$\begin{aligned} L(\vec{x}^{(1)}, \vec{w}, b) &= -\left[y^{(1)} \log P(1|\vec{x}^{(1)}, \vec{w}, b) + (1 - y^{(1)}) \log\left(1 - P(1|\vec{x}^{(1)}, \vec{w}, b)\right) \right] \\ &= -\left[1 \times \log P(1|\vec{x}^{(1)}, \vec{w}, b) + (1 - 1) \log\left(1 - P\left(1|\vec{x}^{(1)}, \vec{w}, b\right)\right)\right] \\ &= -\log P(1|\vec{x}^{(1)}, \vec{w}, b) \end{aligned}$$

$$\begin{aligned} L(\vec{x}^{(2)}, \vec{w}, b) &= -\left[y^{(2)} \log P(1|\vec{x}^{(2)}, \vec{w}, b) + (1 - y^{(2)}) \log\left(1 - P(1|\vec{x}^{(2)}, \vec{w}, b)\right) \right] \\ &= -\left[0 \times \log P(1|\vec{x}^{(2)}, \vec{w}, b) + (1 - 0) \log\left(1 - P(1|\vec{x}^{(2)}, \vec{w}, b)\right)\right] \\ &= -\log\left(1 - P(1|\vec{x}^{(2)}, \vec{w}, b)\right) \\ &= -\log P(0|\vec{x}^{(2)}, \vec{w}, b) \end{aligned}$$

If $y^{(i)}$ is 0, then $-\log\left(1 - P(1|\vec{x}^{(2)}, \vec{w}, b)\right) \to 0^+$ (or $-\log P(0|\vec{x}^{(i)}, \vec{w}, b) \to 0^+$).

Log Likelihood Function

Likelihood Function:

$$L(\vec{w}) = \prod_{i=1}^{N} P(1|\vec{x}_i, \vec{w})^{y_i} \times [1 - P(1|\vec{x}_i, \vec{w})]^{(1-y_i)}$$

Log Likelihood Function:

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

Relevant Logarithmic Rules:

$$\log(A \times B) = \log A + \log B \qquad \log A^N = N \log A$$

Numerical Underflow

$P(1|\vec{x}_i, \vec{w})$ and $1 - P(1|\vec{x}_i, \vec{w})$ are less than 1. Hence, $L(\vec{w}) \to 0$.

This could lead to numerical underflow. *Try: 0.1**1000 (i.e. 1.0×10^{-1000}) in Python.*

Moreover, our gradients would be vanishing as $L(\vec{w}_k) \approx 0$. *Consequently, if no gradient, then gradient descent will not work.*

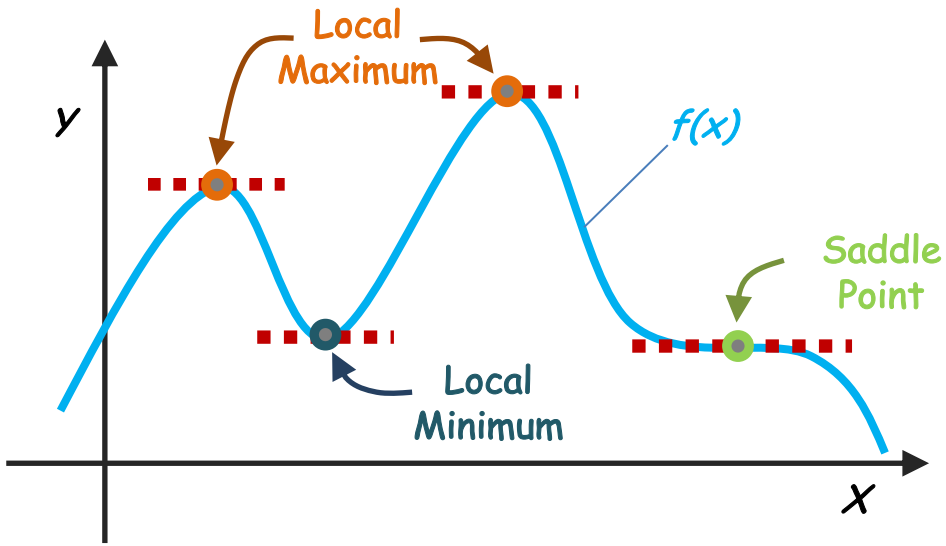
Q: How can we handle very very small numbers?

A: We take Log. 1000xLog(0.1) = 1,000x(-1) = -1,000.

Best Decision Boundary

Maximising Log Likelihood:

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



Q: Which value of x will f(x) be maximum?

A: … Slope (or Gradient) = 0 …

Gradient Descent: Minimisation

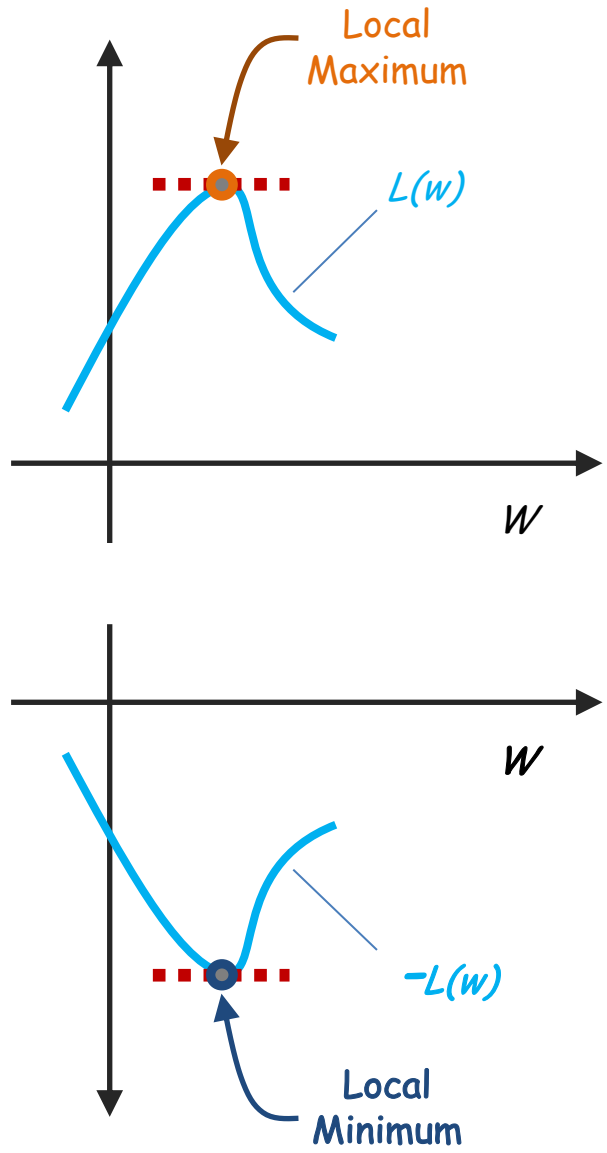
Maximising Log Likelihood:

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

equivalent to

Minimising Negative Log Likelihood:

$$\min_{\vec{w}} -\left(\sum_{i=1}^{N} [y_i \log P(1|\vec{x}_i, \vec{w}) + (1 - y_i) \log(1 - P(1|\vec{x}_i, \vec{w}))] \right)$$



Gradient Vector

Negative Log Likelihood:

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

Derivative of Log Likelihood:

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

Gradient Vector:

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

Pseudocode for Logistic Regression

```
Function Logistic_Regress(new_point, data_points, labels, learning_rate, max_iterations, tolerance)
    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
        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
            errors = labels - predictions
            gradients = -transpose(X) * errors / 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):
    # 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
    weights = gradient_descent_logistic(X, labels, learning_rate, max_iterations, tolerance)

    # 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
```

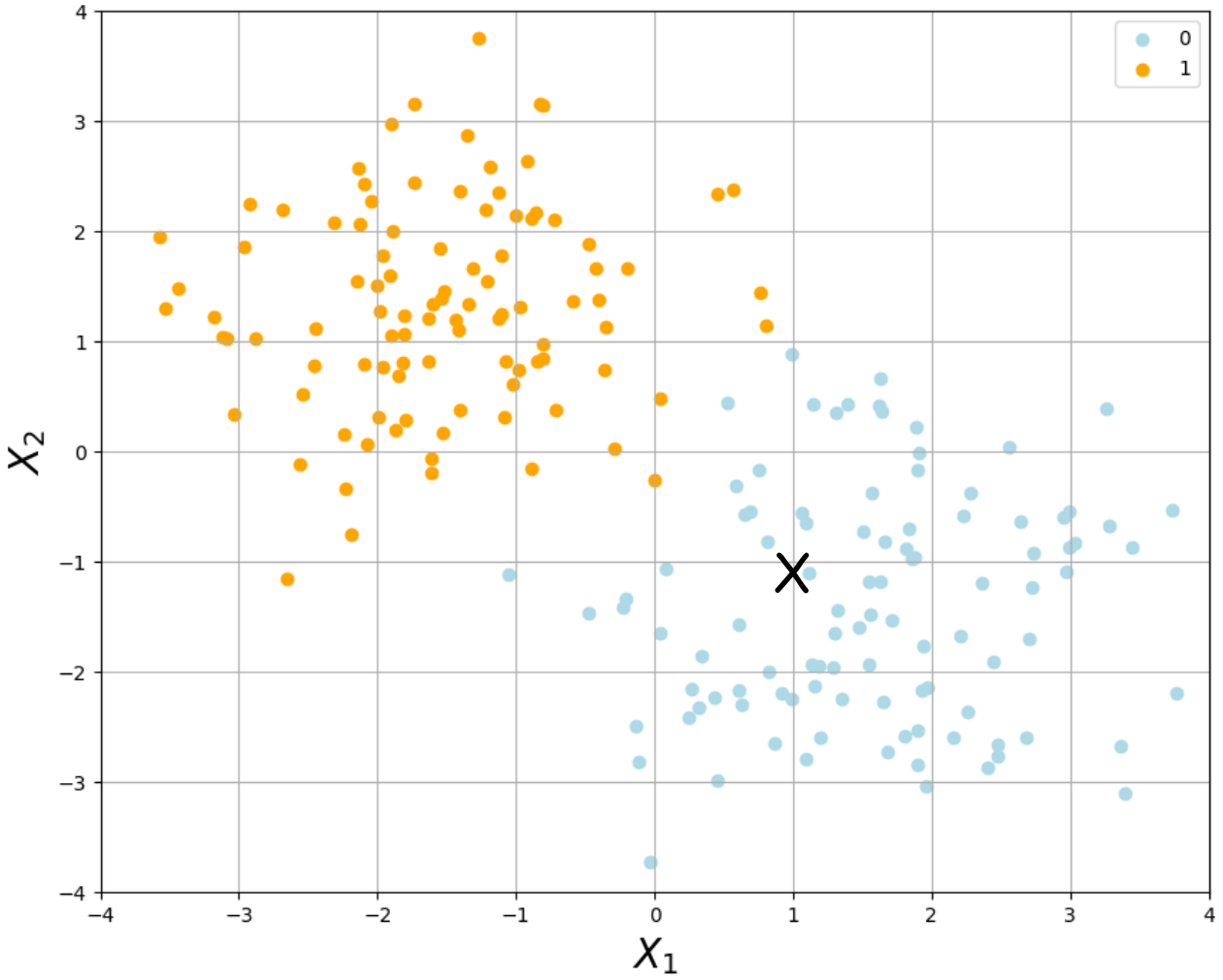
Python Code Snippet (cont.)

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

def compute_gradient(X, y, weights):
    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

    return gradients

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

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

        # 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
learning_rate = 0.1
max_iterations = 1000
tolerance = 1e-6

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

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

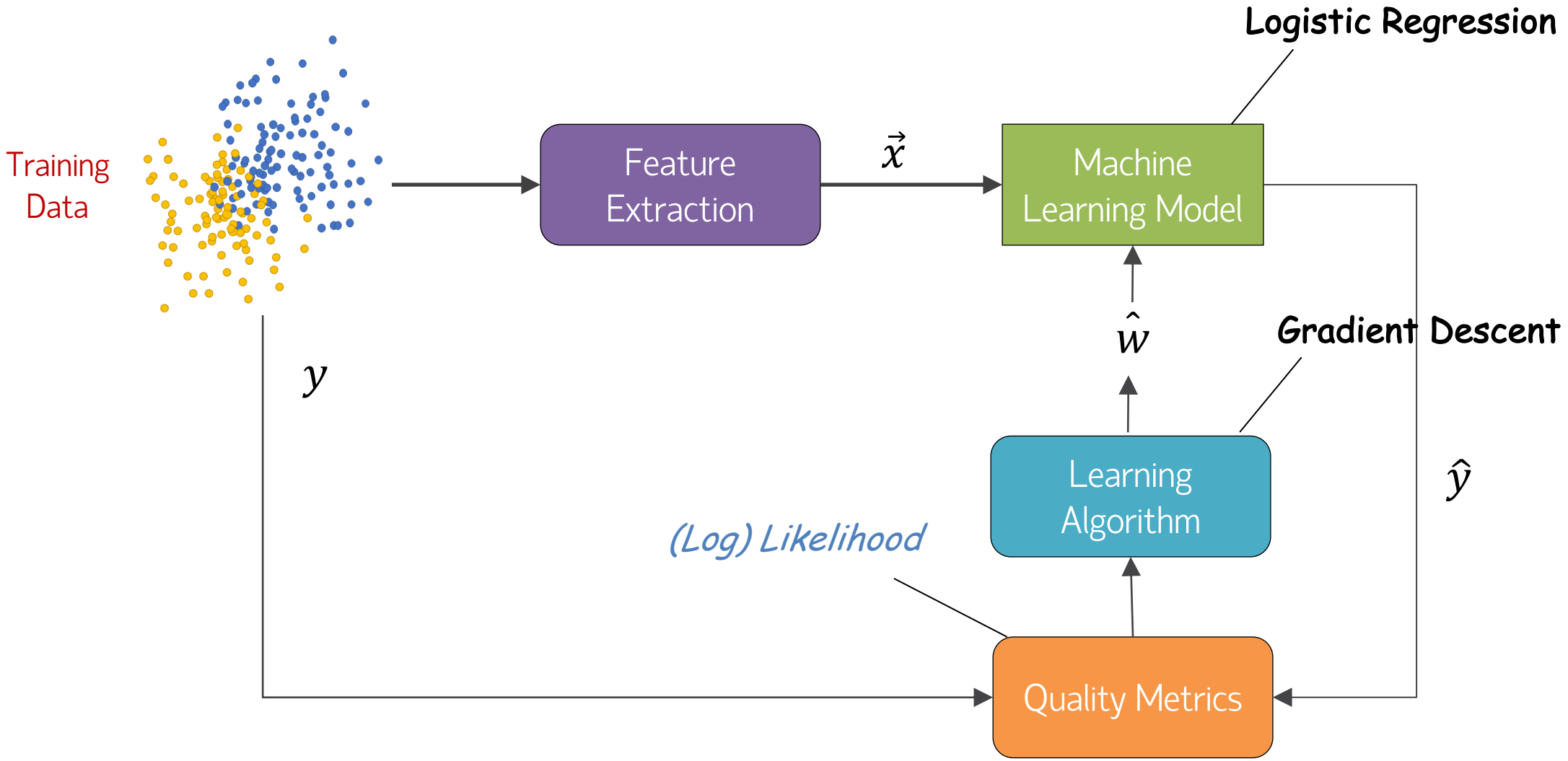
Usage Example (cont.)



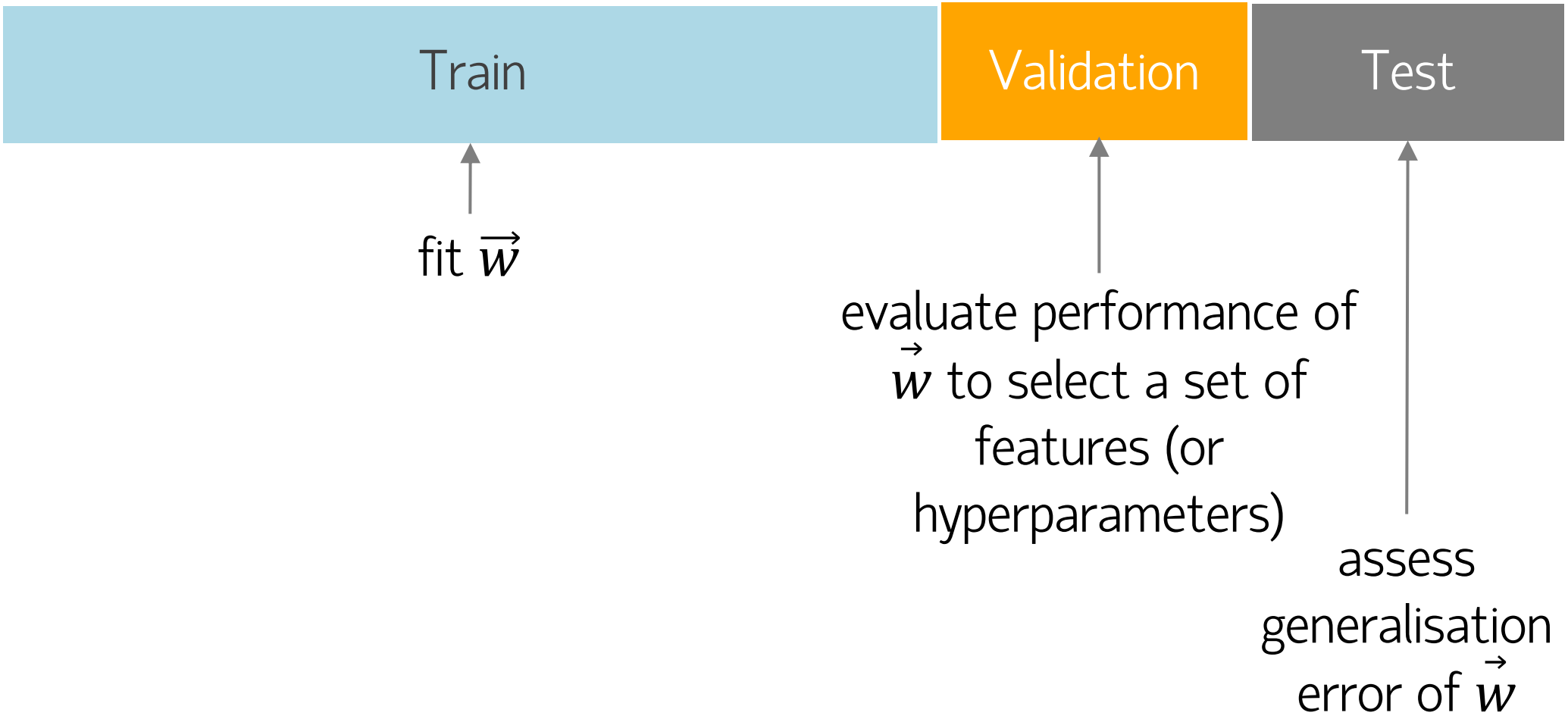

```
Predicted probability for new point: [0.00691733]
```

~0.007 → Class 0

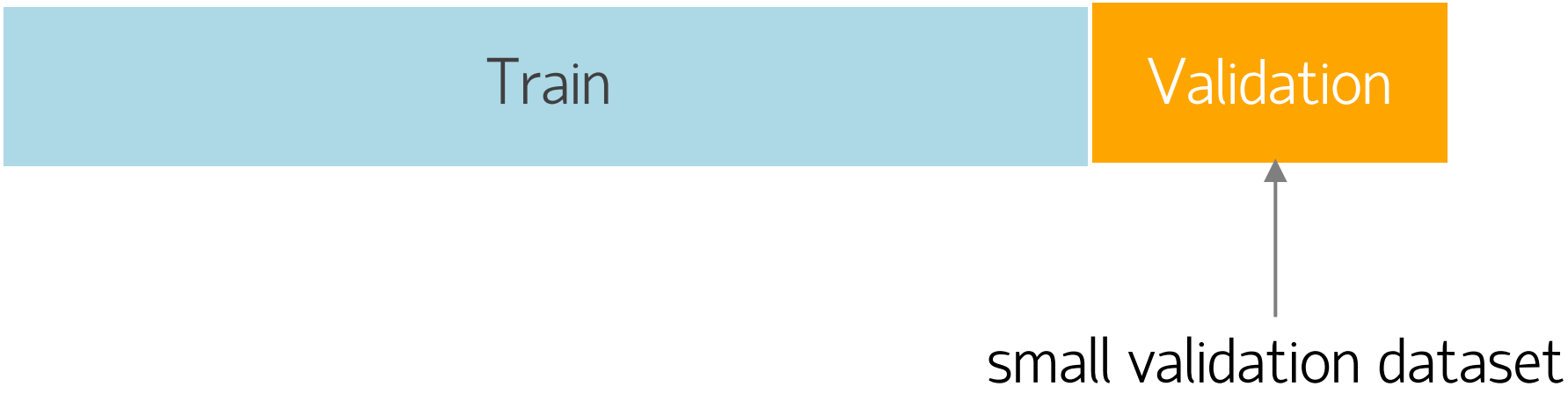
Workflow: Logistic Regression



Large Dataset



Small Dataset



Q: Is validation set enough to compare performance of $\vec{w}$ across a set of features (or hyperparameters)?

A: No.

Choosing Validation Dataset

Q: Which subset should we use?

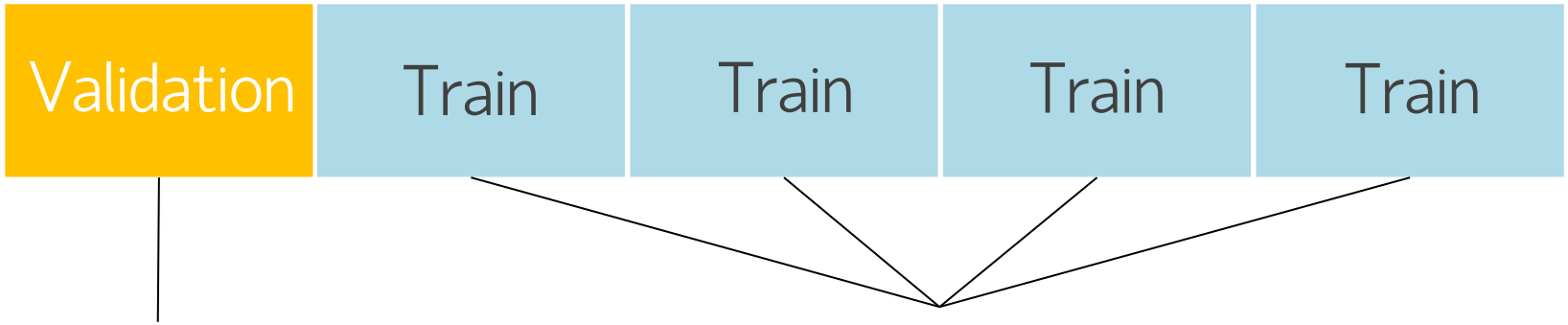
A: All. We can average performance over all choices.

K-Fold Cross Validation

k = 5

We commonly use either 70% or 80% of the overall data for training. For K-fold, we randomly split the data into K groups (or folds), where K is typically either 3, 5 or 10.

K-Fold Cross Validation



K = 5

$\text{Error}_1([x^{(1)}, \ldots, x^{(M)}])$

$\vec{w}^{(1)}_{[x^{(1)},\ldots,x^{(M)}]}$

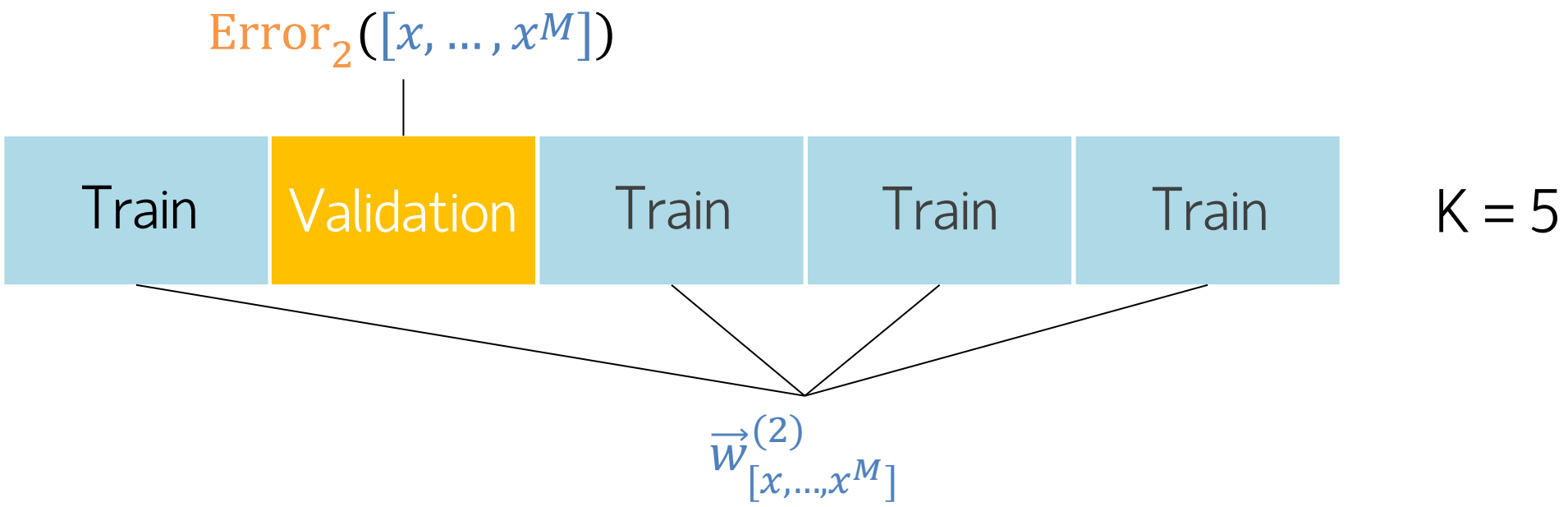
$[x^{(1)}, \ldots, x^{(M)}]$ is a set of features.

For k = 1, ..., K

Step 1: Estimate $\vec{w}^{(k)}_{[x^{(1)},\ldots,x^{(M)}]}$ on the training blocks.

Step 2: Compute error on Validation Block: $\text{Error}_k([x^{(1)}, \ldots, x^{(M)}])$.

K-Fold Cross Validation

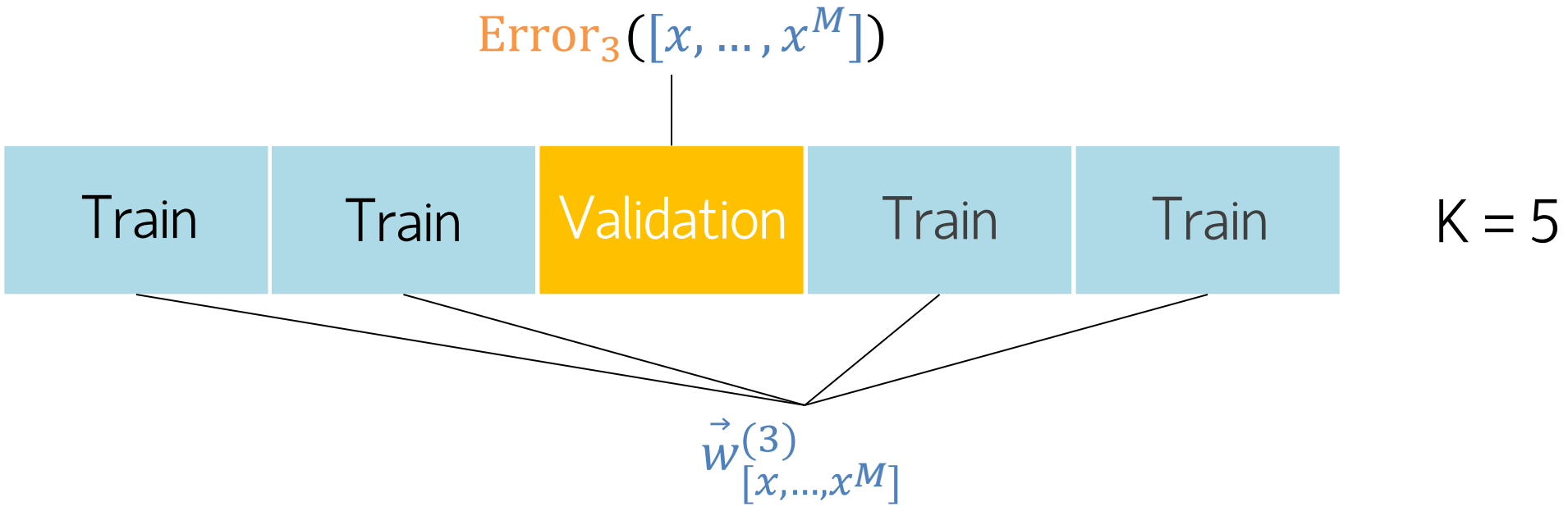


For k = 1, ..., K

Step 1: Estimate $\vec{w}^{(k)}_{[x,\ldots,x^M]}$ on the training blocks.

Step 2: Compute error on Validation Block: $\text{Error}_k([x, \ldots, x^M])$.

K-Fold Cross Validation

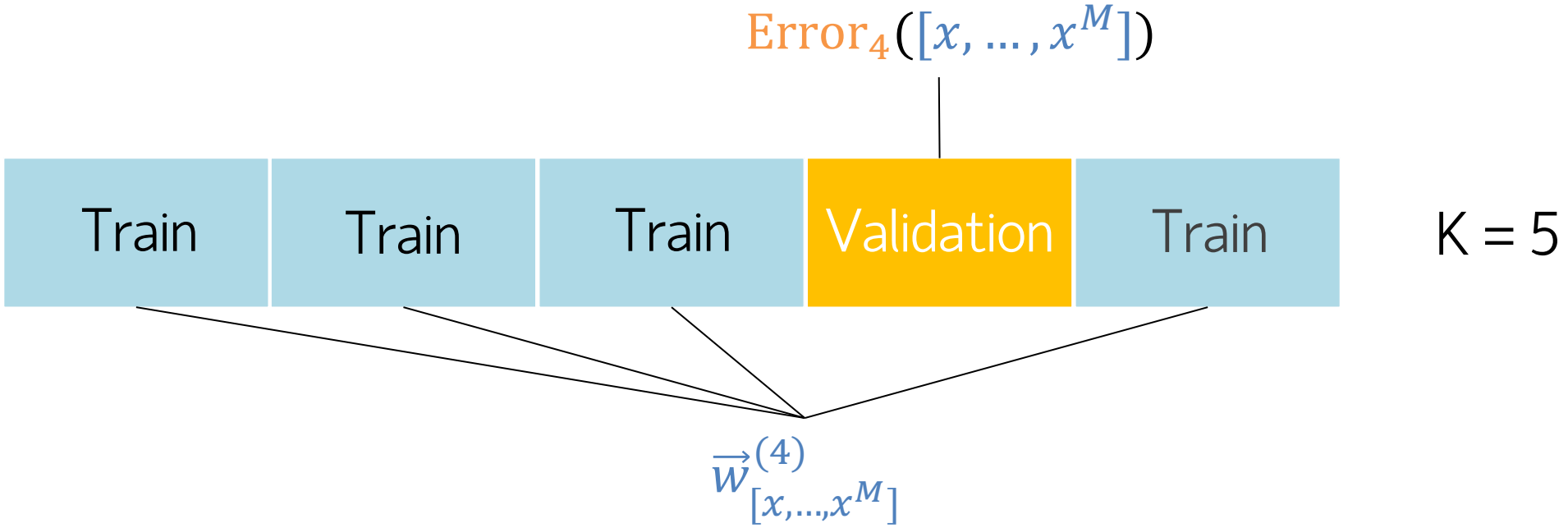


For k = 1, ..., K

Step 1: Estimate $\vec{w}^{(k)}_{[x,\ldots,x^M]}$ on the training blocks.

Step 2: Compute error on Validation Block: $\text{Error}_k([x, \ldots, x^M])$.

K-Fold Cross Validation

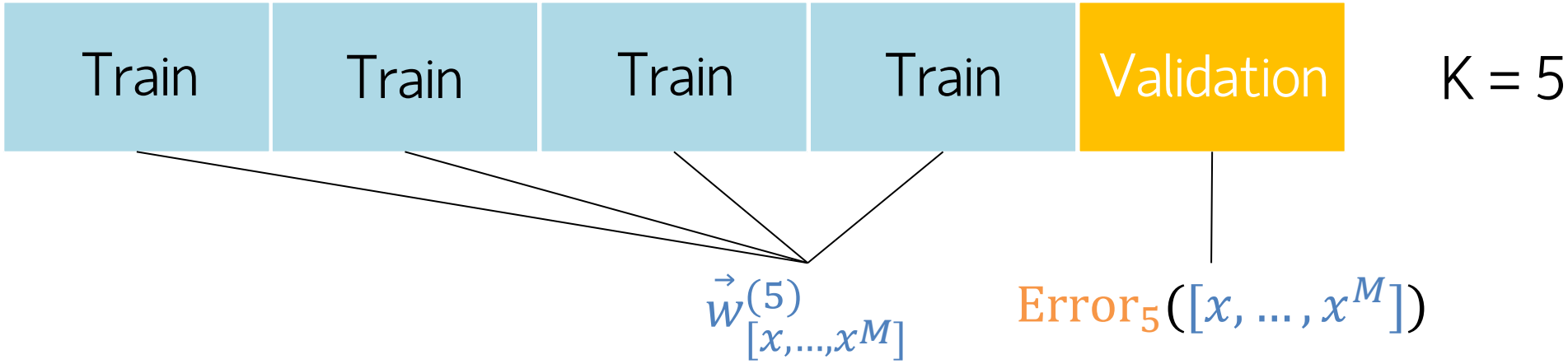


For k = 1, ..., K

Step 1: Estimate $\vec{w}^{(k)}_{[x,\ldots,x^M]}$ on the training blocks.

Step 2: Compute error on Validation Block: $\text{Error}_k([x, \ldots, x^M])$.

K-Fold Cross Validation



For k = 1, …, K

Step 1: Estimate $\vec{w}^{(k)}_{[x,\ldots,x^M]}$ on the training blocks.

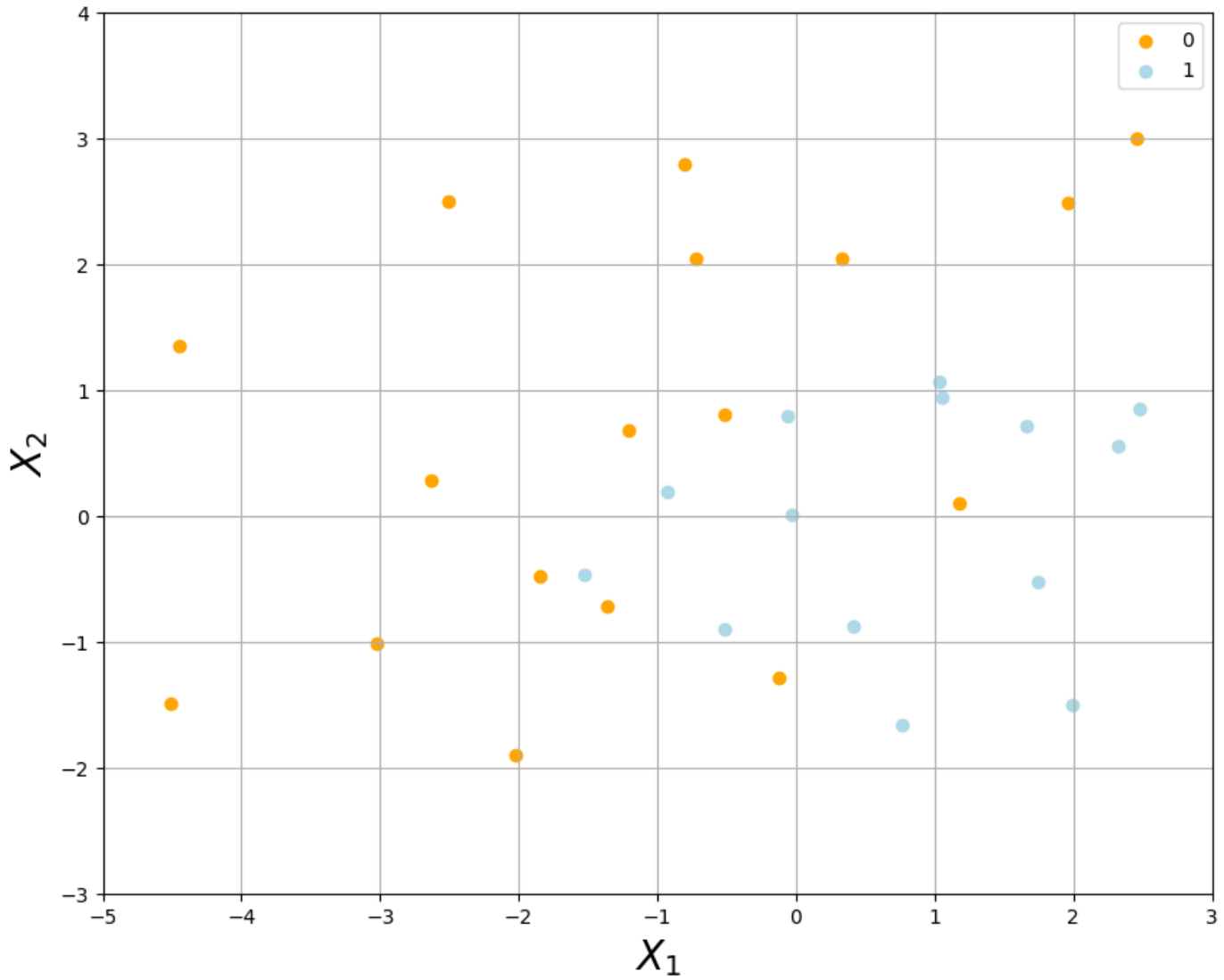
Step 2: Compute error on Validation Block: $\text{Error}_k([x, \ldots, x^M])$.

Compute Average Error: $\text{CV}([x, \ldots, x^M]) = \frac{1}{K}\sum_{k=1}^{K} \text{Error}_k([x, \ldots, x^M])$.

Pseudocode for Cross-Validation

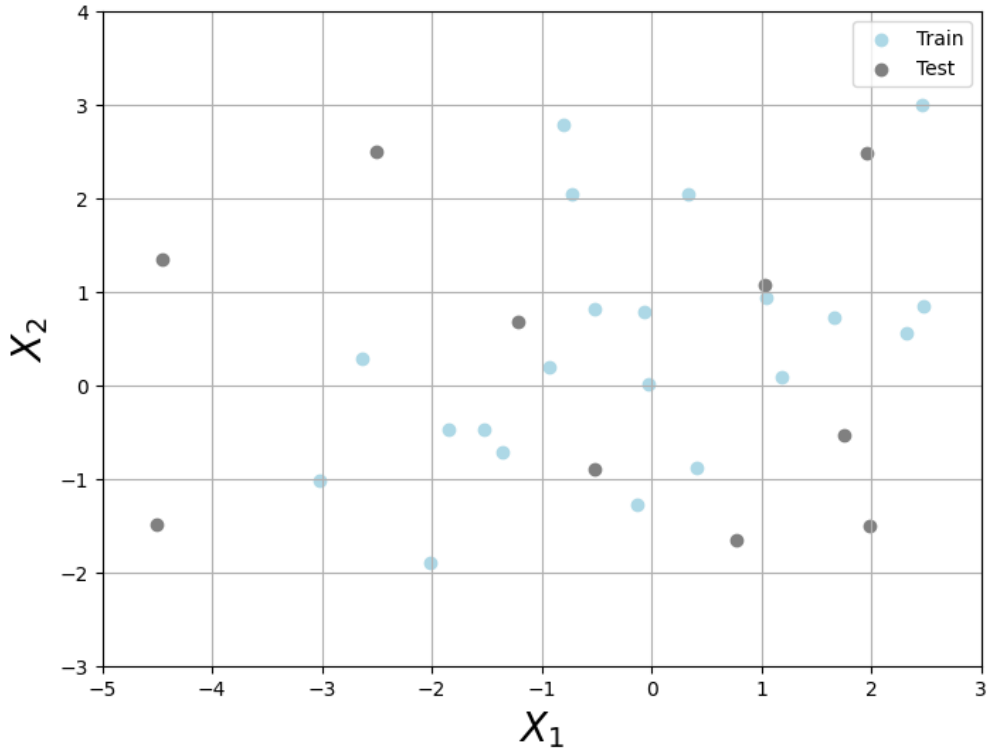
```
Function K_Fold_Cross_Validation(dataset, model, k)
    Begin
        // Step 1: Split the dataset into k folds
        Set folds = Split dataset into k equal parts
        Initialize an empty list: validation_scores

        // Step 2: Perform k rounds of training and validation
        For i = 0 to k - 1 do
            // Step 2.1: Set up training and validation sets
            Set validation_set = folds[i]
            Set training_set = Concatenate all folds except fold i

            // Step 2.2: Train the model
            Train the model on the training_set

            // Step 2.3: Evaluate the model
            Set validation_score = Evaluate the model on the validation_set
            Add validation_score to validation_scores list

        // Step 3: Calculate the average validation score
        Set average_score = Average of all scores in validation_scores

        // Step 4: Return the average validation score
        Return average_score
    End
```

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)
```



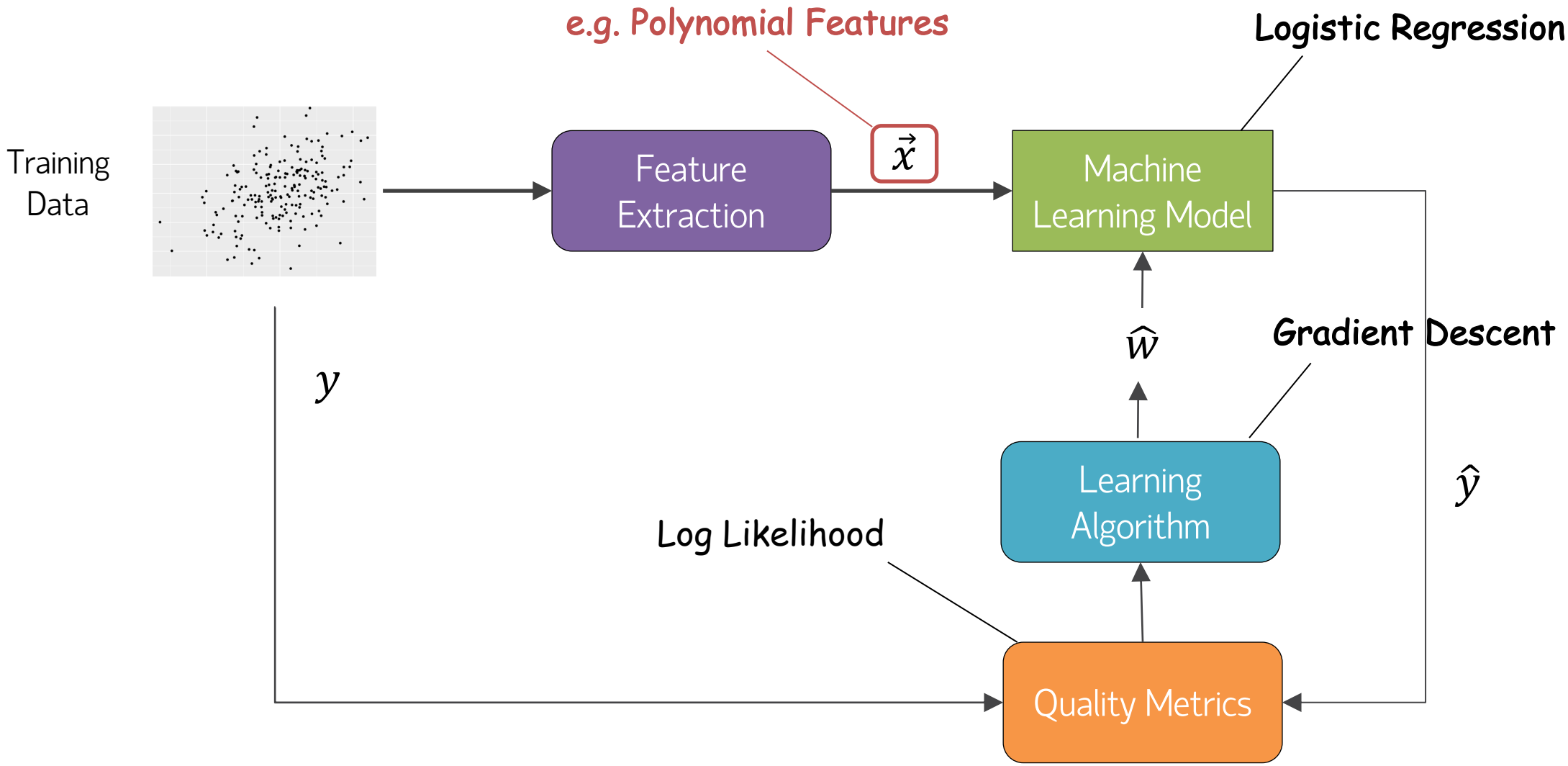
- Train:Test is typicaly either 0.8:0.2 or 0.7:0.3.
- **Test** dataset is a proxy of unseen data, and it will only be used in the final evaluation.
- **Train** dataset is further divided into k folds (e.g., 5 or 10).For each fold, the model is trained on k-1 folds and validated on the remaining fold.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

scikit-learn: Logistic Regression

```
# Step 1: Create an instance of LogisticRegression
model = LogisticRegression(penalty=None)

# 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)))
```


```
Mean Cross-Validation Accuracy: 0.686
```

Workflow: Logistic Regression

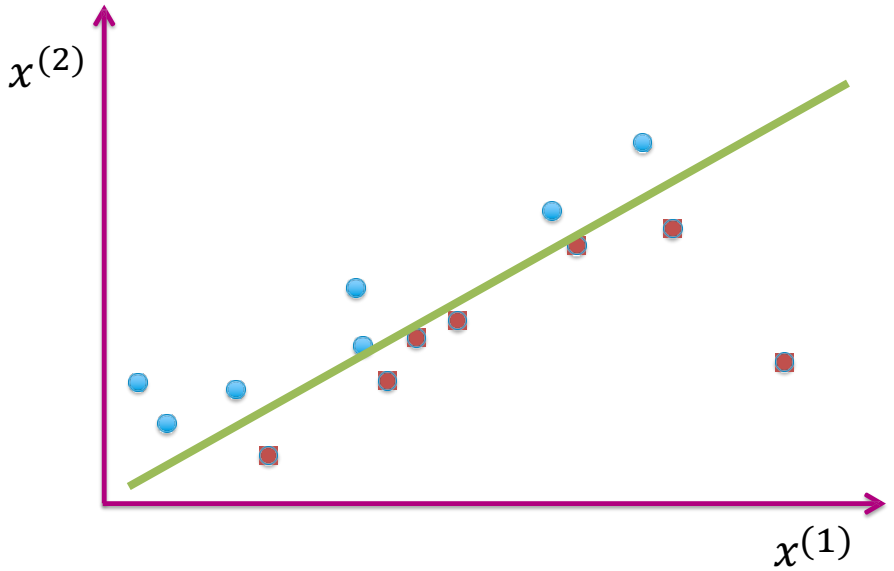


Feature Extraction: Polynomial Features

$$\vec{x} = \begin{bmatrix} x^{(1)} \\ x^{(2)} \end{bmatrix}$$

$$0 = \vec{x}^T \times \vec{w}$$

$$0 = w^{(0)} + w^{(1)}x^{(1)} + w^{(2)}x^{(2)}$$



1st Degree: $\vec{x} = [x^{(1)} \quad x^{(2)}]$

2nd Degree: $\vec{x} = [x^{(1)} \quad x^{(2)} \quad (x^{(1)})^2 \quad (x^{(2)})^2]^T$

3rd Degree: $\vec{x} = [x^{(1)} \quad x^{(2)} \quad (x^{(1)})^2 \quad (x^{(2)})^2 \quad (x^{(1)})^3 \quad (x^{(2)})^3]^T$

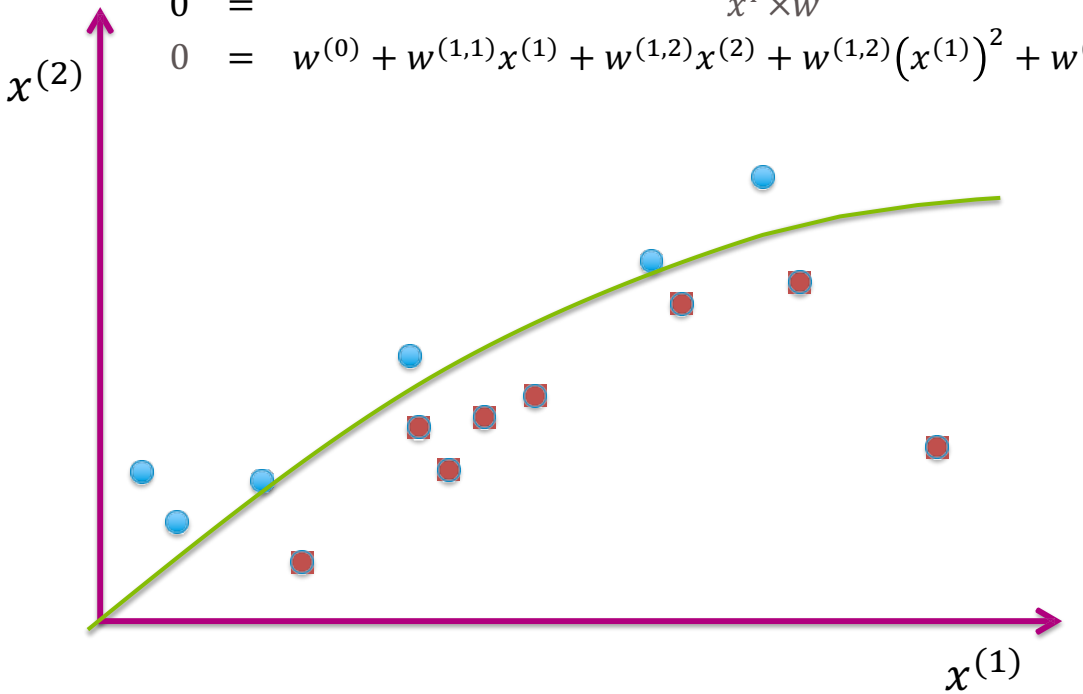
$\vdots$

Nth Degree: $\vec{x} = [x^{(1)} \quad x^{(2)} \quad \cdots \quad (x^{(1)})N \quad (x^{(2)})N]^T$

$$\vec{x} = \begin{bmatrix} x^{(1)} \\ x^{(2)} \end{bmatrix}$$

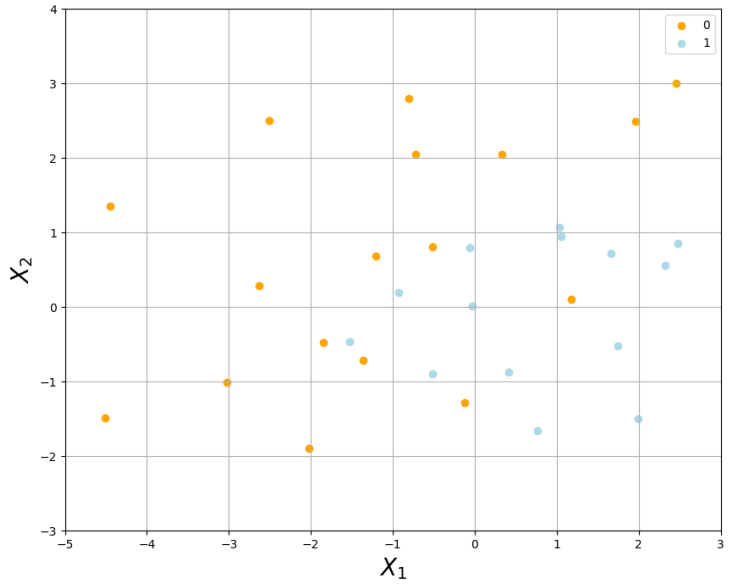
$$0 = \vec{x}^T \times \vec{w}$$

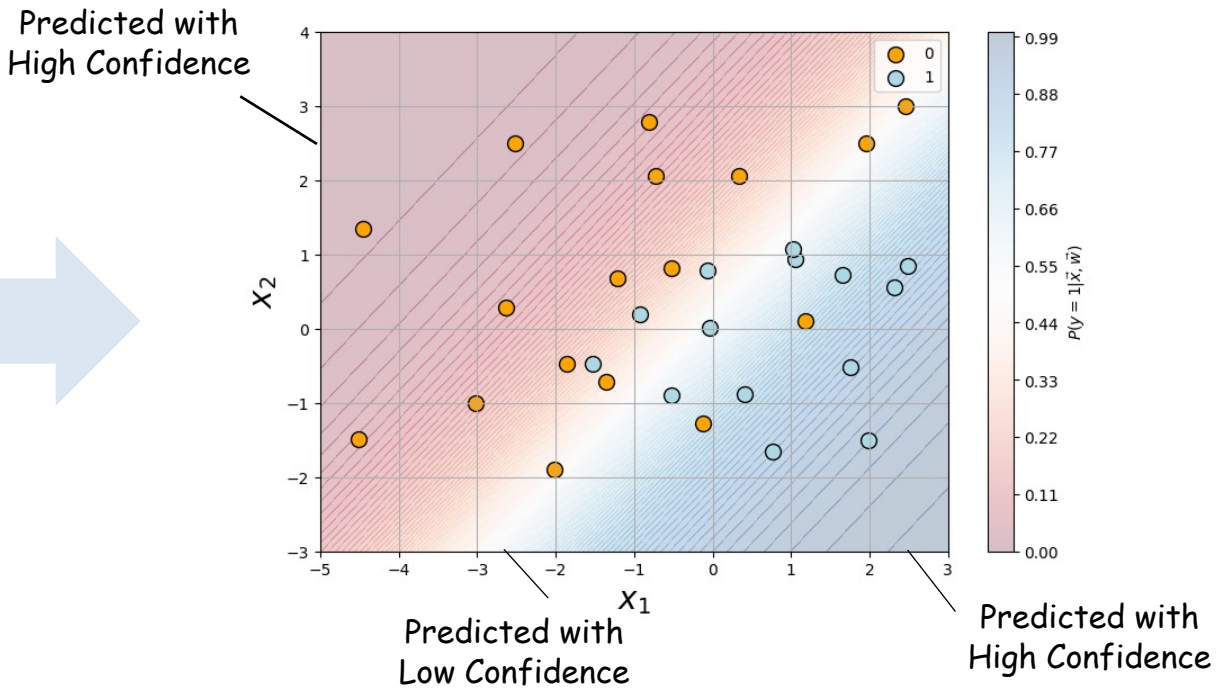
$$0 = w^{(0)} + w^{(1,1)}x^{(1)} + w^{(1,2)}x^{(2)} + w^{(1,2)}(x^{(1)})^2 + w^{(2,2)}(x^{(2)})^2$$



Learned Decision Boundary

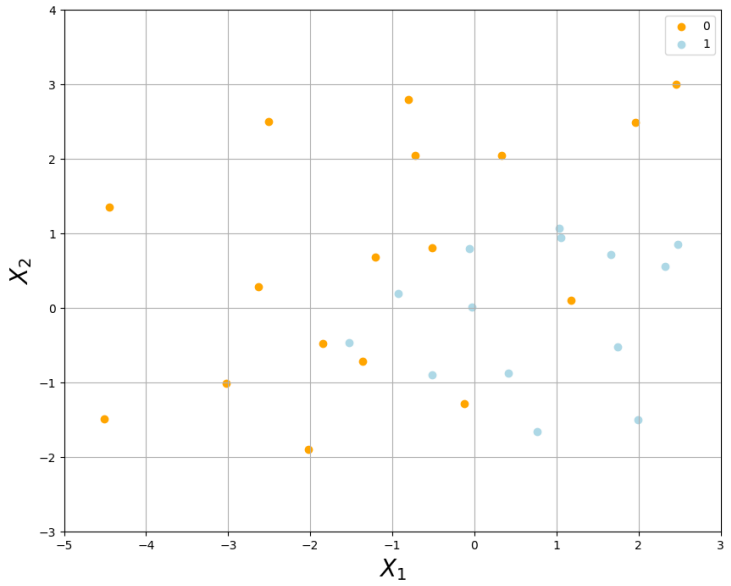
Feature	Coefficient
1	0.246
x_1	1.113
x_2	-1.062

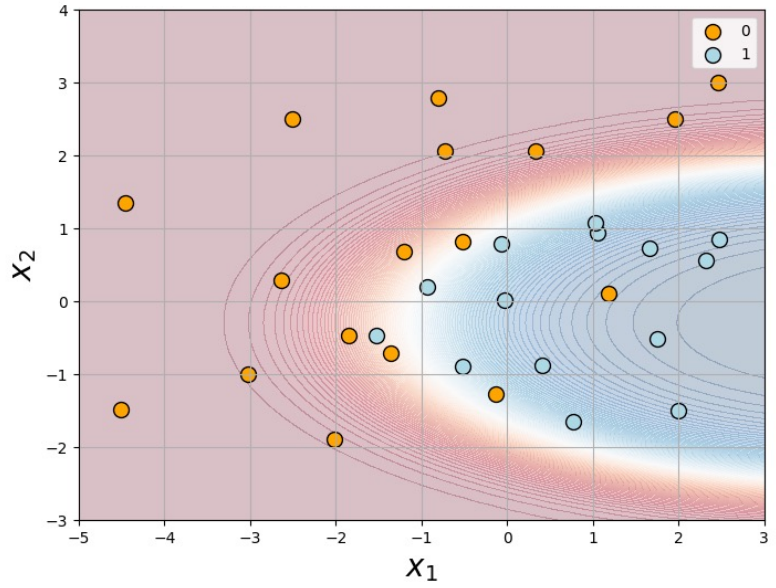




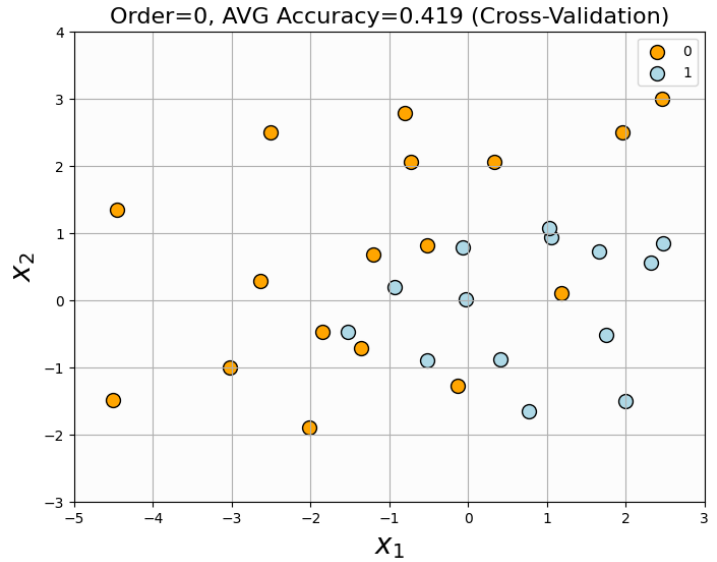
Learned Decision Boundary (2nd Order)

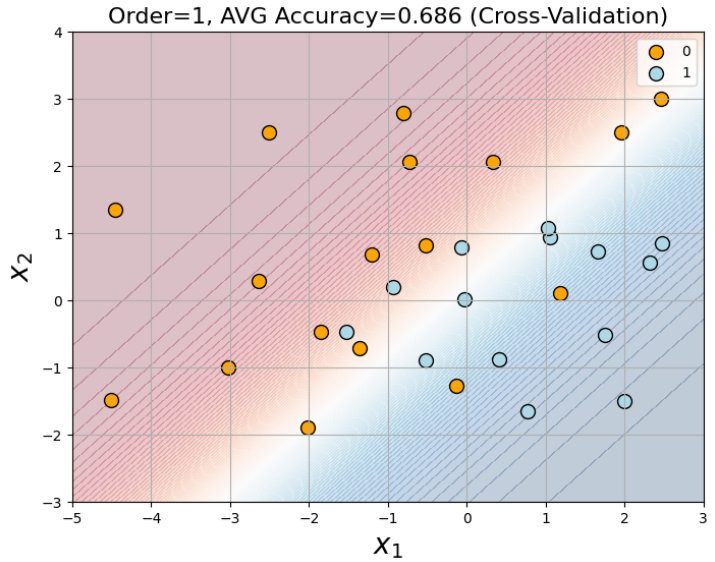
Feature	Coefficient
1	1.717
x_1	1.385
x_2	-0.579
x_1^2	-0.167
x_2^2	0.978

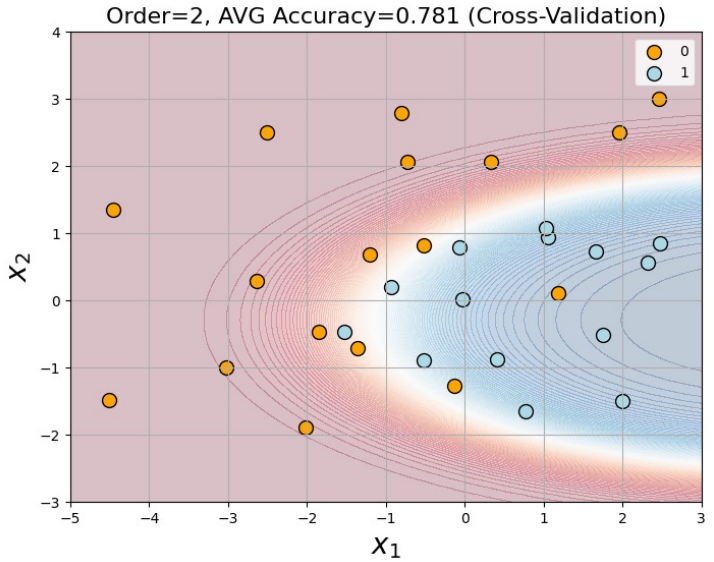


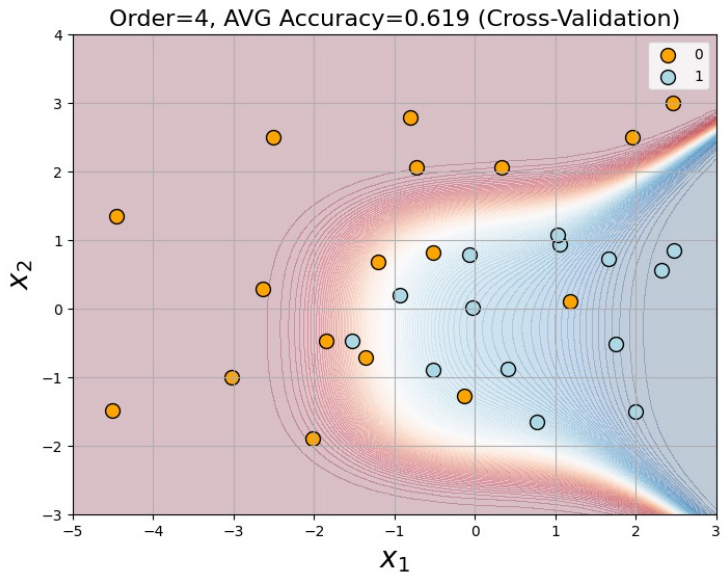


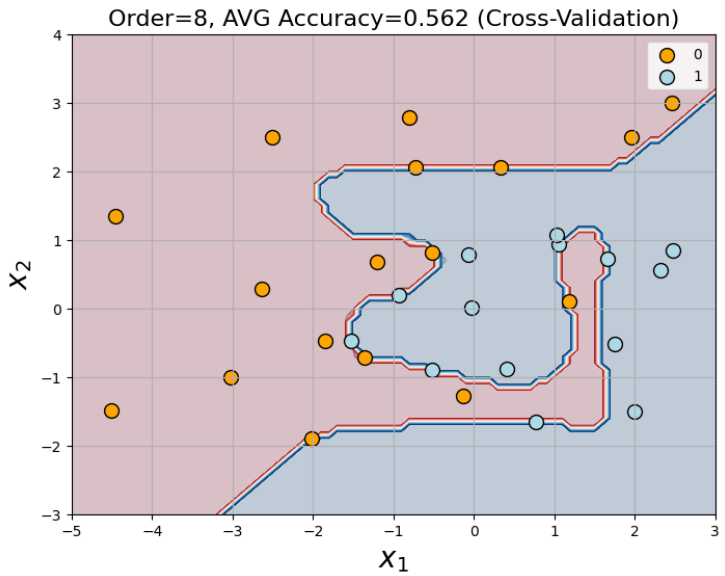
Model Complexity

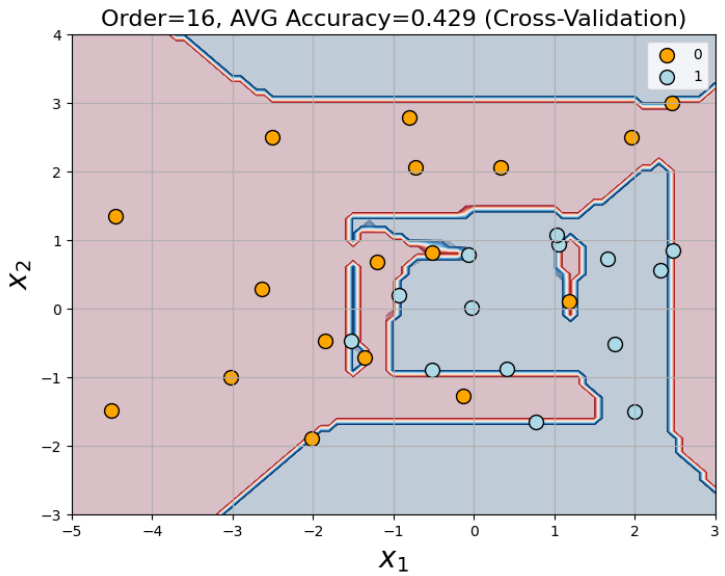








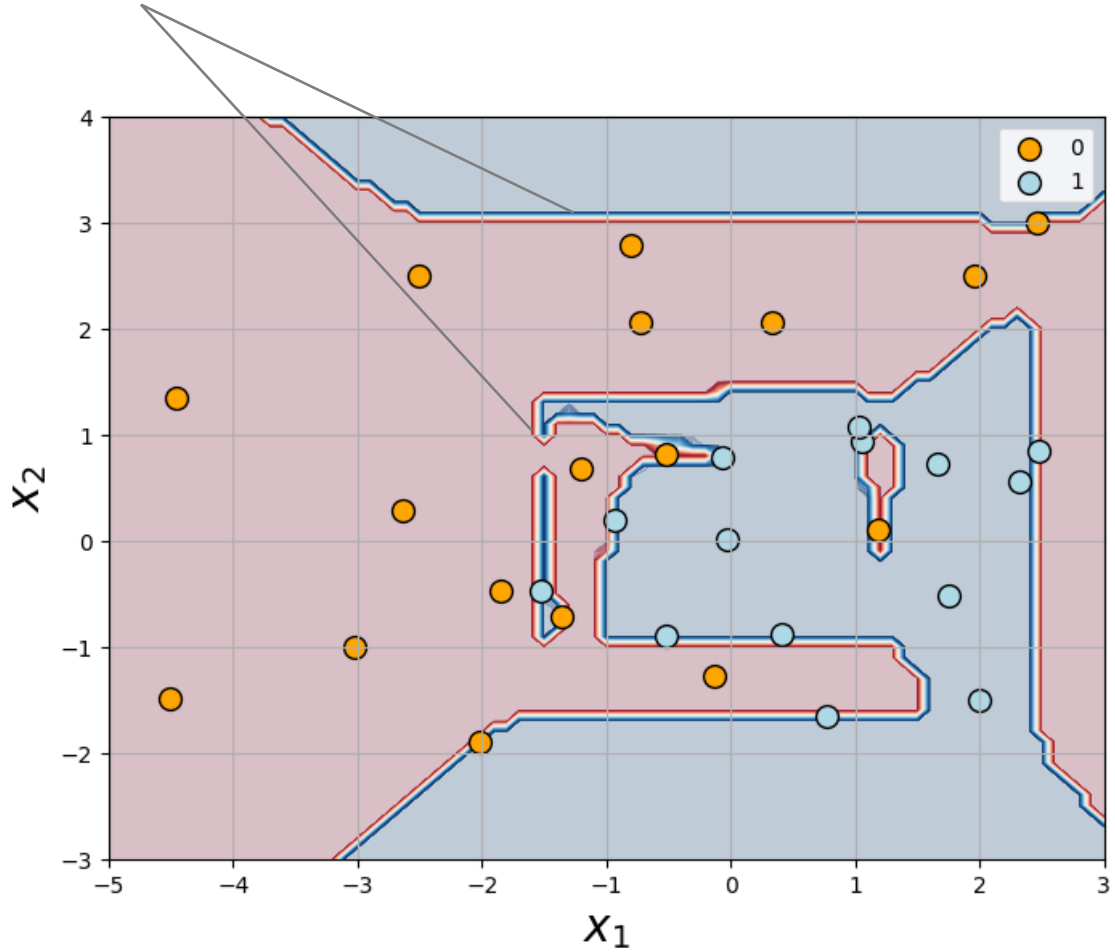




Over-Confident Predictions

Tiny Uncertain Regions → Overfitting & Over-Confident Predictions

We are sure we are right, when we are surely wrong.



MyLogisticRegressor (Revised)

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


class MyLogisticRegressor(BaseEstimator, ClassifierMixin):
    def __init__(self, learning_rate=0.01, max_iterations=1000, tolerance=1e-6):
        self.learning_rate = learning_rate
        self.max_iterations = max_iterations
        self.tolerance = tolerance
        self.weights = None

    def fit(self, X, y):
        # Add bias term (column of ones) to X
        X = np.hstack([np.ones((X.shape[0], 1)), X])

        # Initialize weights to zeros
        self.weights = np.zeros((X.shape[1], 1))

        # Reshape y to a column vector if it's not already
        y = y.reshape(-1, 1)

        # Perform gradient descent to optimize weights
        self.weights = gradient_descent(
            X, y, self.weights, self.learning_rate, self.max_iterations, self.tolerance
        )

        return self
```

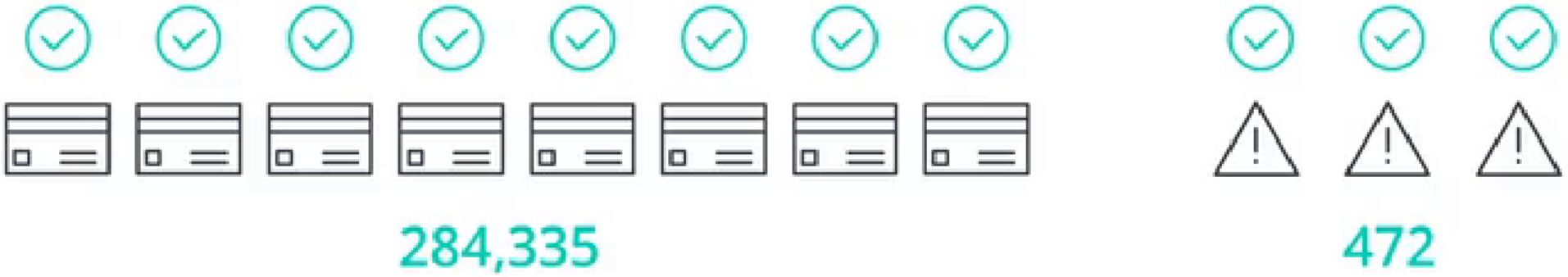
MyLogisticRegressor (cont.)

```
…cont…

    def predict_proba(self, X):
        # Add bias term to X
        X = np.hstack([np.ones((X.shape[0], 1)), X])
        # Compute probabilities
        probabilities = expit(X @ self.weights)
        return np.hstack([1 - probabilities, probabilities])

    def predict(self, X):
        # Predict class labels
        probabilities = self.predict_proba(X)[:, 1]
        return (probabilities ⩾ 0.5).astype(int)
```

MyLogisticRegressor (cont.)

```
def compute_gradient(X, y, weights):
    predictions = expit(X @ weights)  # Predicted probabilities
    errors = y - predictions           # Error between actual and predicted values
    gradients = -X.T @ errors / X.shape[0]  # Average gradient

    return gradients

def gradient_descent(X, y, weights, learning_rate, max_iterations, tolerance):
    for iteration in range(max_iterations):
        gradients = compute_gradient(X, y, weights)
        weights = weights - learning_rate * gradients

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

    return weights
```

Credit Card Fraud



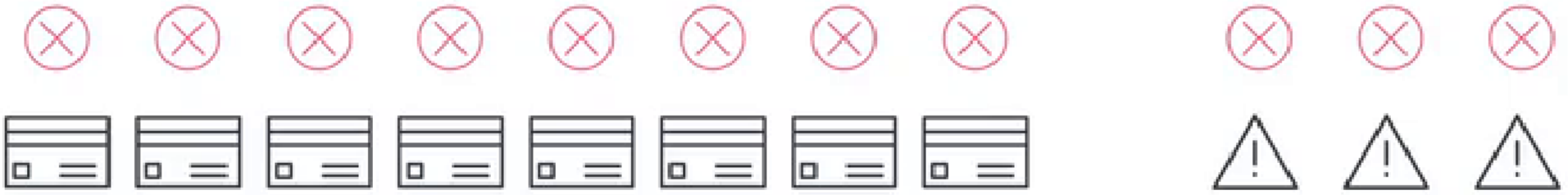
Our model predicted all transaction are good.

$$\text{Accuracy} = \frac{284{,}335}{284{,}887}$$

$$= 99.83\%$$

Problem: We're not catching any of the bad ones.

Credit Card Fraud

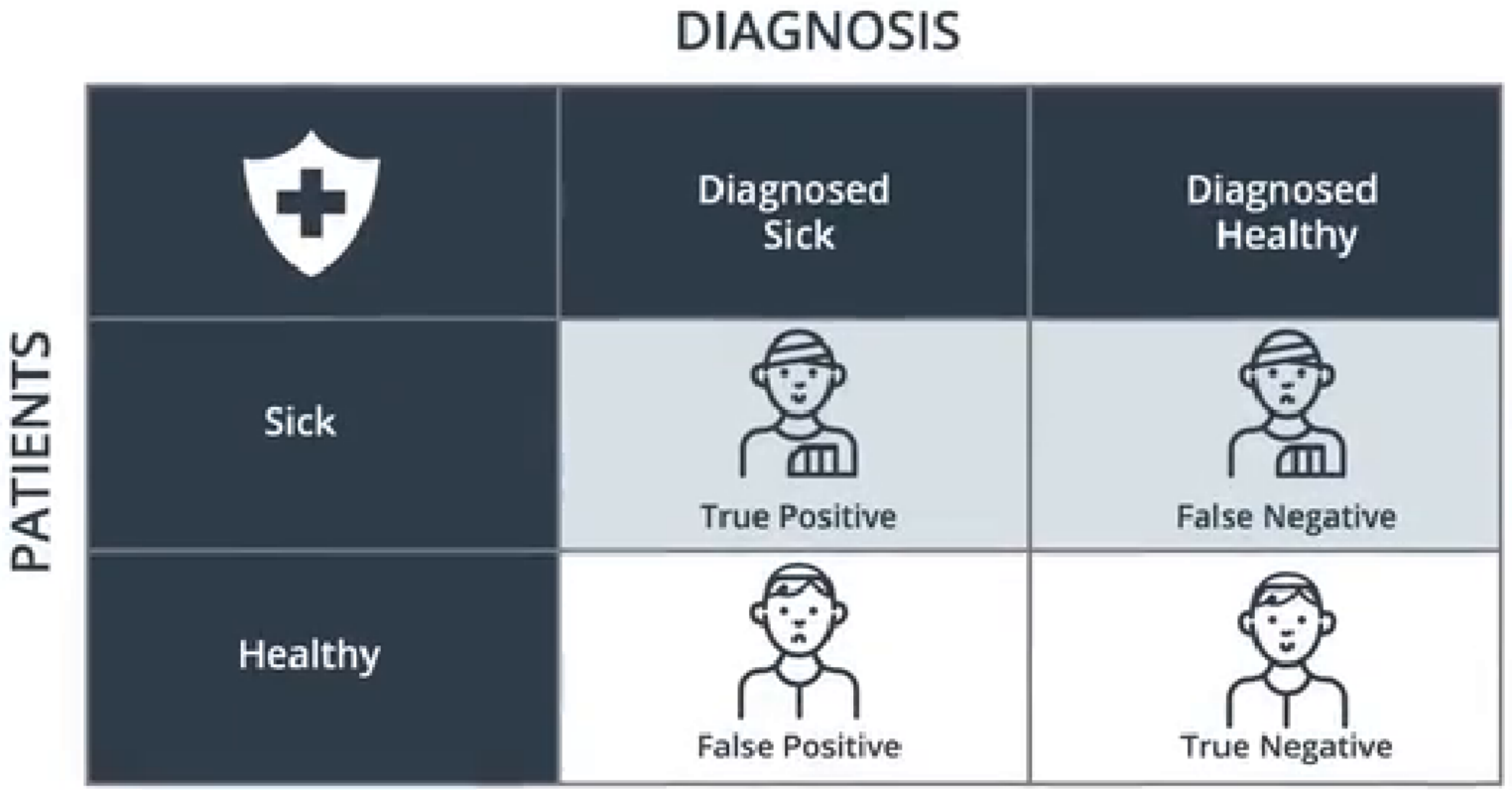


Our model predicted all transactions are fraudulent.

Great! Now, we're catching all of the fraudulent transactions.

Problem: We're accidentally catching all of the good ones.

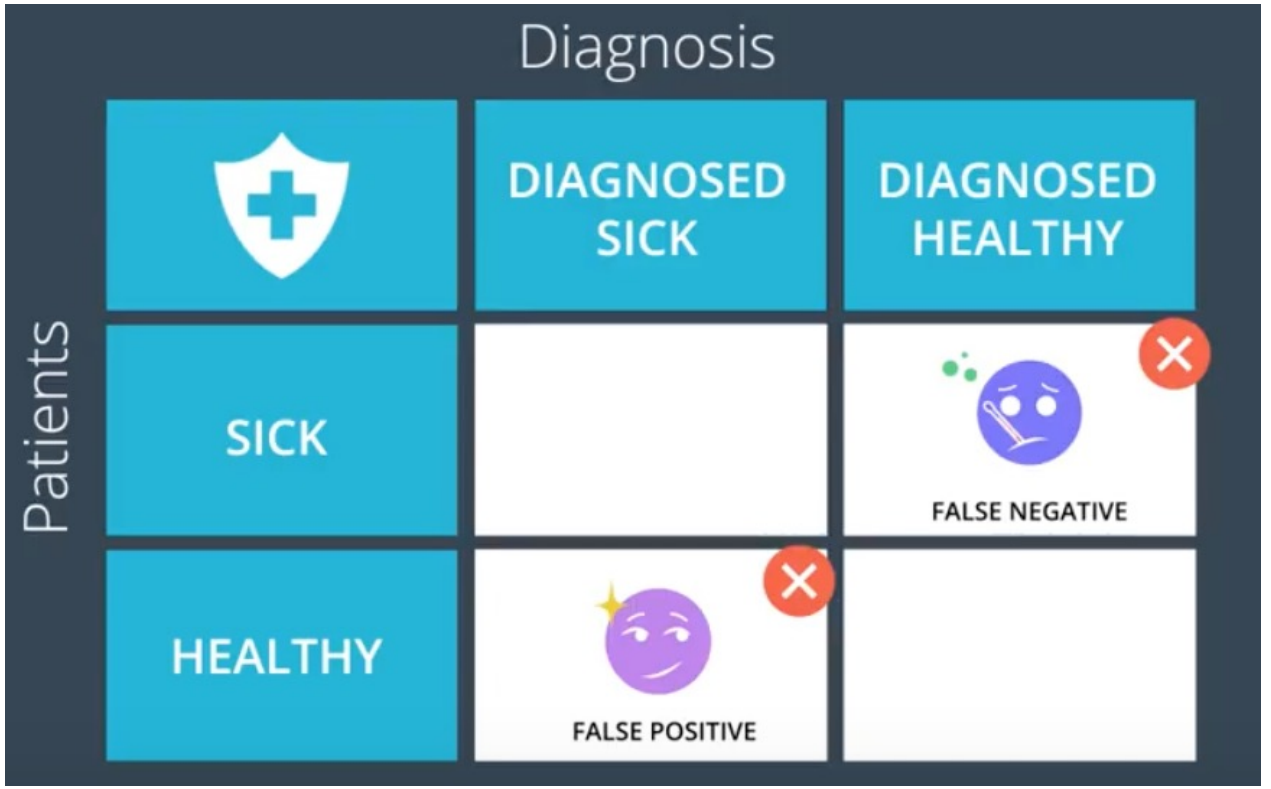
Quiz: False Positives and Negatives



Quiz: False Positives and Negatives

Which one do you think is worse, a false positive or a false negative? In other words, what is the worst mistake, to misdiagnose a healthy patient as sick or a sick patient as healthy?

- ☐ False Positive
- ☐ False Negative



Quiz: False Positives and Negatives

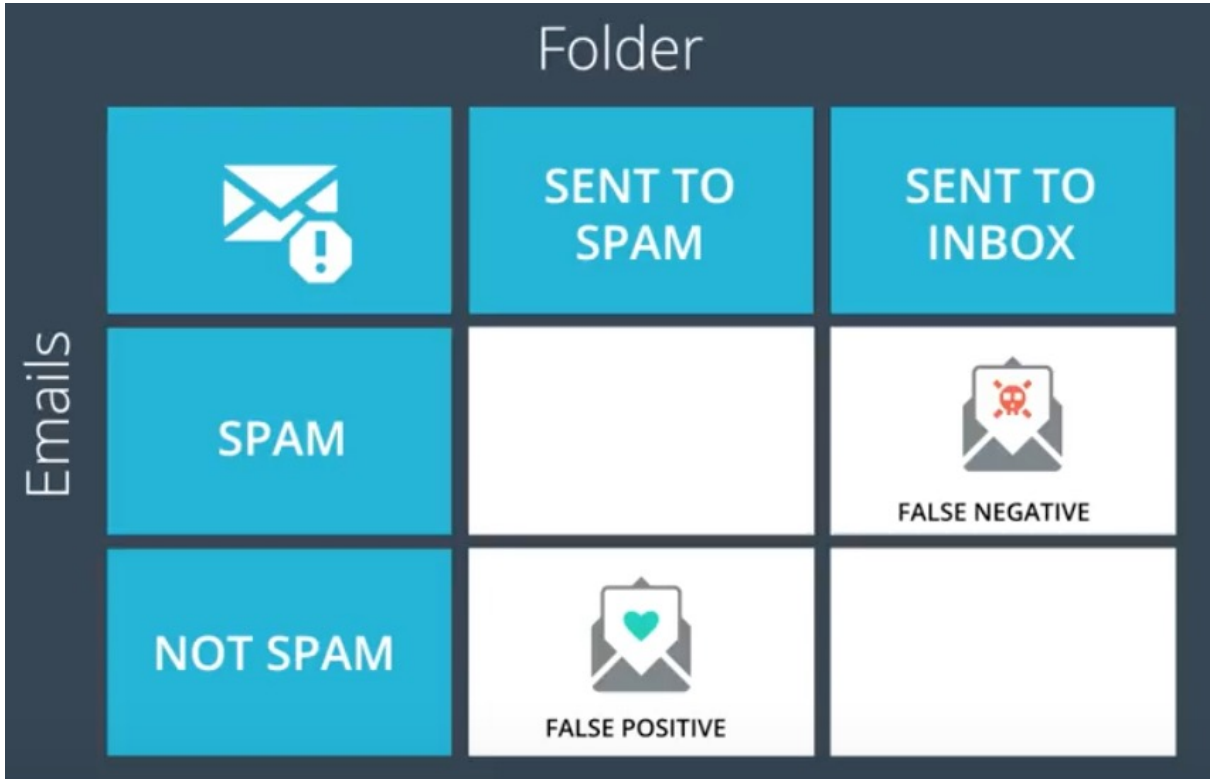
FOLDER

EMAIL	Sent to Spam Folder	Sent to Inbox
Spam	True Positive	False Negative
Not Spam	False Positive	True Negative

Quiz: False Positives and Negatives

Which one do you think is worse, a false positive or a false negative? In other words, what is the worst mistake, to accidentally send your grandma's email to the spam folder or to accidentally send the spam email into your inbox?

- ☐ False Positive
- ☐ False Negative

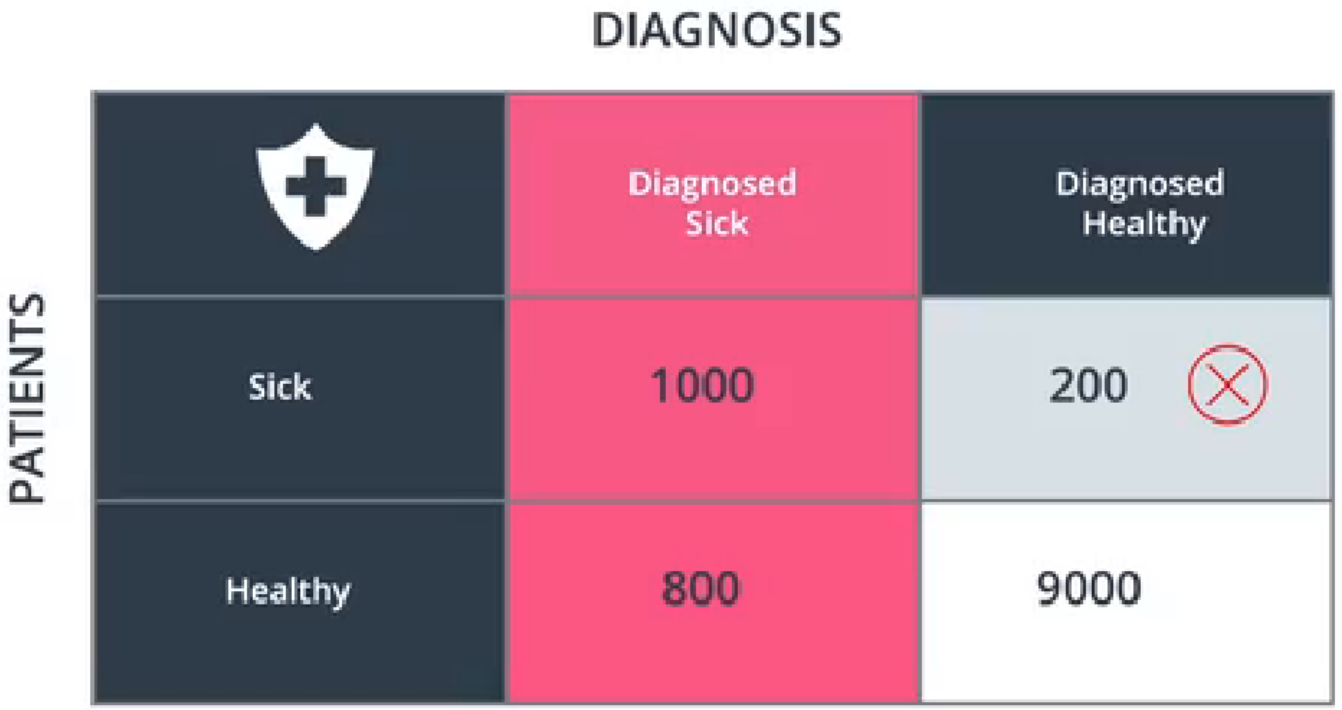


Solution: False Positives and Negatives

High Recall

High Precision

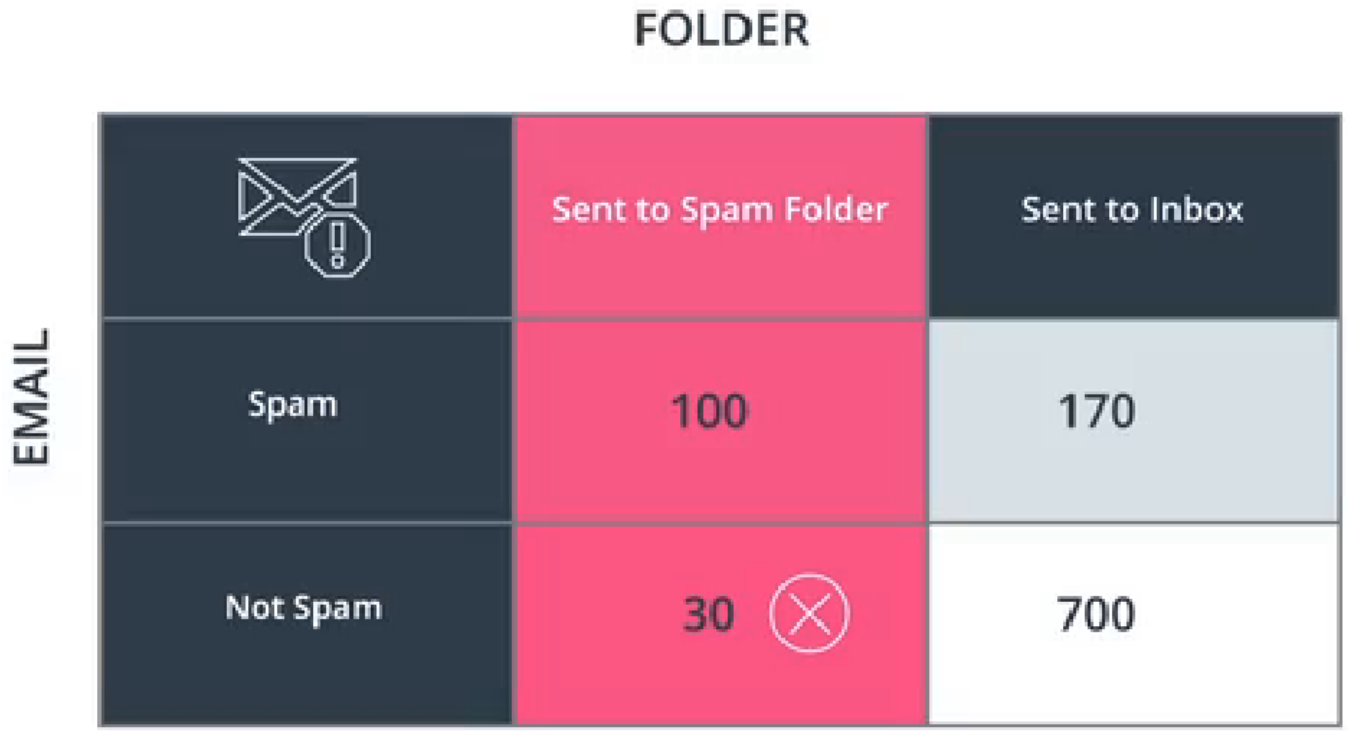
Precision



PRECISION: OUT OF THE PATIENTS WE DIAGNOSED WITH AN ILLNESS, HOW MANY DID WE CLASSIFY CORRECTLY?

$$\text{PRECISION} = \frac{1{,}000}{1{,}000 + 800} = 55.6\%$$

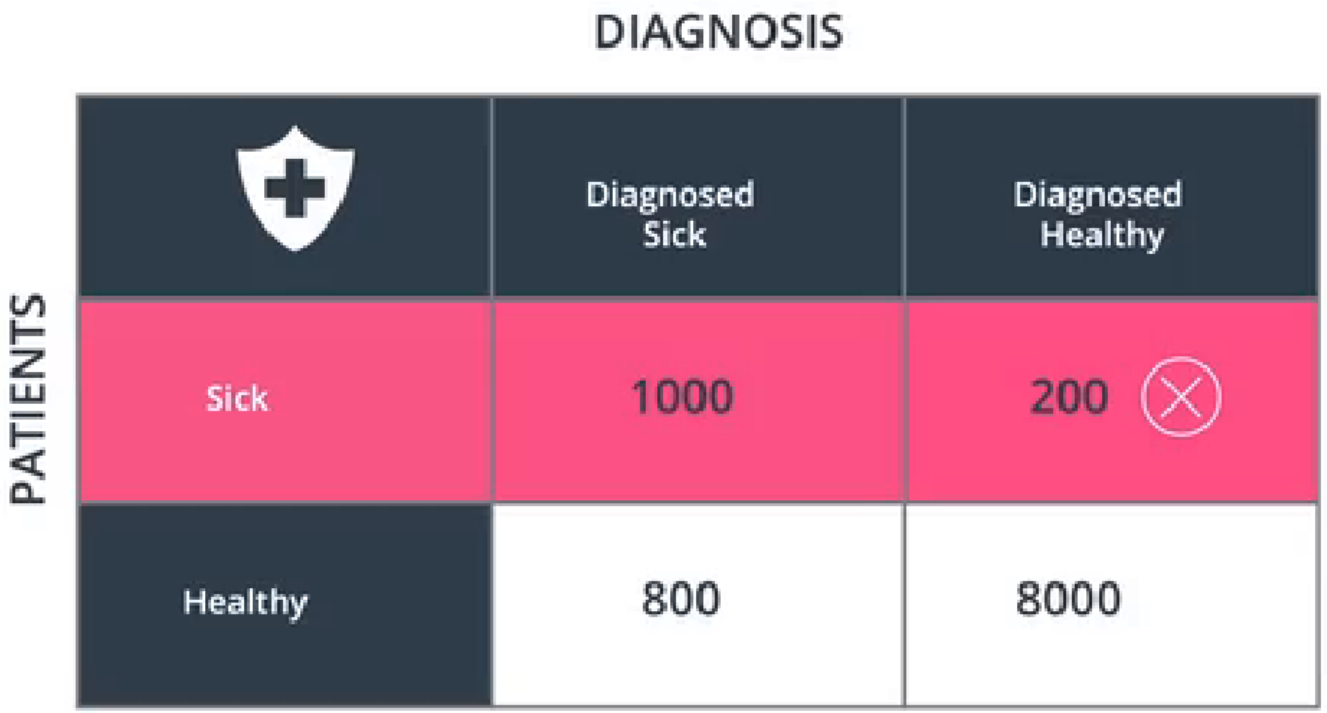
Precision



OUT OF ALL THE E-MAILS
SENT TO THE SPAM FOLDER,
HOW MANY WERE ACTUALLY SPAM?

$$\text{PRECISION} = \frac{100}{100 + 30}$$

Recall



OUT OF THE SICK PATIENTS, HOW MANY DID WE CORRECTLY DIAGNOSE AS SICK?

$$\text{RECALL} = \frac{1{,}000}{1{,}000 + 200} = 83.3\%$$

Recall

FOLDER

EMAIL

	Sent to Spam Folder	Sent to Inbox
Spam	100	170
Not Spam	30	700

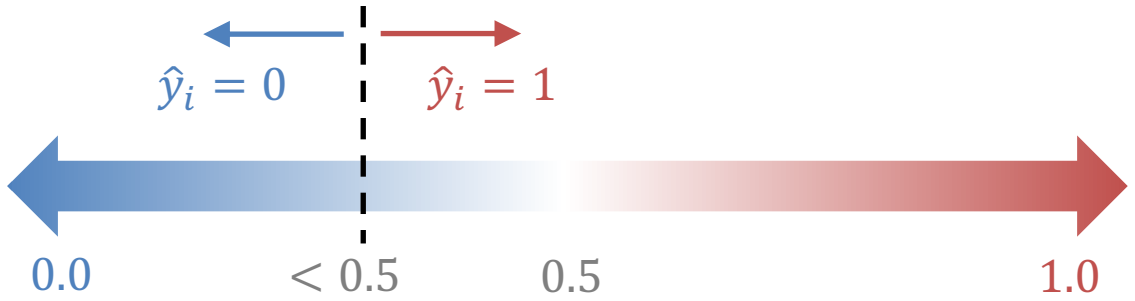
OUT OF ALL THE SPAM E-MAILS, HOW MANY WERE CORRECTLY SENT TO THE SPAM FOLDER?

$$\text{Recall} = \frac{100}{100 + 170} = 37\%$$

Output Probability and Recall Rate

Q: How can we improve recall?

A: We set the threshold, i.e. cut-off probability, to be lower than 0.5.



$$P\left(y = 1 \,\middle|\, \vec{x}_i\right)$$

$$\text{Recall} = \frac{TP}{TP + FN}$$

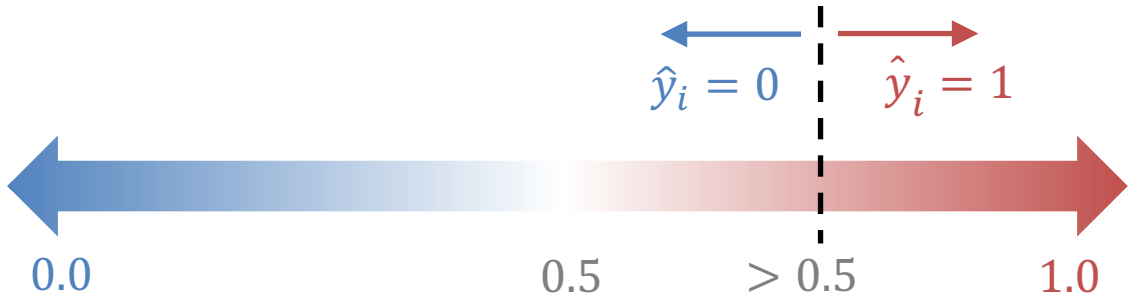
Lower Threshold → Higher TP → Higher Recall.

(also Higher FP)

Output Probability and Precision Rate

Q: How can we improve precision?

A: We set the threshold, i.e. cut-off probability, to be higher than 0.5.



$$P\left(y = 1 \,\middle|\, \vec{x}_i\right)$$

$$\text{Precision} = \frac{TP}{TP + FP}$$

Higher Threshold → Lower FP → Higher Precision.

(also Higher FN)

F_1 Score

MEDICAL MODEL

PRECISION: 55.7%

RECALL: 83.3%

One Score?

SPAM DETECTOR

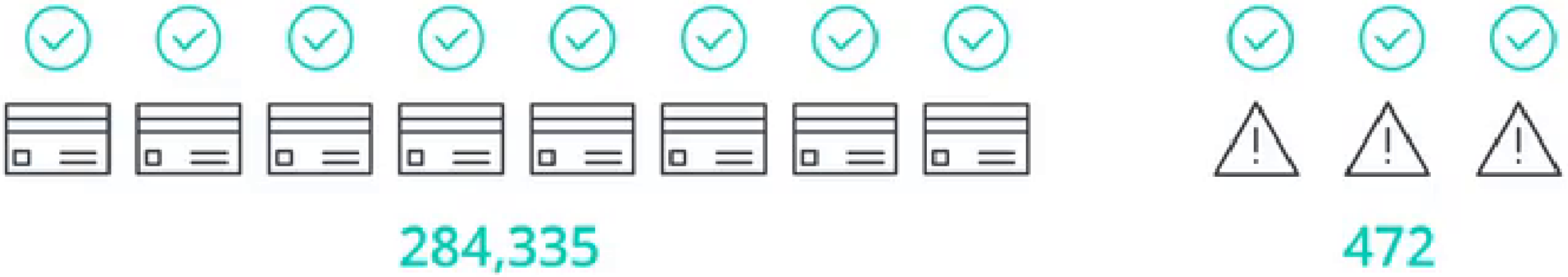
PRECISION: 76.9%

RECALL: 37%

$$F_1 \text{ Score} = \frac{\text{Precision} \times \text{Recall}}{\text{Precision} + \text{Recall}}$$

Geometric Mean

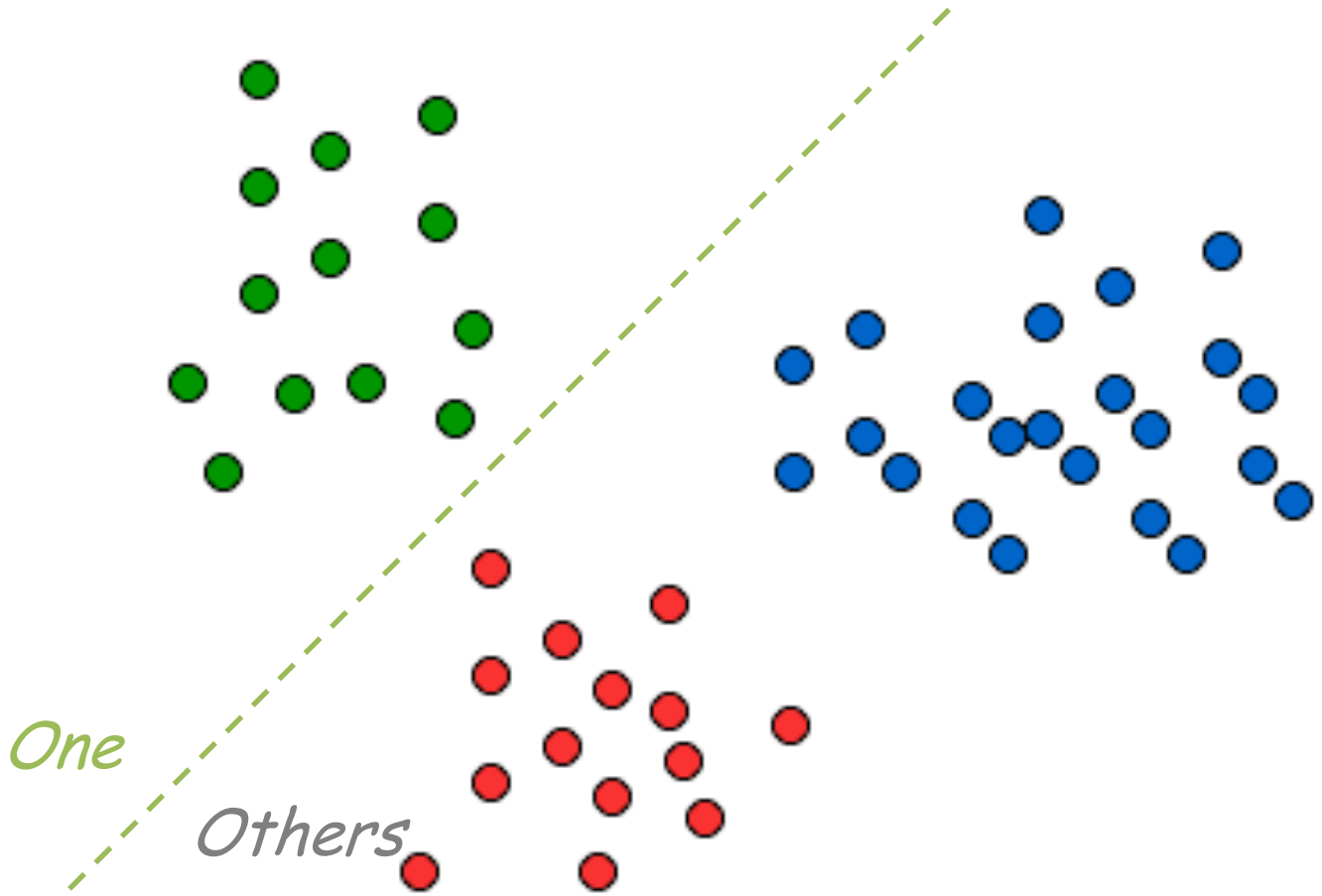
Credit Card Fraud



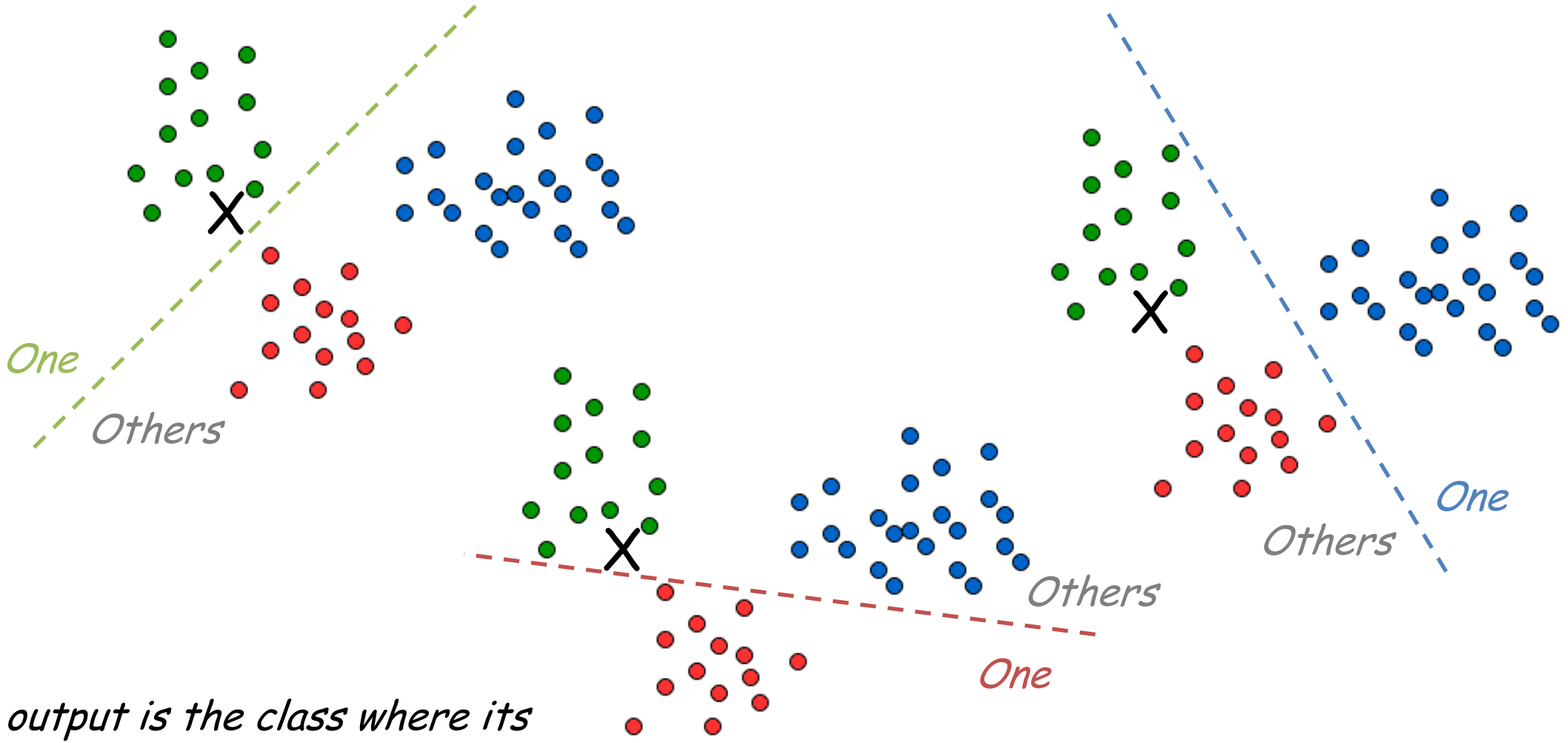
Our model predicted all transaction are good.

$$\text{Prediction} = 100\% \qquad F_1 \text{ Score} = 0 \qquad \text{Recall} = 0\%$$

Multiclass Classification



3 Logistic Classifiers



The output is the class where its output probability is the highest (among the 3 classifiers).

Ex. [0.6, 0.4, 0.1] → Green Class

Pseudocode for Multi-Class Classifiers

```
Function MultiClassClassifier(input x_i, num_classes C, models):
    // Initialize variables to keep track of the highest probability and predicted class
    max_prob = 0
    predicted_class = 0

    // Loop through each class
    For c = 1 to C do:
        // Get the probability that the input belongs to class c using the logistic regression model for that class
        prob = models[c].predict_probability(x_i)

        // If this probability is greater than the current max_prob, update max_prob and predicted_class
        If prob > max_prob then:
            max_prob = prob
            predicted_class = c

    // Return the class with the highest probability
    Return predicted_class
End
```

Summary

- Logistic regression models the probability of a binary outcome by applying a logistic (sigmoid) function to a linear combination of features. This function ensures predicted probabilities are bounded between 0 and 1, making it suitable for classification tasks.
- The decision boundary in logistic regression is defined by a linear combination of features. Data points on either side of this boundary are assigned different class labels, making it useful in applications with linearly separable classes.
- Maximum likelihood estimation (MLE) is used to determine the optimal weights. By maximizing the likelihood (or equivalently, the log-likelihood) of the observed data, logistic regression finds parameters that best explain the class distribution. This is typically achieved through iterative optimization methods like gradient descent.
- Threshold tuning is critical for practical applications. Adjusting the decision threshold allows control over the sensitivity to false positives vs. false negatives. This adjustment makes logistic regression a versatile choice for tasks like fraud detection, medical diagnostics, and risk assessment.
- K-Fold Cross Validation enhances evaluation by splitting data into multiple folds, with each fold used once for validation and the rest for training. This method, often with K=5, maximizes data usage, making it ideal for small datasets. It provides a reliable performance measure by reducing overfitting and ensuring the model generalizes well to unseen data.