Machine Learning Algorithms - Supervised Algorithms

Hao Le Vu

IUH

2025

Outline

- Classification

Logistic Regression: Introduction

Why Logistic Regression?

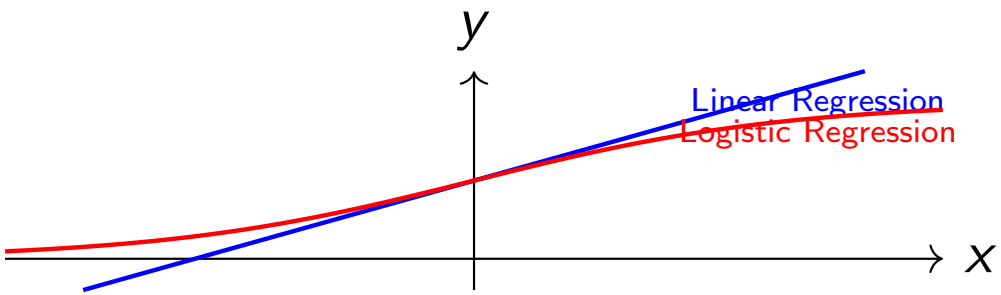
- Linear Regression works well for predicting continuous values.
- But for **classification tasks** (e.g., spam vs. not spam), outputs must be between 0 and 1.
- Logistic Regression predicts probabilities and maps them to discrete classes.

Key Idea

- Use the **sigmoid function** to map linear combinations of inputs to a probability:

$$\sigma(z) = \frac{1}{1 + e^{-z}}$$

Linear vs Logistic Regression



Logistic regression outputs probabilities between 0 and 1.

Logistic Regression: Model

Hypothesis

$$h_\theta(x) = \sigma(\theta^T x) = \frac{1}{1 + e^{-\theta^T x}}$$

where:

- x = feature vector
- θ = model parameters
- $\sigma(z)$ = sigmoid function

Decision Rule

$$\hat{y} = \begin{cases} 1, & \text{if } h_\theta(x) \geq 0.5 \\ 0, & \text{otherwise} \end{cases}$$

Sigmoid Function Visualization

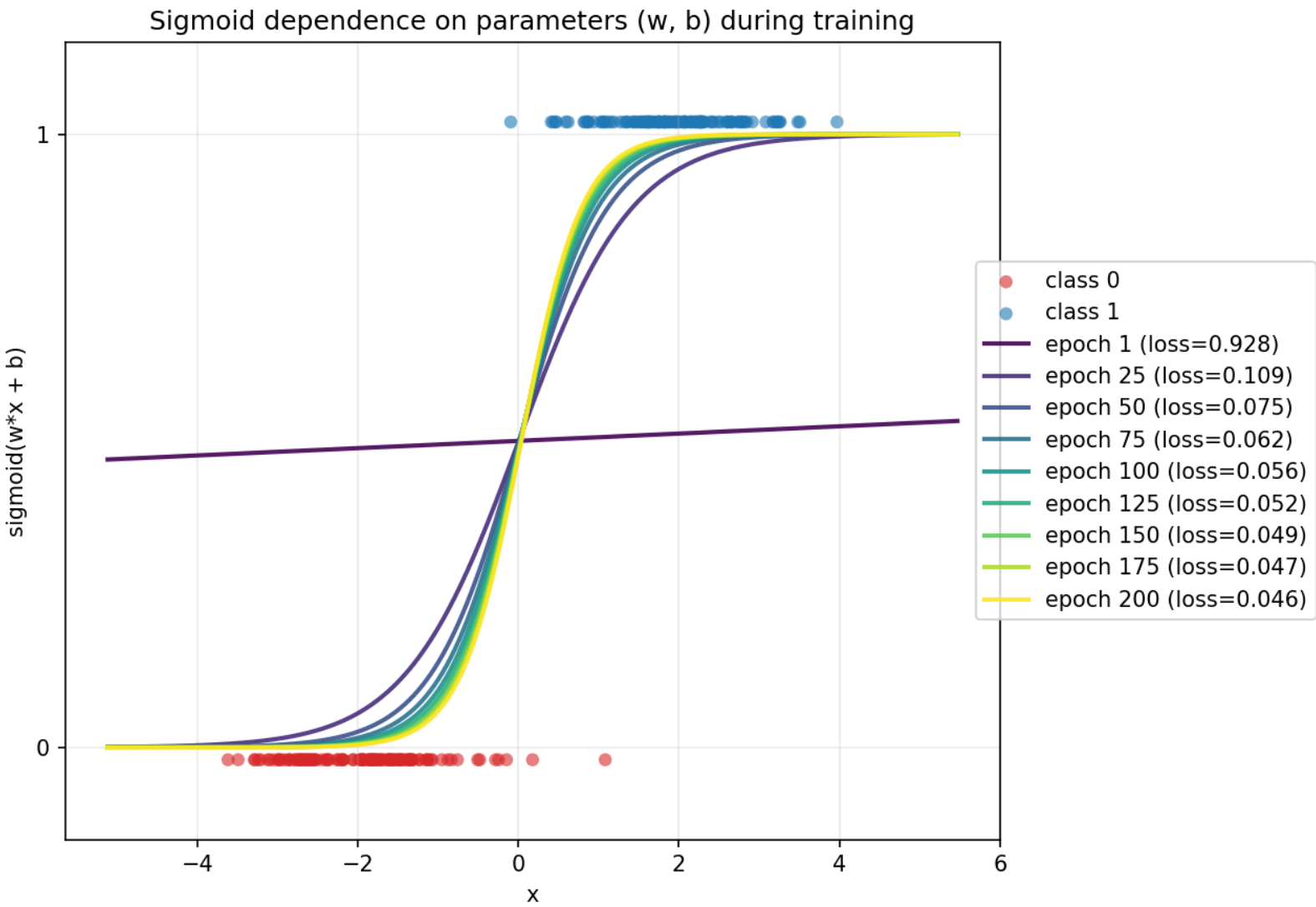


Figure: Sigmoid function

Cost Function and Gradient Descent

Binary Cross-Entropy Loss

$$J(\theta) = -\frac{1}{m}\sum_{i=1}^{m}\left[y^{(i)}\log(h_\theta(x^{(i)})) + (1-y^{(i)})\log(1-h_\theta(x^{(i)}))\right]$$

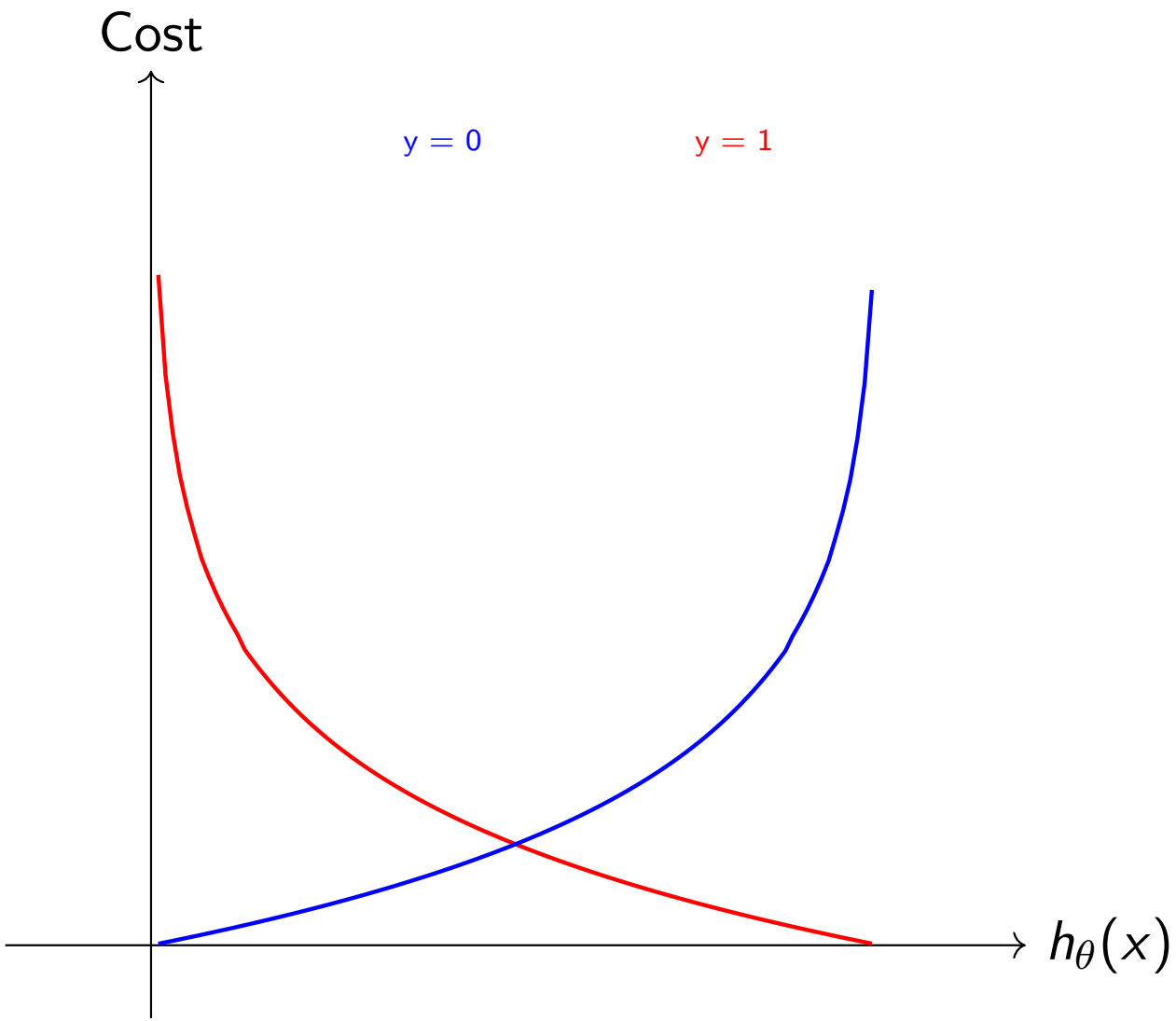
Gradient Descent Update

$$\theta_j := \theta_j - \alpha\frac{\partial J(\theta)}{\partial \theta_j}$$

where:

- α = learning rate
- m = number of training examples

Cost Function Behavior



The cost grows rapidly when predictions are very wrong.

Binary vs Multi-Class Classification

Binary Classification

$$y \in \{0, 1\}, \quad P(y = 1|x) = \sigma(\theta^T x)$$

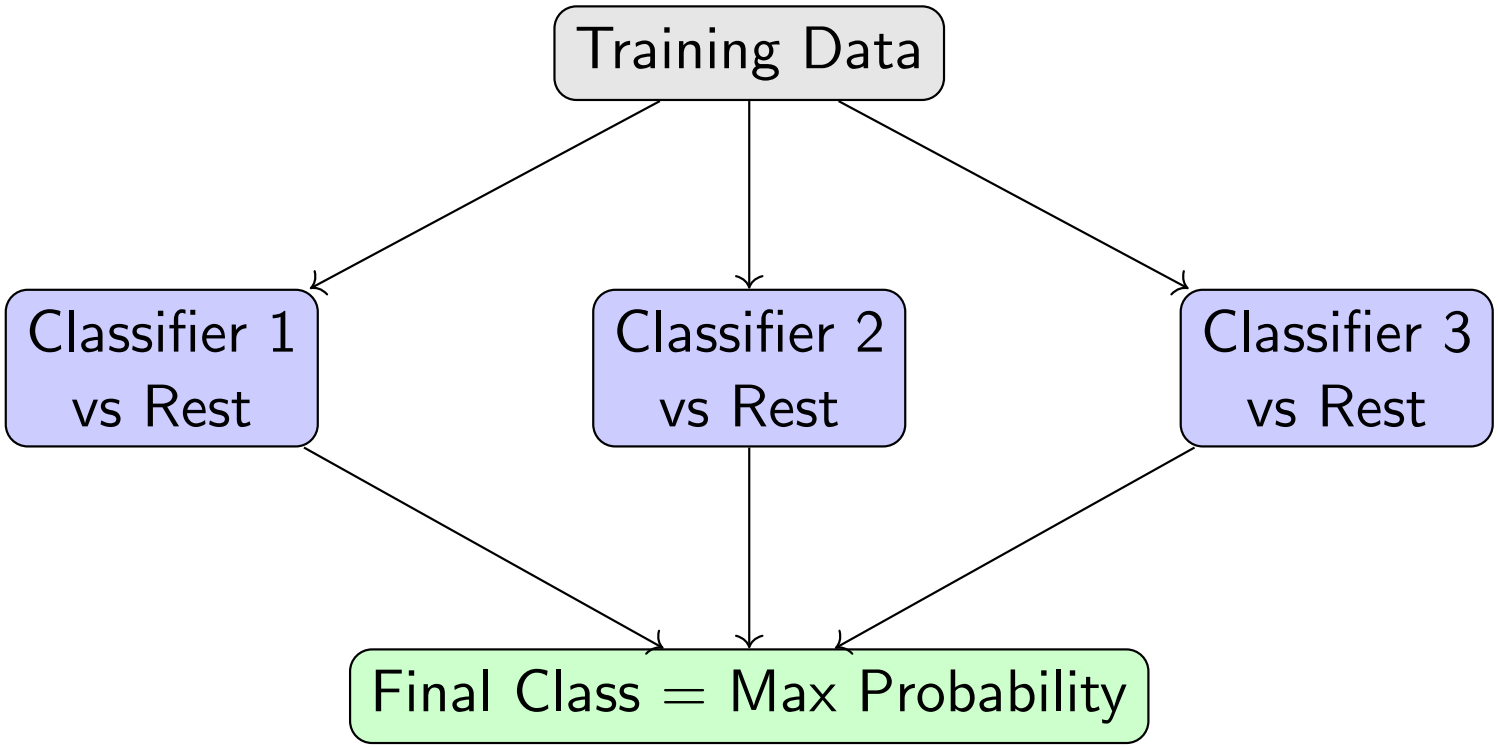
- Decision boundary splits two classes.
- Example: Tumor detection (malignant vs benign).

Multi-Class Classification

- **One-vs-Rest (OvR)**: Train one classifier per class.
- **Softmax Regression**: Generalizes logistic regression:

$$P(y = k|x) = \frac{e^{\theta_k^T x}}{\sum_{j=1}^{K} e^{\theta_j^T x}}$$

One-vs-Rest Multi-Class Visualization



Each classifier predicts its own class probability. The class with the highest probability is chosen.

Evaluation Metric: Accuracy

Definition

$$Accuracy = \frac{\text{Correct Predictions}}{\text{Total Predictions}}$$

Pros and Cons

- **Pros:** Simple and intuitive metric.
- **Cons:** Misleading with imbalanced datasets.

Confusion Matrix Visualization

	Predicted +	Predicted -
Actual +	TP	FN
Actual -	FP	TN

Foundation for calculating Precision, Recall, F1-score.

Precision, Recall, and F1-score

Definitions

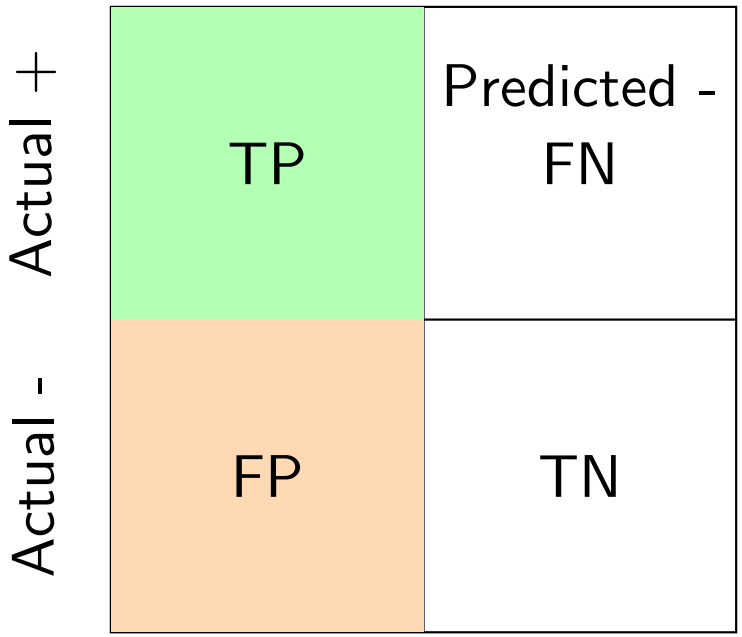
$$Precision = \frac{TP}{TP + FP} \quad ; \quad Recall = \frac{TP}{TP + FN}$$

$$F1 = 2 \times \frac{Precision \times Recall}{Precision + Recall}$$

Interpretation

- **Precision:** How many predicted positives are truly positive.
- **Recall:** How many actual positives are captured.
- **F1-score:** Harmonic mean of Precision and Recall.

Precision vs Recall in Confusion Matrix



Precision focuses on top row (Predicted +). Recall focuses on left column (Actual +).

ROC Curve and AUC

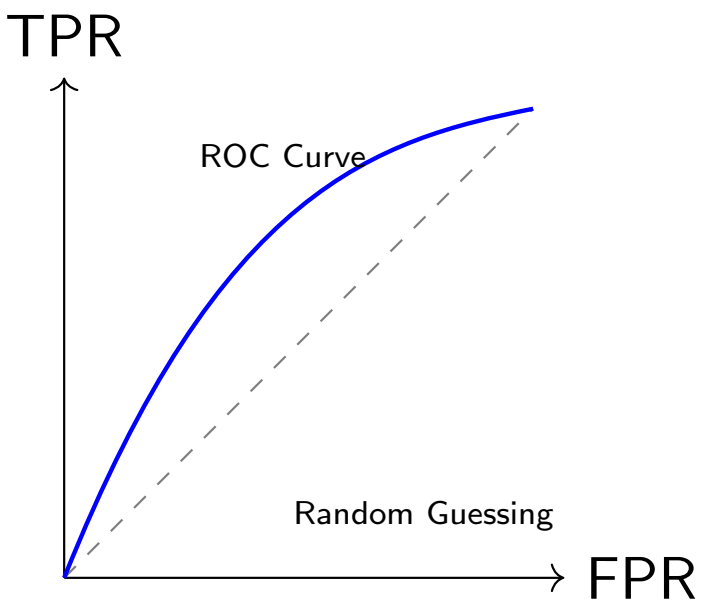
ROC (Receiver Operating Characteristic) Curve

- Plots **True Positive Rate (TPR)** vs. **False Positive Rate (FPR)**.
- $$TPR = \frac{TP}{TP + FN}, \quad FPR = \frac{FP}{FP + TN}$$

AUC (Area Under Curve)

- Measures classifier's ability to distinguish between classes.
- AUC $= 1 \rightarrow$ perfect classifier, AUC $= 0.5 \rightarrow$ random guessing.

ROC Curve Visualization



ROC shows the trade-off between TPR and FPR across thresholds.

Addressing Class Imbalance

Problem

- Many datasets have unequal class distributions.
- Example: Fraud detection (99% normal, 1% fraud).
- Accuracy can be misleading: predicting "all normal" gives 99% accuracy!

Solutions

- **Resampling:** Oversample minority class (SMOTE), or undersample majority class.
- **Adjust decision threshold:** Lower threshold for minority class.
- **Class weighting:** Penalize misclassification of minority class more heavily.
- **Alternative metrics:** Focus on Precision, Recall, F1-score, AUC.

Python Example: Logistic Regression

```python
from sklearn.linear_model import LogisticRegression
import numpy as np

# Training data: Hours studied vs. Pass/Fail outcome
X = np.array([[1], [2], [3], [4], [5], [6]])
y = np.array([0, 0, 0, 1, 1, 1])  # Binary labels

# Train model
model = LogisticRegression()
model.fit(X, y)

# Predict probability for a student who studied 3.5 hours
prob = model.predict_proba([[3.5]])
print(f"Probability of passing: {prob[0][1]:.2f}")
```

Python Example: One-vs-Rest Multi-Class Logistic Regression

```python
from sklearn.datasets import load_iris
from sklearn.linear_model import LogisticRegression
from sklearn.metrics import accuracy_score, classification_report

# Load dataset
iris = load_iris()
X, y = iris.data, iris.target  # 3 classes: Setosa, Versicolor, Virginica

# Train logistic regression model with One-vs-Rest (OvR)
model = LogisticRegression(multi_class='ovr', solver='lbfgs', max_iter=200)
model.fit(X, y)

# Predictions
y_pred = model.predict(X)

# Evaluate model
print("Accuracy:", accuracy_score(y, y_pred))
print("\nClassification Report:")
print(classification_report(y, y_pred, target_names=iris.target_names))
```

Python Example: One-vs-Rest Multi-Class Logistic Regression

where

- `multi_class='ovr'` trains one binary classifier per class.
- Each classifier distinguishes "class vs. rest," and the highest probability wins.
- The Iris dataset is a classic example with 3 classes.

Categorical Data Encoding for Classification

Why Encoding is Needed

Machine learning algorithms work with **numerical data**. Categorical variables must be converted into numeric form for the model to process.

Example:

Sample	Color
1	Red
2	Green
3	Blue
4	Red

Problem: Algorithms can't interpret text values like "Red" or "Blue". We need to encode these categories into numbers.

Label Encoding

Definition

Assign each unique category a unique integer value.

Example: Color Variable

$$\text{Red} = 0, \quad \text{Green} = 1, \quad \text{Blue} = 2$$

Sample	Color	Encoded
1	Red	0
2	Green	1
3	Blue	2
4	Red	0

Pros:

- Simple and space-efficient.

Cons:

- Implies an ordinal relationship that may not exist (e.g., Red ¿ Green ¿ Blue).

One-Hot Encoding

Definition

Creates a new binary feature for each category:

- 1 = presence of category
- 0 = absence of category

Example: Color Variable

Sample	Color	Red	Green	Blue
1	Red	1	0	0
2	Green	0	1	0
3	Blue	0	0	1
4	Red	1	0	0

Pros:

- No implied ordinal relationships.
- Works well for most algorithms (Logistic Regression, Decision Trees, etc.).

Cons:

- Increases dimensionality when there are many unique categories.

Target Encoding

Definition

Replace each category with the **mean of the target variable** for that category.

Example: Predicting Purchase (0 or 1)

Color	Target Mean (Purchase Rate)
Red	0.75
Green	0.40
Blue	0.20

Pros:

- Reduces dimensionality (1 column instead of many).
- Encodes useful information about relationship with target.

Cons:

- Can lead to data leakage if not used carefully (must compute on training set only).

Comparison of Encoding Methods

Method	Pros	Cons	Use Cases
Label Encoding	Simple, space-efficient	Implies ordinal relationship	Tree-based models like Decision Trees, Random Forests
One-Hot Encoding	No ordinal assumptions, widely used	High dimensionality for many categories	Logistic Regression, SVM, Neural Networks
Target Encoding	Compact, includes target info	Risk of data leakage	High-cardinality categorical features

Key Tip: Choose the encoding method based on the model type and the number of unique categories.

Python Example: Encoding with Scikit-Learn

```python
import pandas as pd
from sklearn.preprocessing import LabelEncoder, OneHotEncoder

# Sample data
df = pd.DataFrame({'Color': ['Red', 'Green', 'Blue', 'Red']})

# Label Encoding
label_encoder = LabelEncoder()
df['Color_Label'] = label_encoder.fit_transform(df['Color'])

# One-Hot Encoding
one_hot = pd.get_dummies(df['Color'], prefix='Color')

# Combine results
result = pd.concat([df, one_hot], axis=1)
print(result)
```

k-Nearest Neighbors (k-NN): Introduction

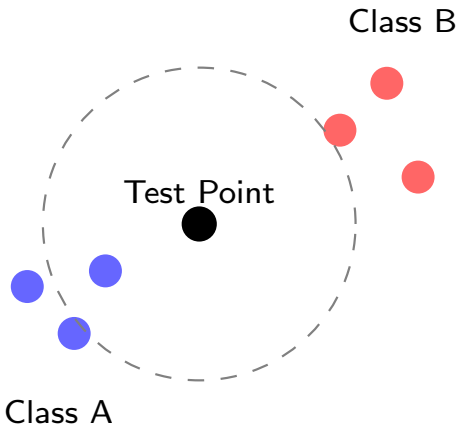
Definition

- A **non-parametric supervised learning algorithm** used for both **classification** and **regression**.
- Predicts the label or value of a new data point by finding the **k closest points** (neighbors) in the training data.
- It is a **lazy learner**, meaning no explicit model is built during training.

Core Intuition

- **Classification**: Majority vote among the k neighbors determines the predicted class.
- **Regression**: Average of the neighbors' target values becomes the prediction.

k-NN Intuition: Visual Example



The algorithm predicts based on the majority label within the dashed circle.

Choosing the Hyperparameter k

Role of k

$$k = \text{Number of nearest neighbors considered}$$

- Controls the complexity and smoothness of the decision boundary.
- **Small k:** Highly sensitive to noise $\rightarrow$ risk of overfitting.
- **Large k:** Smooth boundary $\rightarrow$ risk of underfitting.

Modes of k-NN

- **Classification**: Predict the most common class among neighbors.
- **Regression**: Predict the average target value of neighbors.

How k-NN Works: Step-by-Step

Algorithm Steps

1. Choose the number of neighbors k.
2. Compute the distance between the test point and all training points.
3. Sort and select the k nearest neighbors.
4. Make prediction:
   - **Classification:** Assign the majority class.
   - **Regression:** Predict the mean of target values.

k-NN has no explicit training phase; all computation happens at prediction time.

How k-NN Works: Step-by-Step

Algorithm Steps

1. Choose the number of neighbors k.
2. Compute the distance between the test point and all training points.
3. Sort and select the k nearest neighbors.
4. Make prediction:
   - **Classification:** Assign the majority class.
   - **Regression:** Predict the mean of target values.

k-NN has no explicit training phase; all computation happens at prediction time.

Common Distance Metrics in k-NN

Euclidean Distance (most common)

$$d(p, q) = \sqrt{\sum_{i=1}^{n} (p_i - q_i)^2}$$

Manhattan Distance

$$d(p, q) = \sum_{i=1}^{n} |p_i - q_i|$$

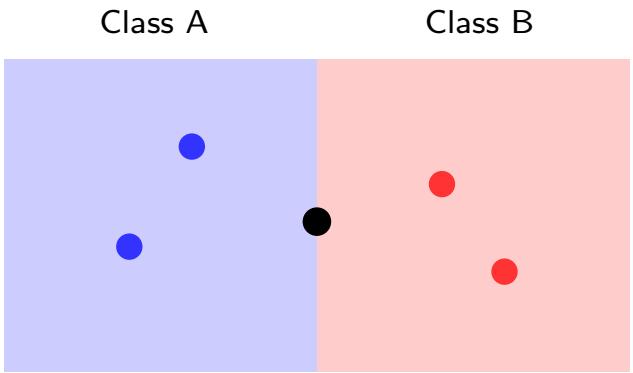
Minkowski Distance (general form)

$$d(p, q) = \left(\sum_{i=1}^{n} |p_i - q_i|^m \right)^{\frac{1}{m}}$$

k-NN: Decision Boundaries

Intuition

- The algorithm divides the feature space into **regions**.
- Each region is dominated by the majority class of its neighbors.
- $k = 1$: Highly complex, jagged boundaries.
- Larger k: Smoother, more generalized boundaries.



Boundaries become smoother as k increases.

Effect of Small vs Large k

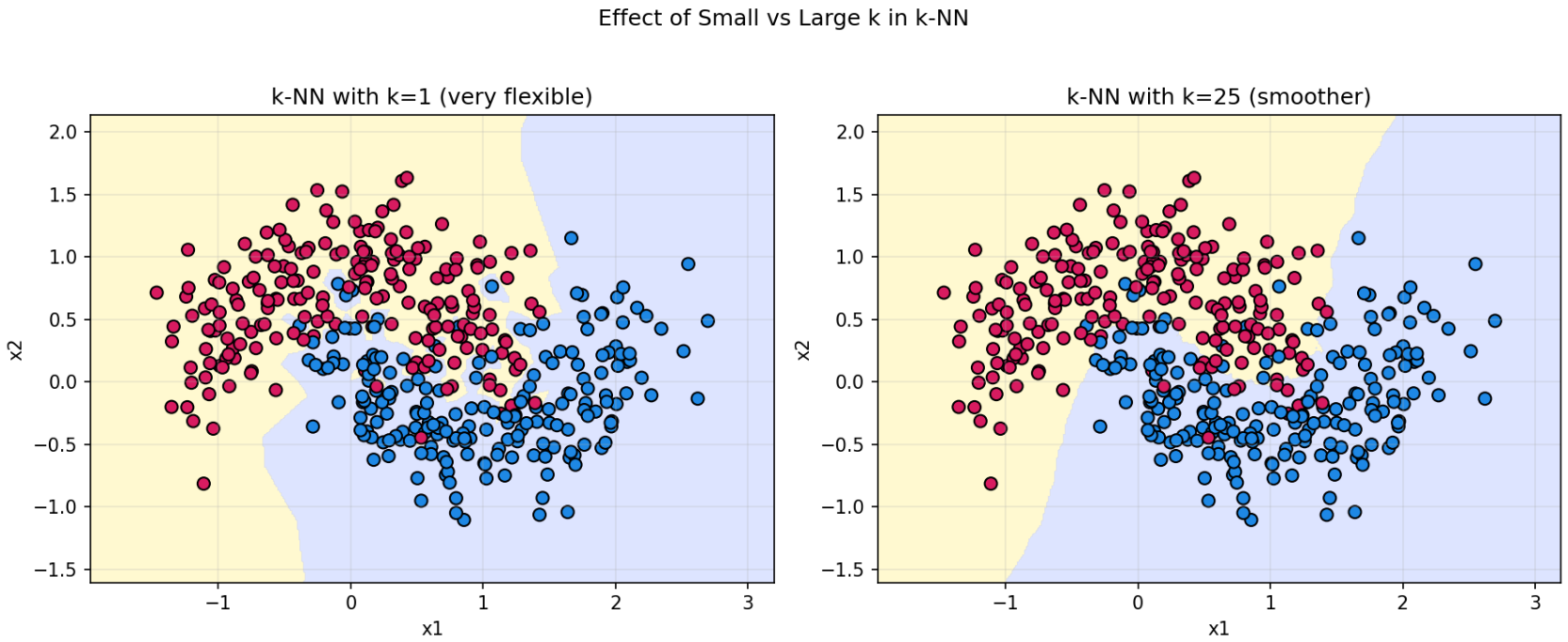


Figure: Small $k \rightarrow$ very localized and complex boundary. Large $k \rightarrow$ smoother boundary but less sensitive to details.

k-NN: Strengths and Weaknesses

Advantages

- Simple and intuitive, easy to implement.
- No explicit training step required.
- Naturally supports multi-class problems.
- Can be applied to both classification and regression tasks.

Limitations

- Computationally expensive for large datasets.
- Performance degrades with high-dimensional data (**curse of dimensionality**).
- Sensitive to noisy or irrelevant features.
- Requires careful tuning of k and distance metric.

Python Example: k-NN Classification

```python
from sklearn.neighbors import KNeighborsClassifier
import numpy as np

# Training data: [Feature1, Feature2]
X = np.array([
    [1.2, 1.0], [1.5, 1.3], [1.1, 0.9],
    [3.5, 3.0], [3.8, 3.2], [4.0, 2.9]
])
y = np.array([0, 0, 0, 1, 1, 1])  # Class labels

# Train k-NN model
model = KNeighborsClassifier(n_neighbors=3)
model.fit(X, y)

# Predict for a new point
test_point = np.array([[2.0, 2.0]])
prediction = model.predict(test_point)
print(f"Predicted Class: {prediction[0]}")
```