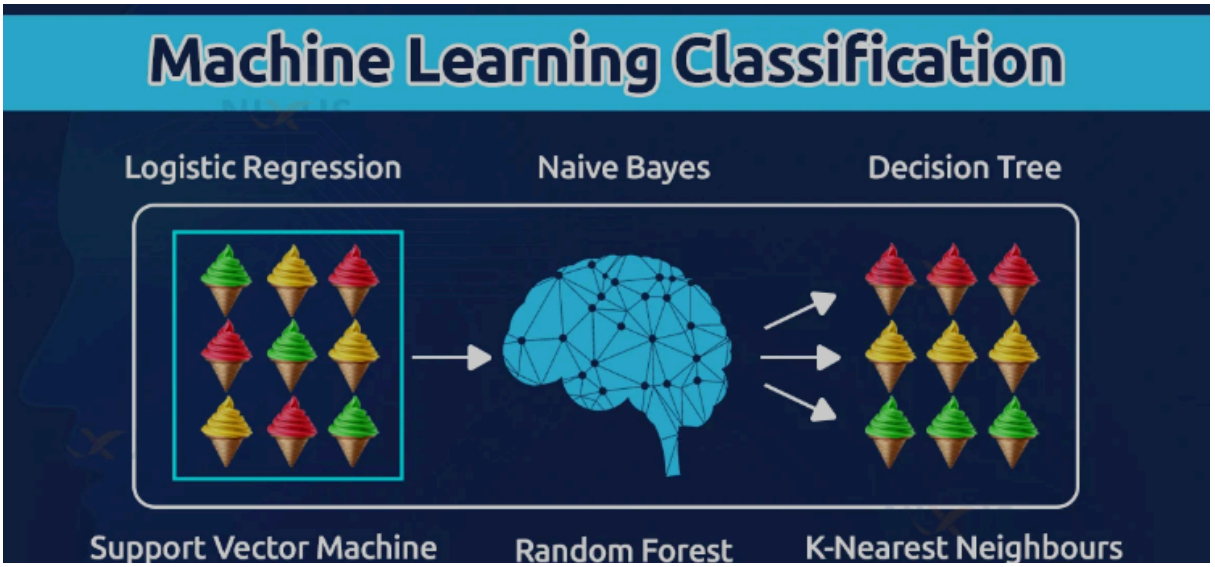


MOOC 3: Supervised Machine Learning - Classification

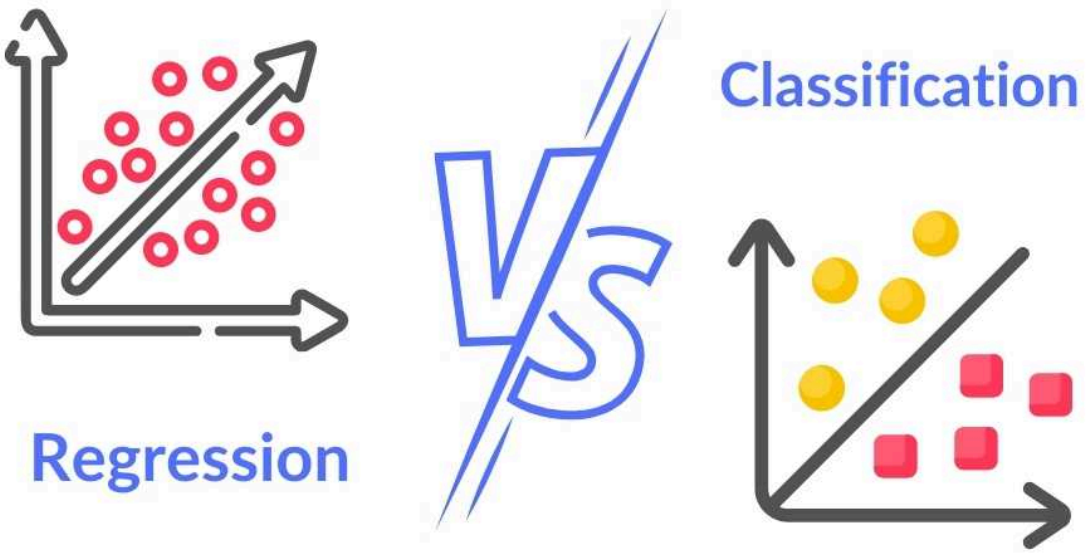
🔗 **View Original Document**: https://www.notion.so/MOOC-3-Supervised-Machine-Learning-Classification-28af95c1eb348053b709c12fb3731671?source=copy_link

---

Module 1: Logistic Regression

Giới thiệu Classification Problems

Classification là bài toán dự đoán biến mục tiêu có giá trị **phân loại (categorical)**, khác với Regression dự đoán giá trị liên tục.



Hai loại chính của Supervised Learning

1. Regression (Hồi quy)

- Target variable là **liên tục (continuous)**
- Ví dụ: Dự đoán giá nhà, nhiệt độ, doanh thu

2. Classification (Phân loại)

- Target variable là **phân loại (categorical)**
- Ví dụ: Email spam/không spam, chẩn đoán bệnh, nhận diện hình ảnh

Các thuật toán Classification phổ biến

Thuật toán	Đặc điểm	Use Case
Logistic Regression	Dự đoán xác suất, high interpretability	Binary classification, baseline model
K-Nearest Neighbors	Distance-based, simple	Small datasets, nonlinear boundaries
Support Vector Machines	Tìm hyperplane tối ưu	High-dimensional data, clear margin
Decision Trees	Rule-based, interpretable	Feature interactions, nonlinear
Random Forests	Ensemble of trees	Robust, reduce overfitting

Thuật toán	Đặc điểm	Use Case
Boosting	Sequential ensemble	High accuracy, competition

Logistic Regression

Logistic Regression mô hình hóa xác suất của một class xảy ra dựa trên các biến độc lập. Mặc dù có tên là "regression" nhưng thực chất là thuật toán **classification**.

Công thức toán học

Logistic (Sigmoid) Function:

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

Linear Combination:

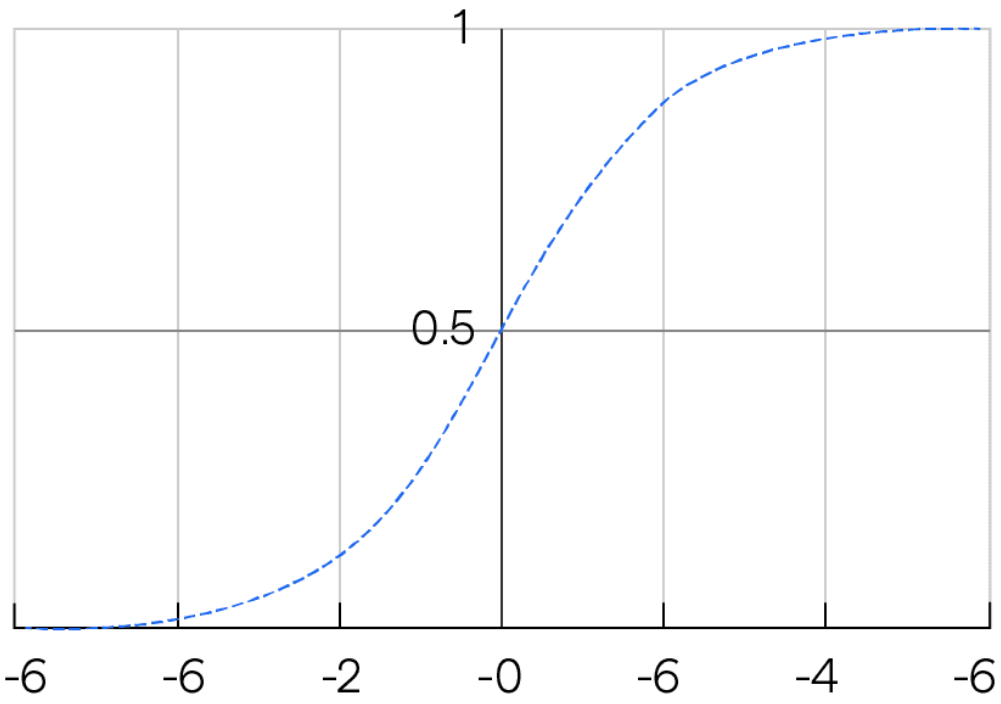
$$z = \beta_0 + \beta_1 x_1 + \beta_2 x_2 + ... + \beta_n x_n = \beta^T x$$

Probability Prediction:

$$P(y = 1|x) = \sigma(z) = \frac{1}{1 + e^{-\beta^T x}}$$

Sigmoid Function

$$f(x) = \frac{1}{1 + e^{-fx}}$$



Đặc điểm của Sigmoid Function

- **Output range**: 0 đến 1 (phù hợp cho xác suất)
- **S-shaped curve**: Smooth, differentiable
- **Threshold**: Thường dùng 0.5 để phân loại
  - P(y=1) ≥ 0.5 → Class 1
  - P(y=1) < 0.5 → Class 0

Cost Function (Binary Cross-Entropy)

Không thể dùng MSE vì **non-convex** với sigmoid function!

Binary Cross-Entropy Loss:

$$J(\beta) = -\frac{1}{m}\sum_{i=1}^{m}[y_i \log(\hat{y}_i) + (1-y_i)\log(1-\hat{y}_i)]$$

Trong đó:

- m = số samples
- y_i = actual label (0 hoặc 1)
- $\hat{y}_i$ = predicted probability

Đặc điểm:

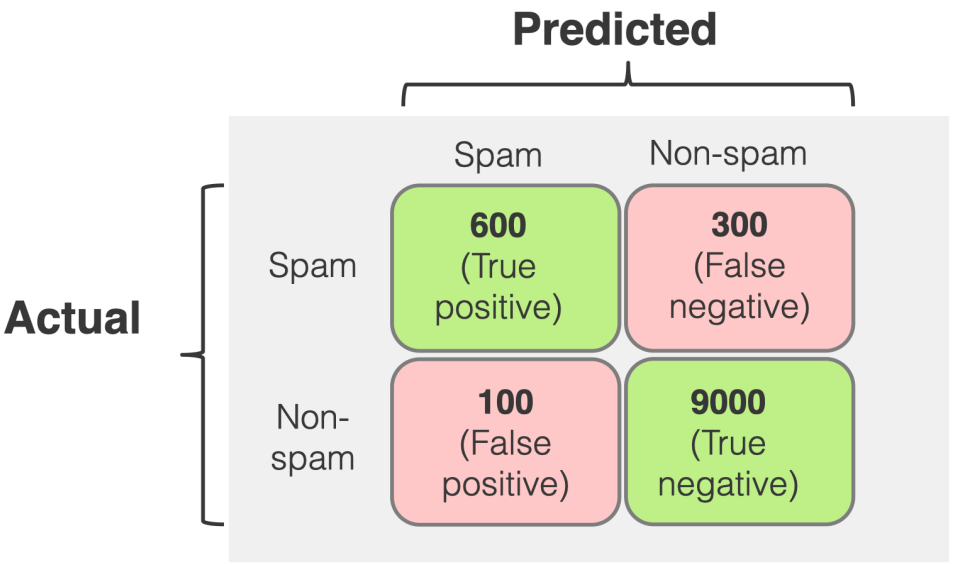
- **Convex function** → Có global minimum
- **Penalize wrong predictions heavily**
- **Differentiable** → Có thể dùng gradient descent

Classification Metrics

Accuracy không phải lúc nào cũng đủ! Với imbalanced data, cần dùng nhiều metrics khác nhau để đánh giá toàn diện.

Confusion Matrix

	Predicted Positive	**Predicted Negative**
Actual Positive	True Positive (TP)	False Negative (FN)
Actual Negative	False Positive (FP)	True Negative (TN)



Các Metrics quan trọng

1. Accuracy (Độ chính xác)

$$\text{Accuracy} = \frac{TP + TN}{TP + TN + FP + FN}$$

- Tỷ lệ predictions đúng
- **Vấn đề**: Không phù hợp với imbalanced data

2. Precision (Độ chính xác dương tính)

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

- "Trong những cái model dự đoán là Positive, bao nhiêu cái đúng?"
- **Use case**: Khi False Positive tốn kém (spam detection)

3. Recall / Sensitivity (Độ nhạy)

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

- "Trong những cái thực tế là Positive, model bắt được bao nhiêu?"
- **Use case**: Khi False Negative nguy hiểm (disease detection)

4. F1-Score

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

- **Balanced metric** cho cả Precision và Recall
- **Use case**: Imbalanced data

ROC Curve & AUC

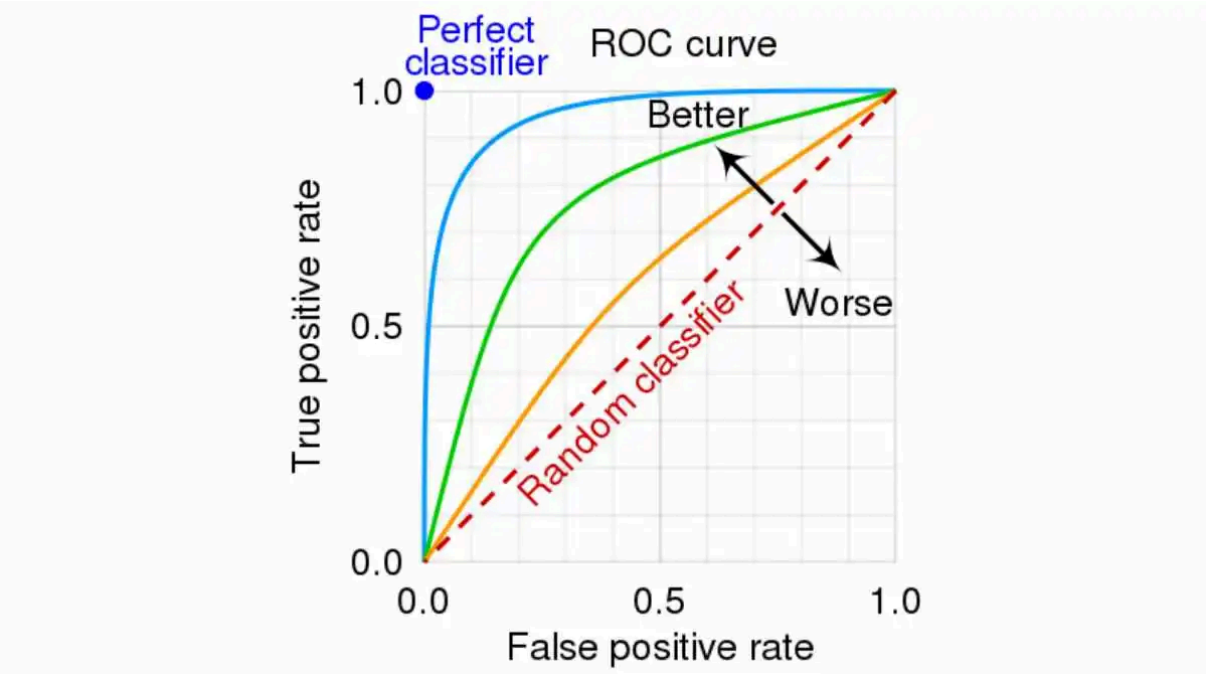
ROC Curve plots True Positive Rate (Sensitivity) vs False Positive Rate (1-Specificity) tại các threshold khác nhau.

AUC (Area Under the Curve):

- **AUC = 1.0**: Perfect classifier
- **AUC = 0.5**: Random classifier
- **AUC > 0.8**: Generally good

Khi nào dùng:

- **ROC Curve**: Balanced classes
- **Precision-Recall Curve**: Imbalanced classes



Code Implementation - Logistic Regression

Train Model và Evaluate

```
from sklearn.linear_model import LogisticRegression
from sklearn.model_selection import train_test_split
from sklearn.metrics import classification_report, confusion_matrix
from sklearn.metrics import accuracy_score, precision_score, recall_score,
f1_score
from sklearn.preprocessing import StandardScaler
import numpy as np

# Load và prepare data
from sklearn.datasets import load_breast_cancer
data = load_breast_cancer()
X, y = data.data, data.target

# Scale features
scaler = StandardScaler()
X_scaled = scaler.fit_transform(X)

# Split data
X_train, X_test, y_train, y_test = train_test_split(
    X_scaled, y, test_size=0.2, random_state=42, stratify=y
)

# Train Logistic Regression
lr = LogisticRegression(max_iter=10000, random_state=42)
lr.fit(X_train, y_train)

# Predictions
y_pred = lr.predict(X_test)
y_pred_proba = lr.predict_proba(X_test)[:, 1]

# Evaluate
print("Accuracy:", accuracy_score(y_test, y_pred))
print("Precision:", precision_score(y_test, y_pred))
```

```
print("Recall:", recall_score(y_test, y_pred))
print("F1-Score:", f1_score(y_test, y_pred))
print("\nClassification Report:")
print(classification_report(y_test, y_pred))
print("\nConfusion Matrix:")
print(confusion_matrix(y_test, y_pred))
```

ROC Curve Visualization

```
from sklearn.metrics import roc_curve, roc_auc_score
import matplotlib.pyplot as plt

# Calculate ROC
fpr, tpr, thresholds = roc_curve(y_test, y_pred_proba)
auc_score = roc_auc_score(y_test, y_pred_proba)

# Plot
plt.figure(figsize=(10, 6))
plt.plot(fpr, tpr, 'darkorange', lw=2, label=f'ROC (AUC = {auc_score:.3f})')
plt.plot([0, 1], [0, 1], 'navy', lw=2, linestyle='--', label='Random')
plt.xlabel('False Positive Rate')
plt.ylabel('True Positive Rate')
plt.title('ROC Curve')
plt.legend()
plt.grid(True, alpha=0.3)
plt.show()
```

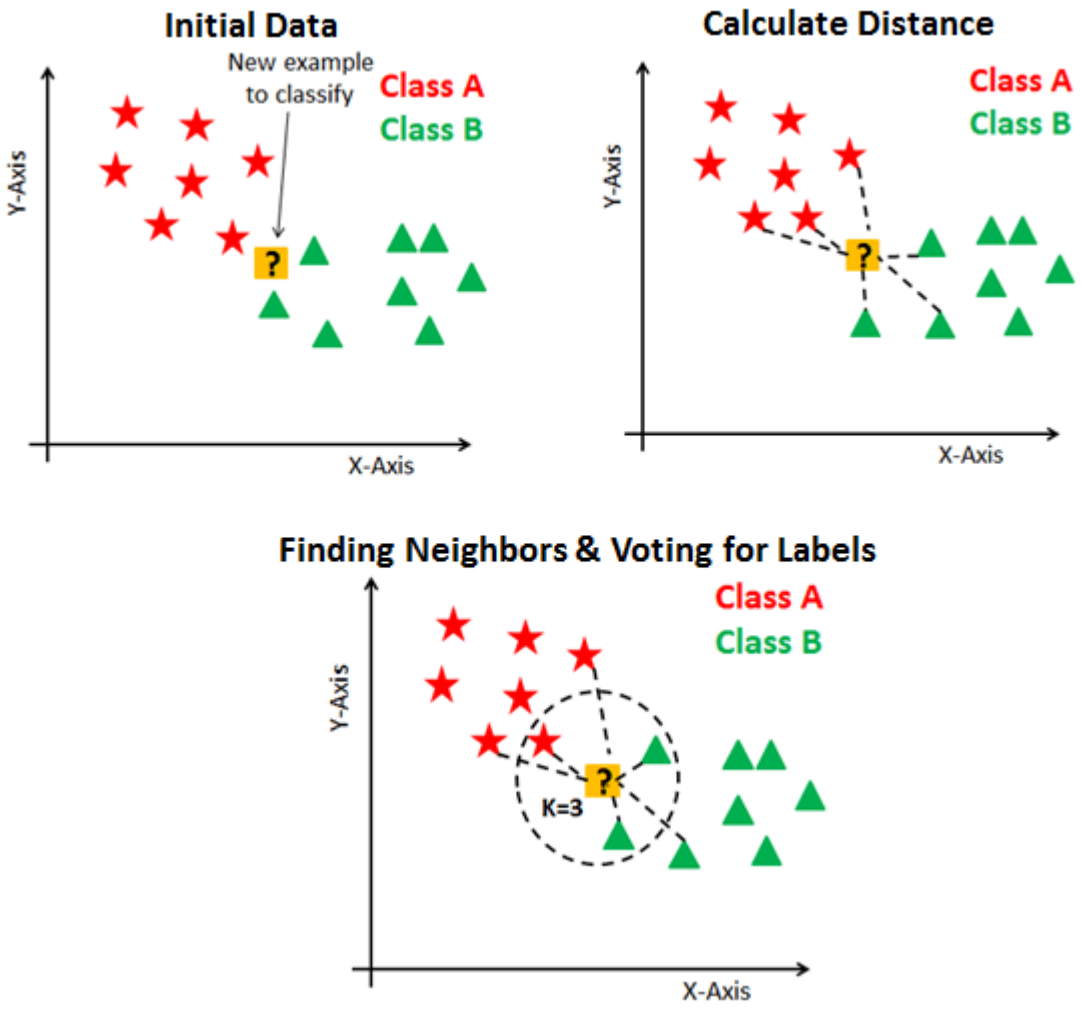
Module 2: K-Nearest Neighbors (KNN)

K-Nearest Neighbors là thuật toán **instance-based learning** đơn giản nhưng hiệu quả. Ý tưởng: *"Show me your friends, and I'll tell you who you are!"*

Nguyên lý hoạt động

KNN Algorithm:

1. **Chọn K** (số neighbors)
2. **Tính distance** từ điểm mới đến tất cả training points
3. **Tìm K nearest neighbors**
4. **Vote**: Class nào có nhiều neighbors nhất → Predict class đó



Distance Metrics

1. Euclidean Distance (Phổ biến nhất)

$$d(x, x') = \sqrt{\sum_{i=1}^{n}(x_i - x_i')^2}$$

2. Manhattan Distance

$$d(x, x') = \sum_{i=1}^{n} |x_i - x'_i|$$

3. Minkowski Distance (Generalized)

$$d(x, x') = \left(\sum_{i=1}^{n} |x_i - x'_i|^p \right)^{1/p}$$

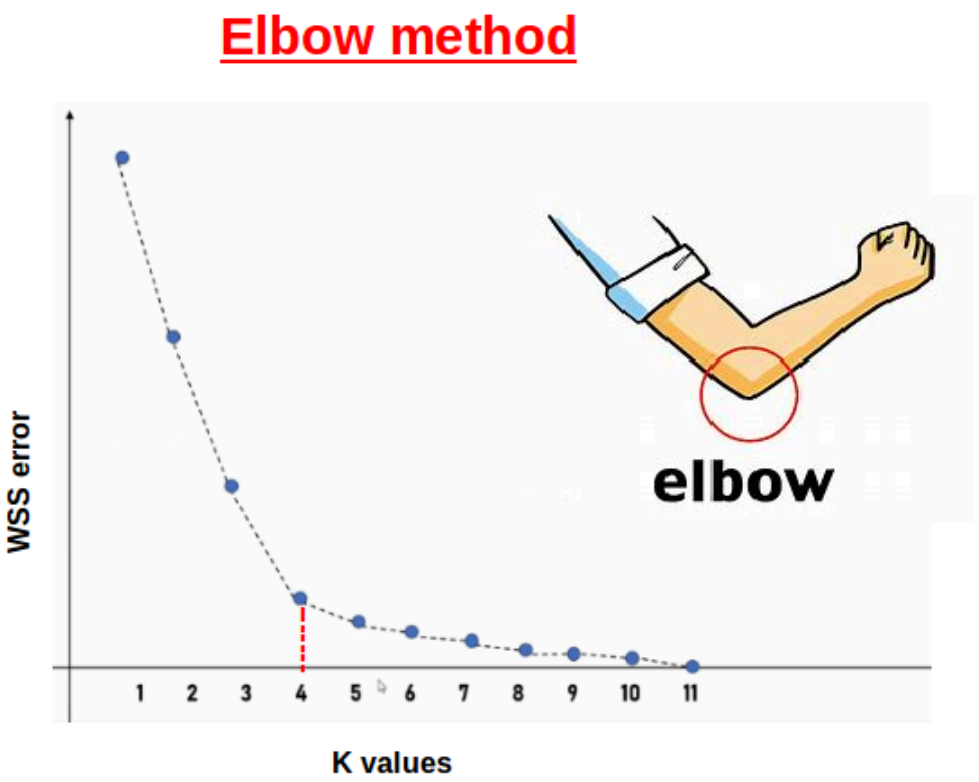
- p=1: Manhattan
- p=2: Euclidean

Chọn K tối ưu: Elbow Method

Elbow Method: Test nhiều K values và plot error rate. Chọn K tại "elbow" (điểm gấp khúc).

Trade-offs:

- **K nhỏ**: Flexible, capture local patterns ➜ Risk overfitting
- **K lớn**: Smooth boundary, reduce noise ➜ Risk underfitting



Ưu và Nhược điểm

Ưu điểm:

- Simple, intuitive
- No training phase (lazy learning)
- Good with nonlinear data

Nhược điểm:

- **Slow prediction** với large datasets
- **Memory intensive**
- **Requires feature scaling**
- **Curse of dimensionality**

Code Implementation - KNN

```
from sklearn.neighbors import KNeighborsClassifier
from sklearn.datasets import load_iris
from sklearn.model_selection import cross_val_score
import matplotlib.pyplot as plt

# Load data
iris = load_iris()
X, y = iris.data, iris.target

# Scale features (IMPORTANT!)
from sklearn.preprocessing import StandardScaler
scaler = StandardScaler()
X_scaled = scaler.fit_transform(X)

# Split
X_train, X_test, y_train, y_test = train_test_split(
    X_scaled, y, test_size=0.2, random_state=42
)

# KNN với K=5
knn = KNeighborsClassifier(n_neighbors=5)
knn.fit(X_train, y_train)

# Evaluate
print(f"Training Accuracy: {knn.score(X_train, y_train):.4f}")
print(f"Test Accuracy: {knn.score(X_test, y_test):.4f}")
```

Finding Optimal K

```
# Test K từ 1 đến 30
k_values = range(1, 31)
train_scores = []
test_scores = []

for k in k_values:
    knn = KNeighborsClassifier(n_neighbors=k)
    knn.fit(X_train, y_train)
```

```
    train_scores.append(knn.score(X_train, y_train))
    test_scores.append(knn.score(X_test, y_test))

# Plot Elbow
plt.figure(figsize=(12, 5))
plt.plot(k_values, train_scores, 'o-', label='Training Score')
plt.plot(k_values, test_scores, 's-', label='Test Score')
plt.xlabel('K (Number of Neighbors)')
plt.ylabel('Accuracy')
plt.title('KNN: Choosing Optimal K')
plt.legend()
plt.grid(True, alpha=0.3)
plt.show()

# Best K
best_k = k_values[np.argmax(test_scores)]
print(f"Optimal K: {best_k}")
```

Module 3: Support Vector Machines (SVM)

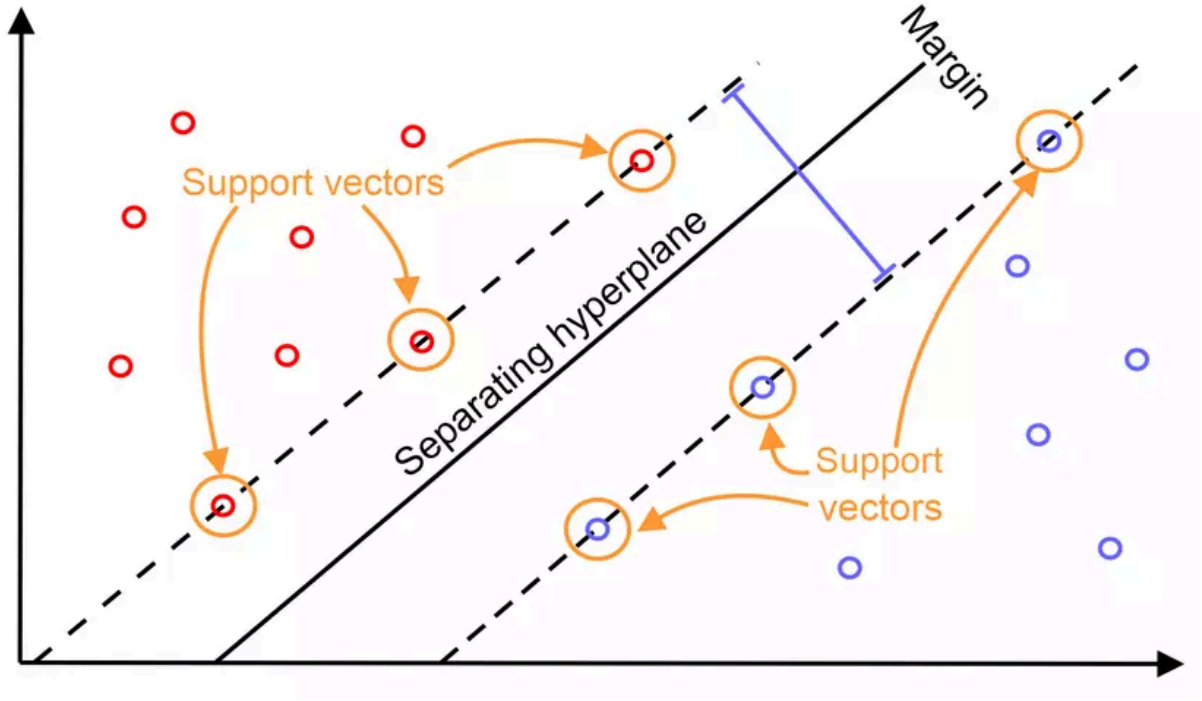
Support Vector Machines tìm **hyperplane tối ưu** để phân tách classes bằng cách **maximize margin** giữa các classes.

Ý tưởng cơ bản

Mục tiêu: Tìm decision boundary sao cho:

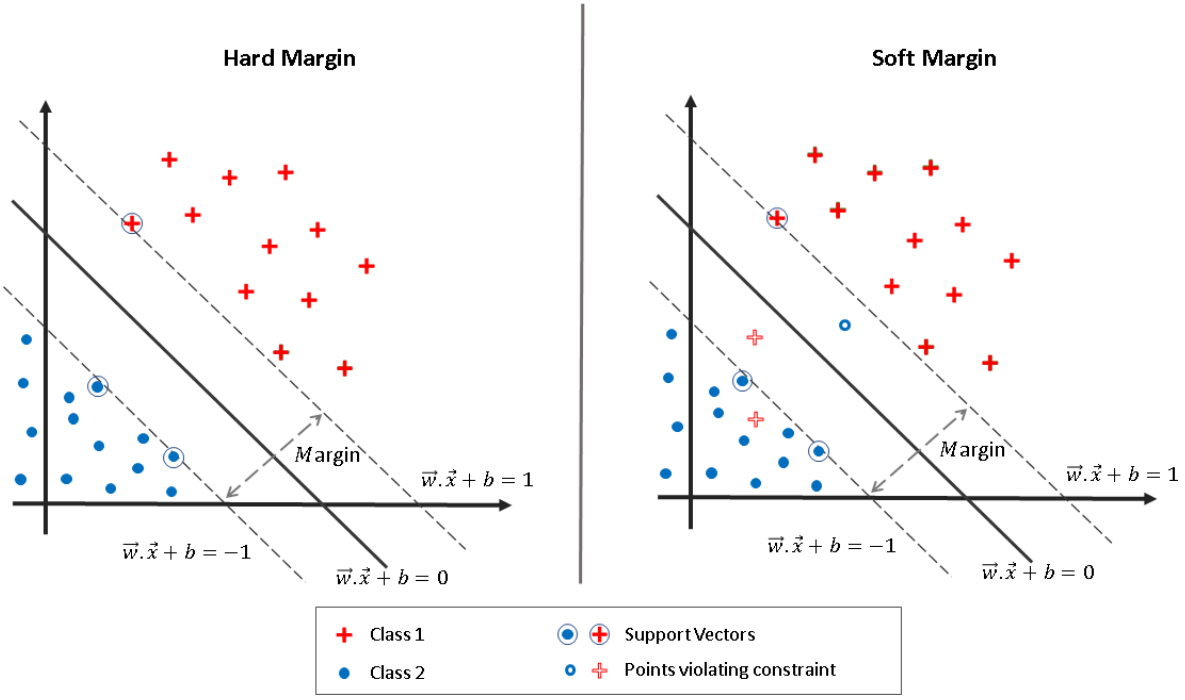
1. Phân tách classes chính xác
2. **Maximize margin** (khoảng cách đến nearest points)

Support Vectors: Những điểm gần decision boundary nhất



Hard vs Soft Margin SVM

Type	Đặc điểm	Use Case
Hard Margin	Data phải linearly separable, No errors	Hiếm dùng
Soft Margin	Cho phép misclassifications, Parameter C	Phổ biến ⭐



Cost Function: Hinge Loss

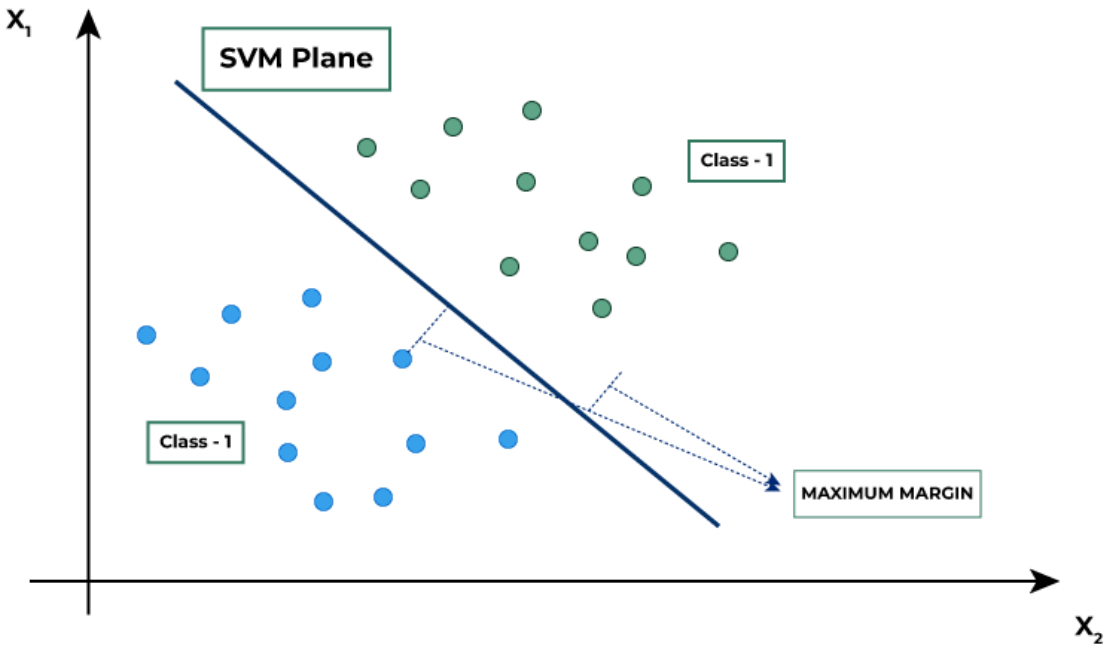
Hinge Loss penalize misclassifications và points too close to decision boundary.

Hinge Loss Formula:

$$L(y, f(x)) = \max(0, 1 - y \cdot f(x))$$

SVM Objective:

$$\min_{w,b} \left[\frac{1}{2}||w||^2 + C \sum_{i=1}^{m} \max(0, 1 - y_i(w^T x_i + b)) \right]$$



Kernel Trick

Kernel Trick transform data sang higher-dimensional space để data trở nên linearly separable!

Popular Kernels:

1. Linear Kernel

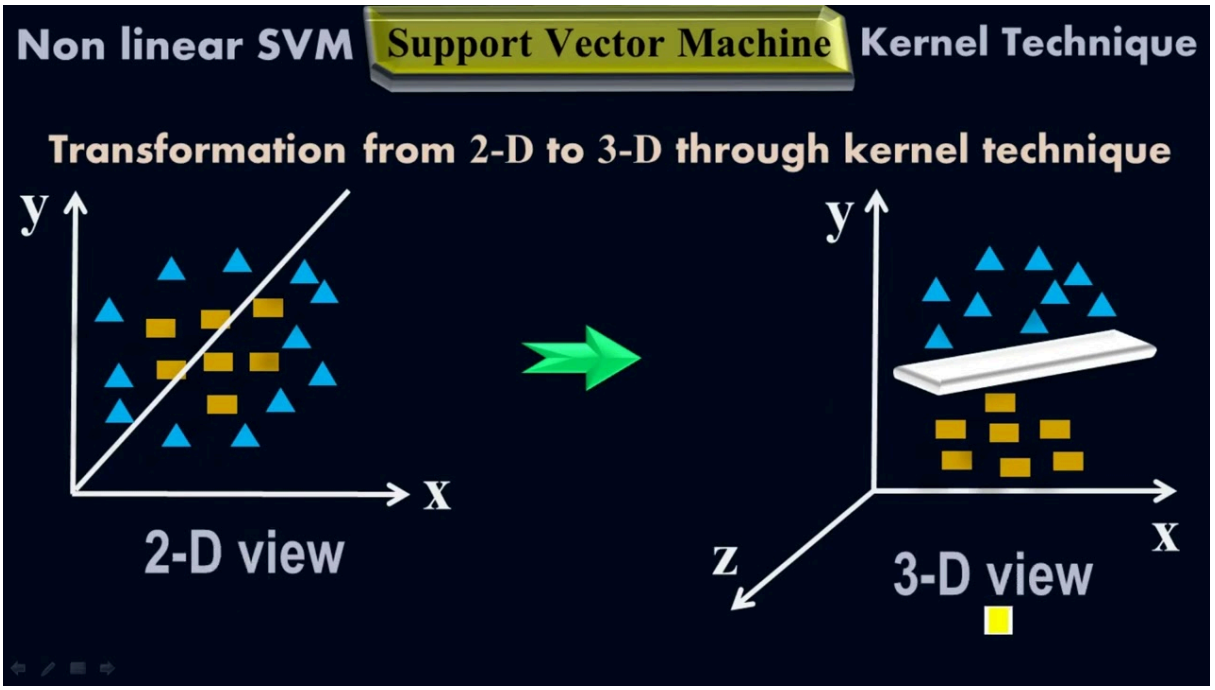
$$K(x, x') = x^T x'$$

2. Polynomial Kernel

$$K(x, x') = (x^T x' + c)^d$$

3. RBF (Gaussian) Kernel Most Popular

$$K(x, x') = \exp\left(-\gamma ||x - x'||^2\right)$$



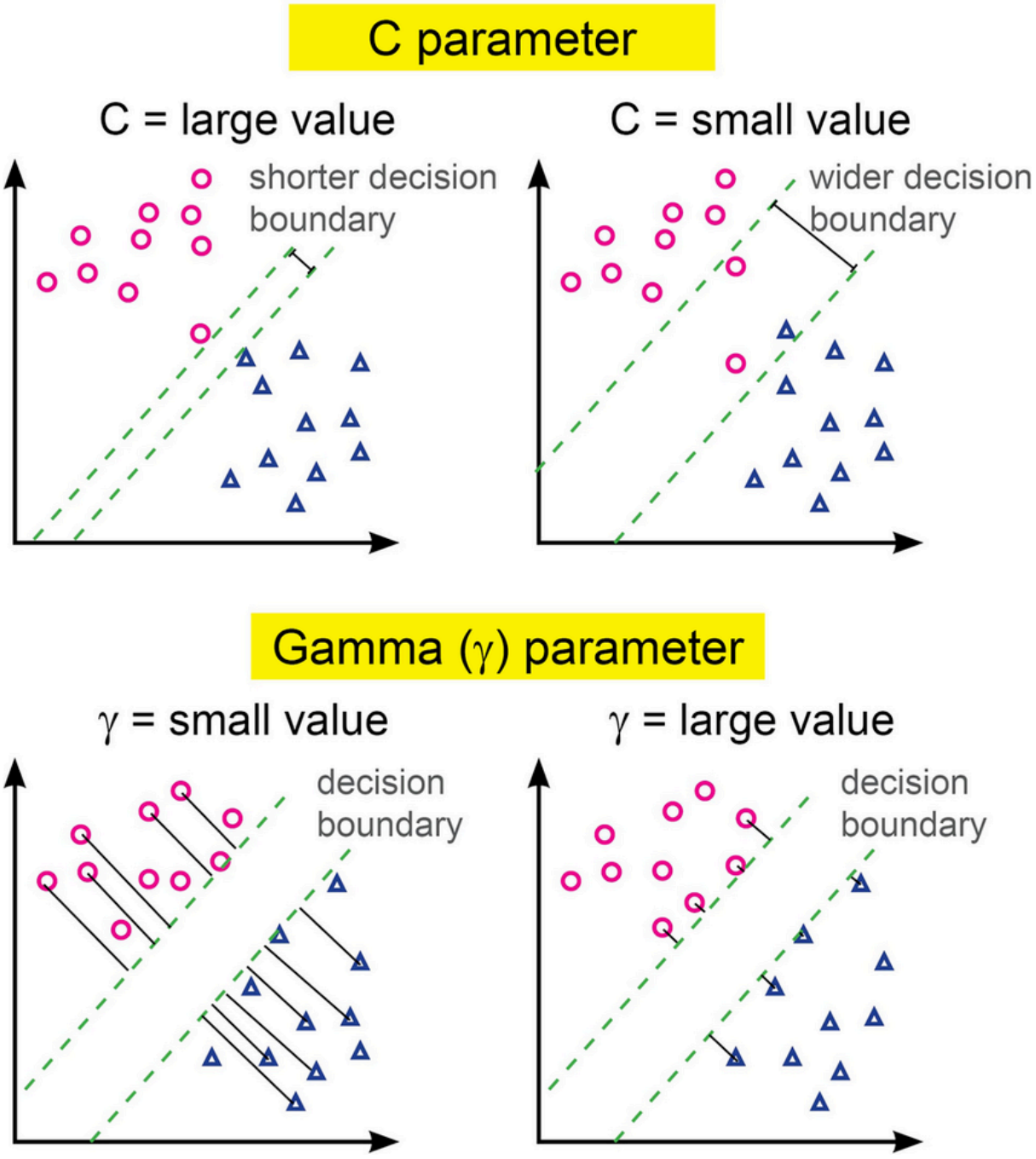
Hyperparameters

C (Regularization):

- **C nhỏ**: Wide margin, allow errors ➜ Prevent overfitting
- **C lớn**: Narrow margin, fewer errors ➜ Risk overfitting

Gamma (RBF kernel):

- **γ nhỏ**: Smooth boundary ➜ Risk underfitting
- **γ lớn**: Complex boundary ➜ Risk overfitting



Code Implementation - SVM

```
from sklearn.svm import SVC
from sklearn.datasets import load_breast_cancer

# Load data
data = load_breast_cancer()
X, y = data.data, data.target
```

```
# Scale (VERY IMPORTANT cho SVM!)
scaler = StandardScaler()
X_scaled = scaler.fit_transform(X)

# Split
X_train, X_test, y_train, y_test = train_test_split(
    X_scaled, y, test_size=0.2, random_state=42
)

# Linear SVM
svm_linear = SVC(kernel='linear', C=1.0)
svm_linear.fit(X_train, y_train)
print(f"Linear SVM Accuracy: {svm_linear.score(X_test, y_test):.4f}")

# RBF SVM
svm_rbf = SVC(kernel='rbf', C=1.0, gamma='scale')
svm_rbf.fit(X_train, y_train)
print(f"RBF SVM Accuracy: {svm_rbf.score(X_test, y_test):.4f}")

# Polynomial SVM
svm_poly = SVC(kernel='poly', degree=3, C=1.0)
svm_poly.fit(X_train, y_train)
print(f"Polynomial SVM Accuracy: {svm_poly.score(X_test, y_test):.4f}")
```

Grid Search for Optimal Hyperparameters

```
from sklearn.model_selection import GridSearchCV

# Define parameter grid
param_grid = {
    'C': [0.1, 1, 10, 100],
    'gamma': ['scale', 'auto', 0.001, 0.01, 0.1],
    'kernel': ['rbf']
}

# Grid Search
grid = GridSearchCV(SVC(), param_grid, cv=5, scoring='accuracy', verbos
```

```
e=1)
grid.fit(X_train, y_train)

print(f"Best Parameters: {grid.best_params_}")
print(f"Best CV Score: {grid.best_score_:.4f}")
print(f"Test Accuracy: {grid.score(X_test, y_test):.4f}")
```

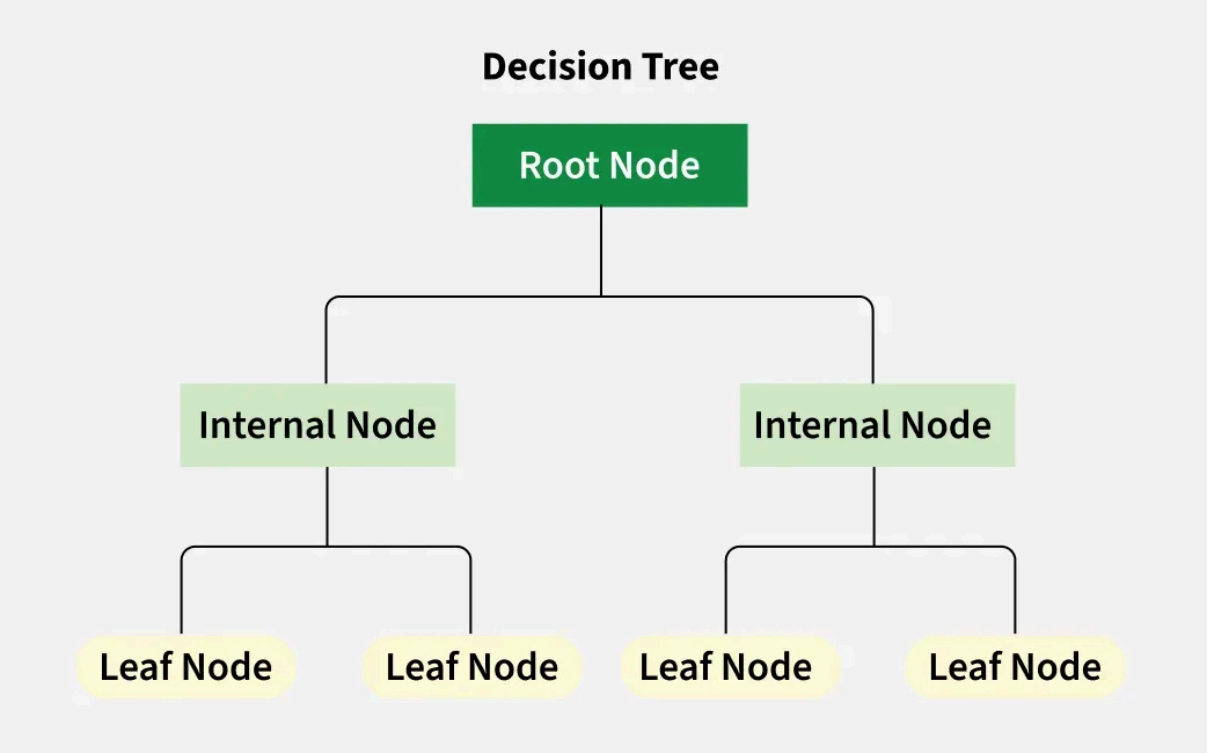
Module 4: Decision Trees

Decision Trees là thuật toán powerful và interpretable, chia data thành các subsets dựa trên **feature values**.

Nguyên lý hoạt động

Decision Tree giống flowchart:

1. **Root node**: Toàn bộ dataset
2. **Internal nodes**: Quyết định split
3. **Branches**: Outcomes
4. **Leaf nodes**: Final predictions



Greedy Algorithm

Decision Trees dùng **greedy algorithm** - chọn best split để **maximize purity** (giảm impurity nhất).

Impurity Measures

1. Gini Impurity Most Popular

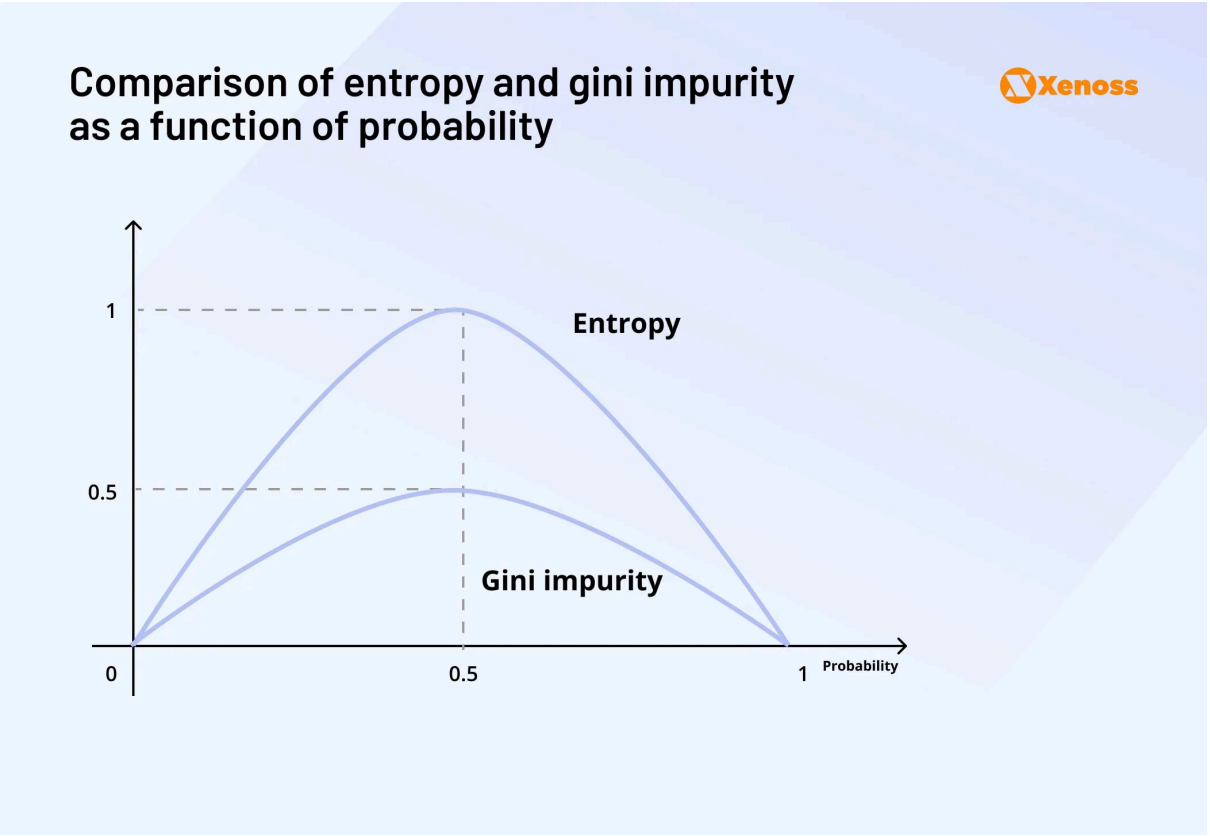
$$\text{Gini}(S) = 1 - \sum_{i=1}^{c} p_i^2$$

2. Entropy (Information Gain)

$$\text{Entropy}(S) = -\sum_{i=1}^{c} p_i \log_2(p_i)$$

Đặc điểm:

- **Gini/Entropy = 0**: Pure node
- **Gini ≈ 0.5 / Entropy = 1**: Maximum impurity



Overfitting và Solutions

Overfitting là vấn đề nghiêm trọng! Tree quá deep sẽ "học thuộc" training data.

Solutions:

1. **Pre-pruning**: Set max_depth, min_samples_split
2. **Cross Validation**: Tune hyperparameters
3. **Ensemble Methods**: Random Forest

Ưu và Nhược điểm

Ưu điểm:

- Very interpretable
- No feature scaling needed
- Handle nonlinear relationships
- Fast predictions

Nhược điểm:

- High variance
- Easy to overfit
- Sensitive to data changes

Code Implementation - Decision Trees

```
from sklearn.tree import DecisionTreeClassifier, plot_tree
import matplotlib.pyplot as plt

# Load data
iris = load_iris()
X, y = iris.data, iris.target

X_train, X_test, y_train, y_test = train_test_split(
    X, y, test_size=0.2, random_state=42
)

# Train Decision Tree
dt = DecisionTreeClassifier(
    max_depth=3,
    min_samples_split=5,
    min_samples_leaf=2,
    random_state=42
)
dt.fit(X_train, y_train)

print(f"Training Accuracy: {dt.score(X_train, y_train):.4f}")
print(f"Test Accuracy: {dt.score(X_test, y_test):.4f}")

# Visualize Tree
plt.figure(figsize=(20, 10))
plot_tree(dt,
          feature_names=iris.feature_names,
          class_names=iris.target_names,
          filled=True,
```

```
          rounded=True,
          fontsize=12)
plt.title("Decision Tree Visualization")
plt.show()
```

Feature Importance

```
import pandas as pd

# Get feature importance
importance = pd.DataFrame({
    'Feature': iris.feature_names,
    'Importance': dt.feature_importances_
}).sort_values('Importance', ascending=False)

print(importance)

# Visualize
plt.figure(figsize=(10, 6))
plt.barh(importance['Feature'], importance['Importance'])
plt.xlabel('Importance')
plt.title('Feature Importance - Decision Tree')
plt.gca().invert_yaxis()
plt.show()
```

Module 5: Ensemble Models

Ensemble Learning: *"Sức mạnh của tập thể!"* Combine nhiều weak learners để tạo strong learner với performance tốt hơn.

Ba phương pháp Ensemble chính

1. Bagging (Bootstrap Aggregating)

- Train nhiều models **parallel** trên different subsets
- **Reduce variance**

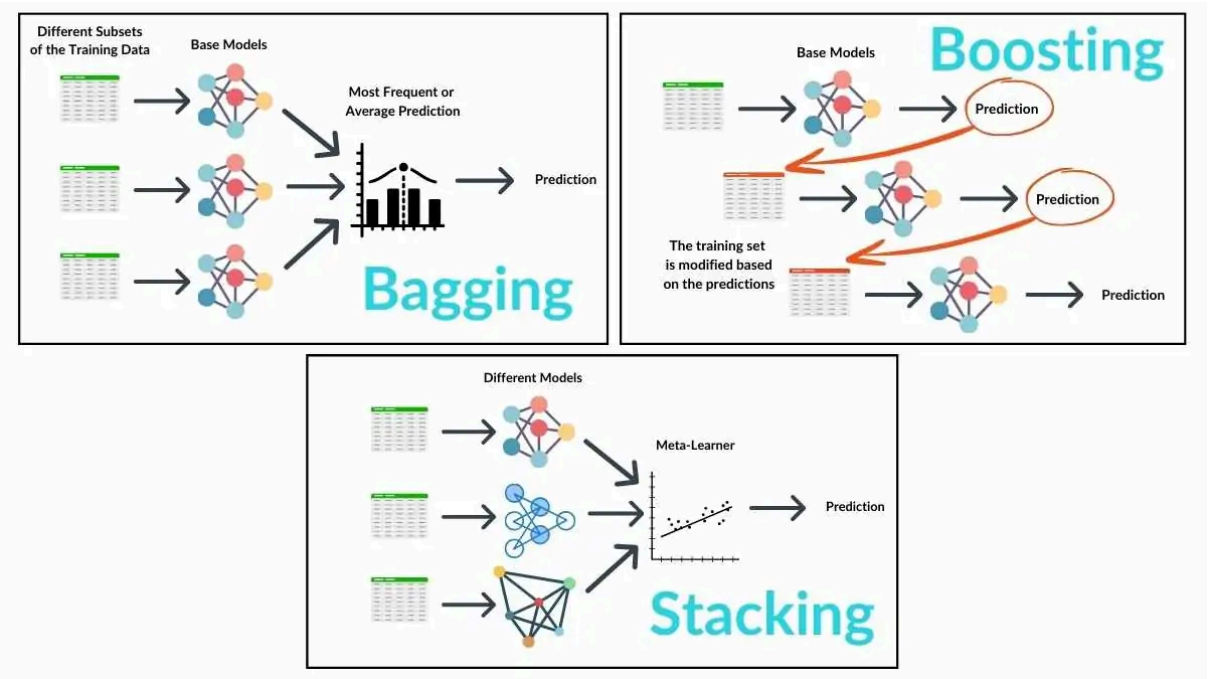
- Ví dụ: Random Forest

2. Boosting

- Train nhiều models **sequential**
- Mỗi model học từ errors của model trước
- **Reduce bias**
- Ví dụ: AdaBoost, Gradient Boosting

3. Stacking

- Combine predictions của nhiều models bằng meta-model
- **Most flexible**



Random Forest

Random Forest = Bagging + **Random Feature Selection** tại mỗi split → Decorrelated trees → Reduce variance hơn!

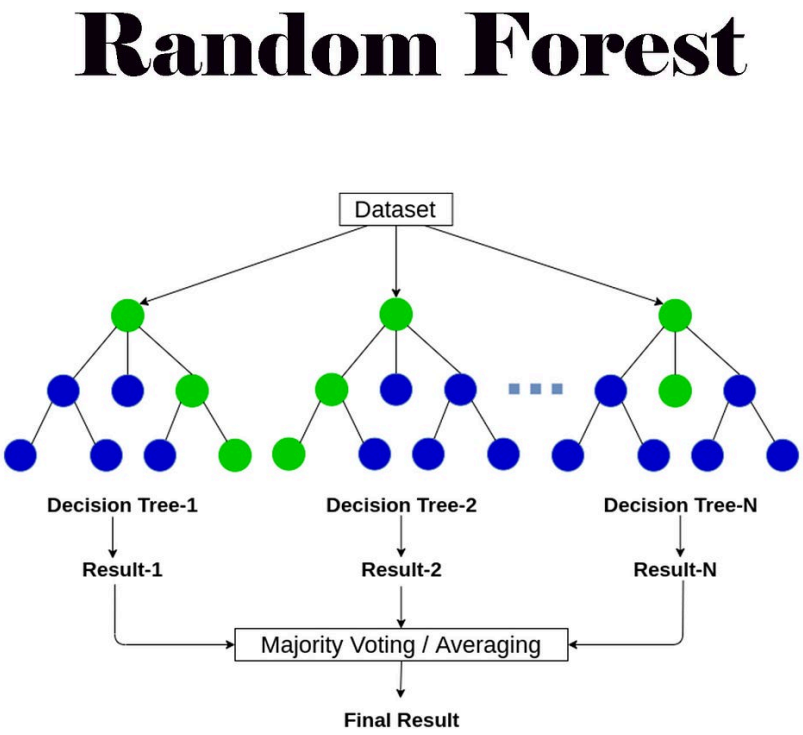
Tại sao Random Forest tốt?

Problem với Bagging:

- Nếu có strong predictors → Trees become **correlated**
- Averaging correlated trees → Ít giảm variance

Solution - Random Forest:

- Mỗi split chỉ consider random subset of m features
- Typically $m = \sqrt{p}$ cho classification
- Create **decorrelated trees** → Greater variance reduction!



Out-of-Bag (OOB) Error

OOB Error đánh giá model **không cần validation set** riêng! Mỗi bootstrap sample dùng ~63% data, 37% còn lại để evaluate.

Code Implementation - Random Forest

```
from sklearn.ensemble import RandomForestClassifier

# Load data
data = load_breast_cancer()
X, y = data.data, data.target

X_train, X_test, y_train, y_test = train_test_split(
    X, y, test_size=0.2, random_state=42, stratify=y
)

# Random Forest
rf = RandomForestClassifier(
    n_estimators=100,
    max_depth=10,
    max_features='sqrt',
    oob_score=True,
    random_state=42,
    n_jobs=-1
)
rf.fit(X_train, y_train)

print(f"OOB Score: {rf.oob_score_:.4f}")
print(f"Training Accuracy: {rf.score(X_train, y_train):.4f}")
print(f"Test Accuracy: {rf.score(X_test, y_test):.4f}")
```

Feature Importance

```
# Random Forest Feature Importance
importance_rf = pd.DataFrame({
    'Feature': data.feature_names,
    'Importance': rf.feature_importances_
}).sort_values('Importance', ascending=False).head(10)

# Visualize
plt.figure(figsize=(10, 6))
plt.barh(importance_rf['Feature'], importance_rf['Importance'], color='fore
stgreen')
```

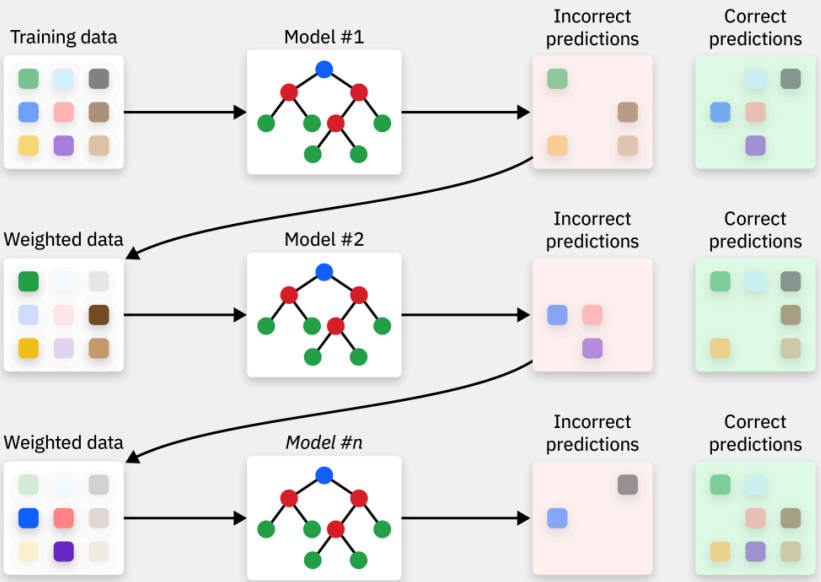
```
plt.xlabel('Importance')
plt.title('Top 10 Features - Random Forest')
plt.gca().invert_yaxis()
plt.grid(True, alpha=0.3)
plt.show()
```

Gradient Boosting

Gradient Boosting fit new models vào **residuals (errors)** của models trước, giống gradient descent optimization!

Algorithm (Simplified)

1. Initialize: $F_0(x) = \bar{y}$
2. For $m = 1$ to M:
   - Calculate residuals: $r_{im} = y_i - F_{m-1}(x_i)$
   - Fit tree h_m to residuals
   - Update: $F_m(x) = F_{m-1}(x) + \nu \cdot h_m(x)$



Loss Functions

3 Loss Functions phổ biến:

1. Exponential Loss (AdaBoost)

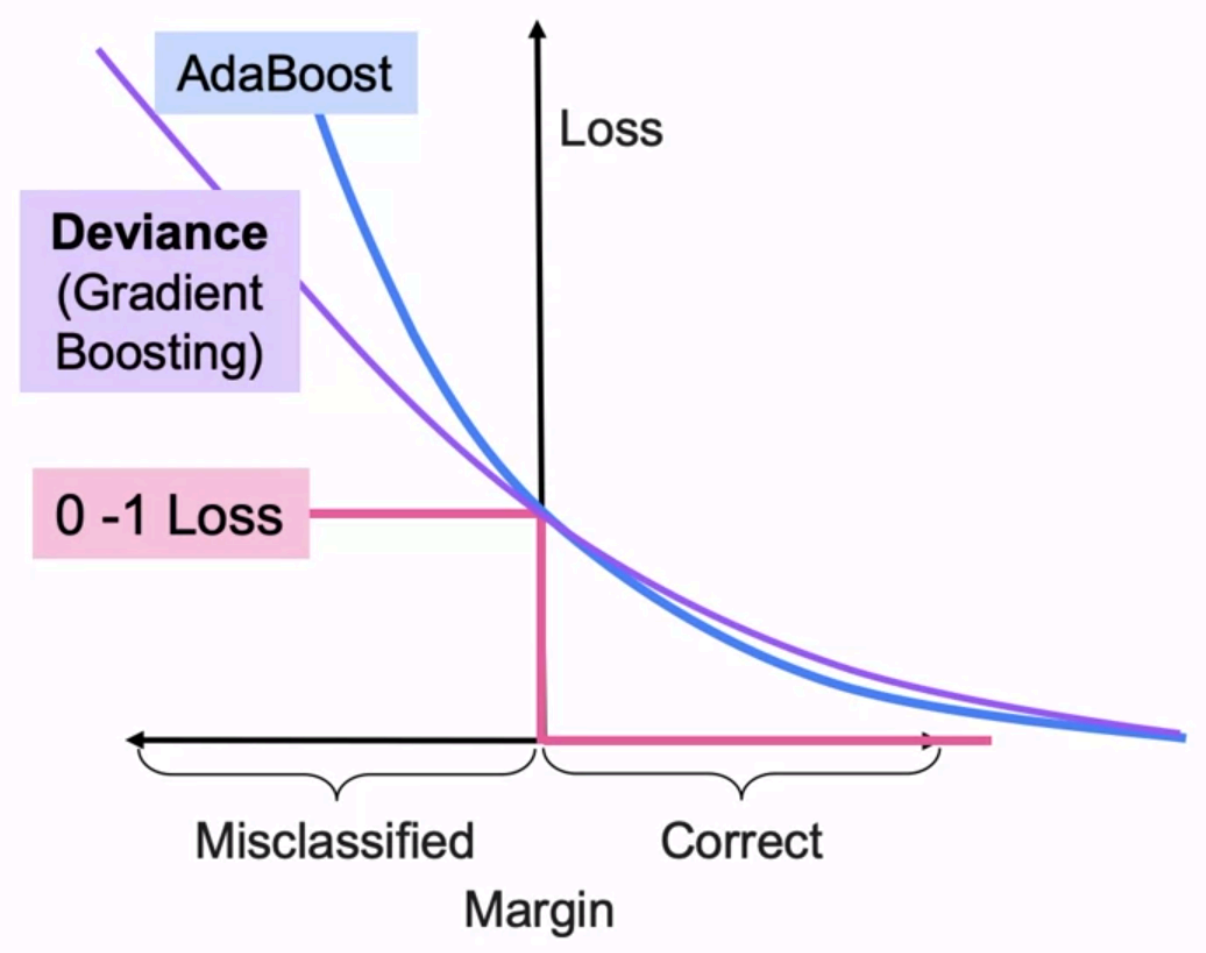
$$L(y, f(x)) = \exp(-y \cdot f(x))$$

- Sensitive to outliers

2. Deviance/Log Loss (Gradient Boosting)

$$L(y, f(x)) = -\log(1 + \exp(-2yf(x)))$$

- Robust to outliers
- Most common



Key Hyperparameters

Parameter	Range	Effect
n_estimators	50-500	More trees → Better fit, slower
learning_rate	0.01-0.1	Smaller → Need more trees, better generalization
max_depth	3-6	Shallow trees typical for boosting
subsample	0.5-1.0	<1.0 = Stochastic GB, reduce overfitting

Code Implementation - Boosting

```
from sklearn.ensemble import GradientBoostingClassifier, AdaBoostClassifi
er

# Gradient Boosting
gb = GradientBoostingClassifier(
    n_estimators=100,
    learning_rate=0.1,
    max_depth=3,
    subsample=0.8,
    random_state=42
)
gb.fit(X_train, y_train)
print(f"Gradient Boosting Test Accuracy: {gb.score(X_test, y_test):.4f}")

# AdaBoost
ada = AdaBoostClassifier(
    n_estimators=100,
    learning_rate=1.0,
    random_state=42
)
ada.fit(X_train, y_train)
print(f"AdaBoost Test Accuracy: {ada.score(X_test, y_test):.4f}")
```

Compare All Ensemble Methods

```
from sklearn.ensemble import BaggingClassifier

models = {
```

```
    'Bagging': BaggingClassifier(n_estimators=100, random_state=42),
    'Random Forest': RandomForestClassifier(n_estimators=100, random_stat
e=42),
    'AdaBoost': AdaBoostClassifier(n_estimators=100, random_state=42),
    'Gradient Boosting': GradientBoostingClassifier(n_estimators=100, rando
m_state=42)
}

results = {}
for name, model in models.items():
    model.fit(X_train, y_train)
    train_acc = model.score(X_train, y_train)
    test_acc = model.score(X_test, y_test)
    results[name] = {'Train': train_acc, 'Test': test_acc}
    print(f"{name}:")
    print(f"  Train: {train_acc:.4f}")
    print(f"  Test: {test_acc:.4f}\n")

# Visualize
import pandas as pd
results_df = pd.DataFrame(results).T
results_df.plot(kind='bar', figsize=(10, 6))
plt.title('Ensemble Methods Comparison')
plt.ylabel('Accuracy')
plt.xticks(rotation=45)
plt.legend(['Training', 'Test'])
plt.grid(True, alpha=0.3)
plt.tight_layout()
plt.show()
```

Stacking

Stacking combines predictions của nhiều **different models** (heterogeneous) bằng **meta-model** để tạo final prediction.

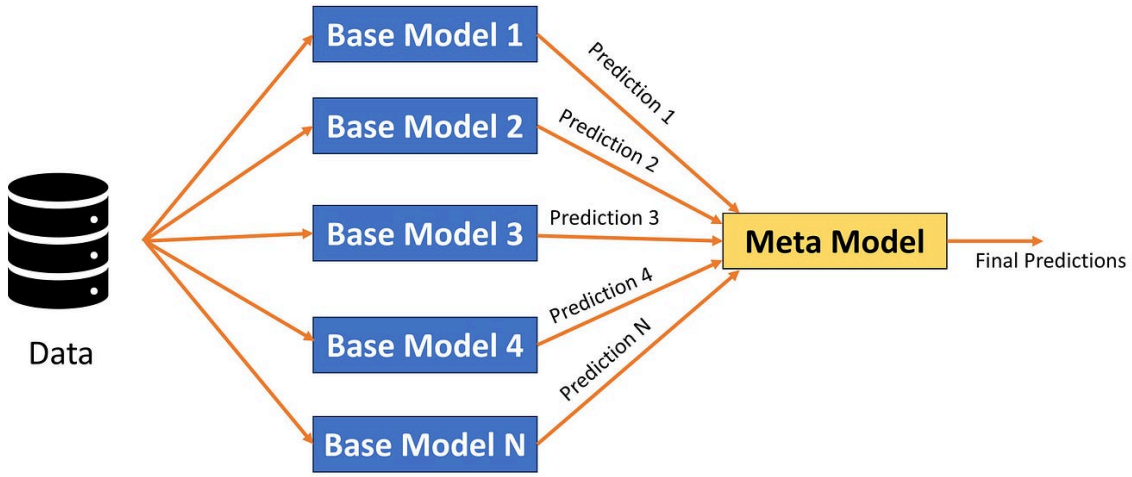
Architecture

Level 0 (Base Models):

- Multiple diverse models (Logistic Regression, SVM, Random Forest, etc.)
- Train on original training data

Level 1 (Meta-Model):

- Train on predictions của base models
- Learn how to best combine predictions
- Common: Logistic Regression, Ridge



Code Implementation

```
from sklearn.ensemble import StackingClassifier

# Base models
base_models = [
    ('lr', LogisticRegression(max_iter=10000)),
    ('rf', RandomForestClassifier(n_estimators=100)),
    ('svm', SVC(probability=True))
]

# Stacking
stacking = StackingClassifier(
```

```
    estimators=base_models,
    final_estimator=LogisticRegression(),
    cv=5
)

stacking.fit(X_train, y_train)
print(f"Stacking Test Accuracy: {stacking.score(X_test, y_test):.4f}")
```

Module 6: Modeling Imbalanced Classes

Imbalanced Data: Một class có **nhiều samples hơn đáng kể**. Vấn đề phổ biến: fraud detection (99% vs 1%), disease diagnosis (95% vs 5%).

Vấn đề với Imbalanced Data

3 Problems chính:

1. Model bias towards majority class

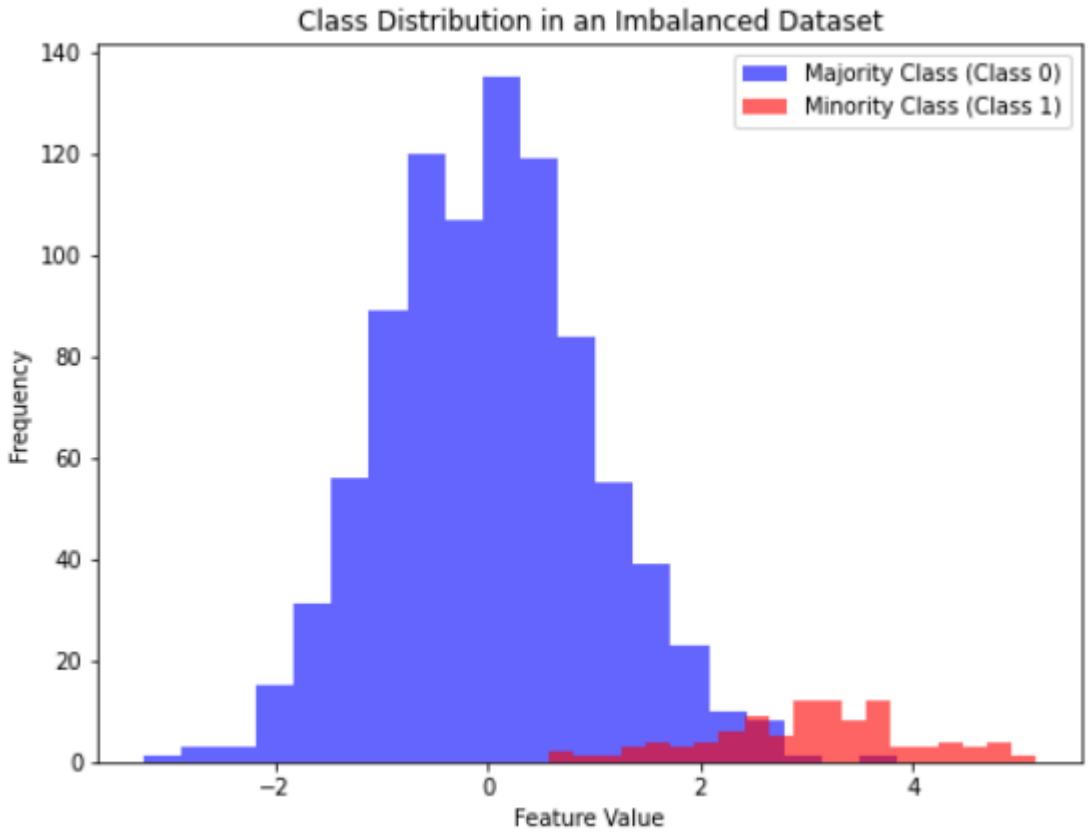
- Predict majority class cho tất cả → High accuracy nhưng useless!

2. Metrics misleading

- Accuracy không phù hợp
- Cần: Precision, Recall, F1, AUC

3. Poor minority class detection

- False Negatives cao → Nguy hiểm!



Approaches xử lý Imbalanced Data

1. Algorithm Level: Class Weights

Assign **higher weights** cho minority class → Model pay more attention!

```
# Auto-balance weights
model = LogisticRegression(class_weight='balanced')

# Custom weights
model = LogisticRegression(class_weight={0: 1, 1: 10})
```

Ưu điểm: Fast, không thay đổi data

2. Data Level: Resampling

A. Downsampling (Undersampling)

Remove samples từ **majority class** để balance.

- **Ưu điểm:** Fast training
- **Nhược điểm:** Loss of information

B. Upsampling (Oversampling)

Duplicate samples từ **minority class** để balance.

Methods:

- **Random Oversampling**: Duplicate ngẫu nhiên
- **SMOTE** : Create synthetic samples

SMOTE (Synthetic Minority Oversampling)

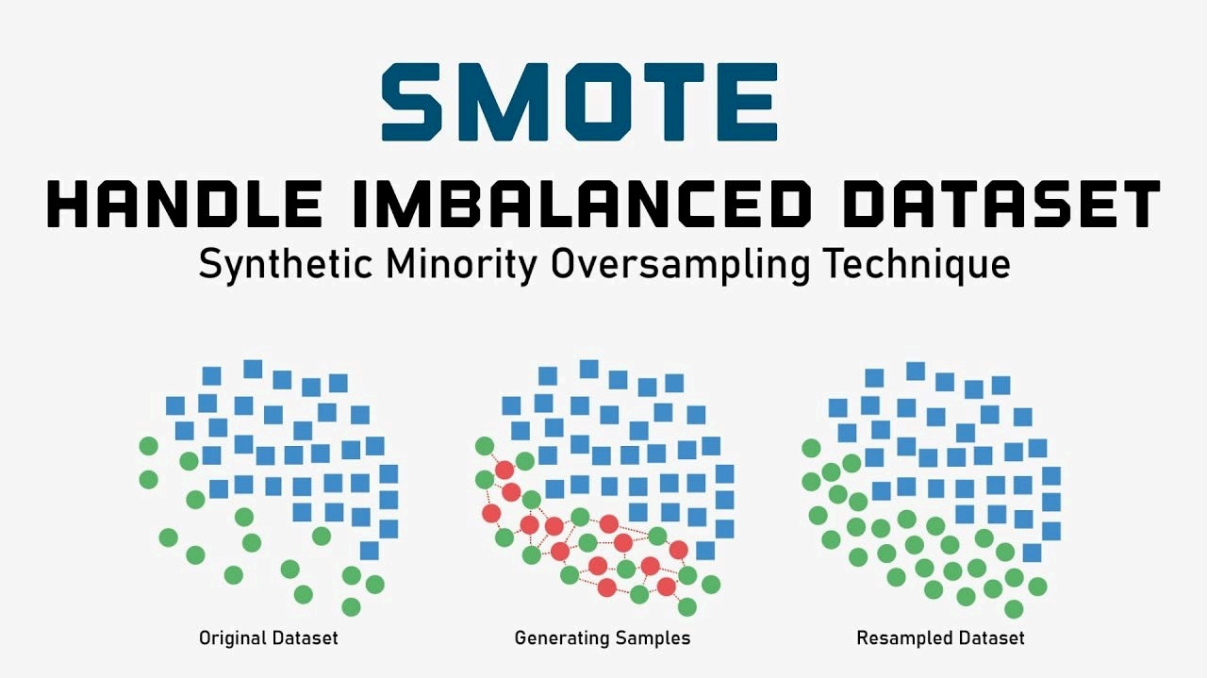
SMOTE creates **synthetic samples** thay vì duplicate! Chọn minority sample → Find K neighbors → Create new sample between them.

Algorithm:

1. Choose minority sample x_i
2. Find K nearest neighbors (K=5)
3. Randomly select neighbor x_{nn}
4. Create synthetic:

$$x_{new} = x_i + \lambda \cdot (x_{nn} - x_i)$$

where $\lambda \in [0, 1]$ random



Ưu điểm:

- No information loss
- Creates diverse samples
- Reduce overfitting

---

3. Combination Methods

SMOTE + Tomek Links

- SMOTE to oversample
- Tomek Links to clean overlapping samples

SMOTE + ENN

- SMOTE to oversample
- ENN to remove noisy samples

Code Implementation - Imbalanced Data

Setup Imbalanced Dataset

```
from sklearn.datasets import make_classification
from collections import Counter
```

```
from imblearn.over_sampling import SMOTE
from imblearn.under_sampling import RandomUnderSampler
from imblearn.combine import SMOTETomek

# Create imbalanced dataset
X_imb, y_imb = make_classification(
    n_samples=1000, n_features=20,
    weights=[0.95, 0.05],  # 95% vs 5%
    random_state=42
)

print(f"Class distribution: {Counter(y_imb)}")
# Output: {0: 950, 1: 50}

X_train, X_test, y_train, y_test = train_test_split(
    X_imb, y_imb, test_size=0.2, stratify=y_imb, random_state=42
)
```

Compare Methods

```
from sklearn.metrics import f1_score, classification_report

# 1. Baseline (No handling)
baseline = LogisticRegression(max_iter=10000)
baseline.fit(X_train, y_train)
y_pred_baseline = baseline.predict(X_test)
print("Baseline (No handling):")
print(classification_report(y_test, y_pred_baseline))

# 2. Class Weights
weighted = LogisticRegression(class_weight='balanced', max_iter=10000)
weighted.fit(X_train, y_train)
y_pred_weighted = weighted.predict(X_test)
print("\nClass Weights:")
print(classification_report(y_test, y_pred_weighted))

# 3. Random Undersampling
rus = RandomUnderSampler(random_state=42)
```

```
X_train_rus, y_train_rus = rus.fit_resample(X_train, y_train)
under_model = LogisticRegression(max_iter=10000)
under_model.fit(X_train_rus, y_train_rus)
y_pred_under = under_model.predict(X_test)
print("\nRandom Undersampling:")
print(classification_report(y_test, y_pred_under))

# 4. SMOTE
smote = SMOTE(random_state=42)
X_train_smote, y_train_smote = smote.fit_resample(X_train, y_train)
smote_model = LogisticRegression(max_iter=10000)
smote_model.fit(X_train_smote, y_train_smote)
y_pred_smote = smote_model.predict(X_test)
print("\nSMOTE:")
print(classification_report(y_test, y_pred_smote))

# 5. SMOTE + Tomek Links
smote_tomek = SMOTETomek(random_state=42)
X_train_st, y_train_st = smote_tomek.fit_resample(X_train, y_train)
st_model = LogisticRegression(max_iter=10000)
st_model.fit(X_train_st, y_train_st)
y_pred_st = st_model.predict(X_test)
print("\nSMOTE + Tomek Links:")
print(classification_report(y_test, y_pred_st))
```

Best Practices cho Imbalanced Data

7 Tips quan trọng khi làm việc với imbalanced data:

1. Always Check Class Distribution

```
print(Counter(y_train))

# Visualize
import seaborn as sns
sns.countplot(x=y_train)
```

```
plt.title('Class Distribution')
plt.show()
```

2. Use Appropriate Metrics

DON'T: Rely on Accuracy alone

DO USE:

- **Precision**: When False Positives costly
- **Recall**: When False Negatives dangerous
- **F1-Score**: Balanced metric
- **ROC-AUC**: Overall performance
- **PR-AUC**: Better for imbalanced

3. Stratified Splitting

```
# Always stratify!
X_train, X_test, y_train, y_test = train_test_split(
    X, y, stratify=y, test_size=0.2
)
```

4. Cross-Validation với Stratification

```
from sklearn.model_selection import StratifiedKFold

skf = StratifiedKFold(n_splits=5, shuffle=True)
for train_idx, val_idx in skf.split(X, y):
    # Train and evaluate
    pass
```

5. Try Multiple Approaches

Recommended workflow:

1. Start with **Class Weights** (fastest)
2. Try **SMOTE** (usually works well)
3. Try **combination methods**

4. Use **Ensemble methods**

6. Adjust Classification Threshold

```
# Get probabilities
y_proba = model.predict_proba(X_test)[:, 1]

# Find optimal threshold
from sklearn.metrics import precision_recall_curve
precision, recall, thresholds = precision_recall_curve(y_test, y_proba)

# Maximize F1
f1_scores = 2 * (precision * recall) / (precision + recall)
optimal_idx = np.argmax(f1_scores)
optimal_threshold = thresholds[optimal_idx]

print(f"Optimal Threshold: {optimal_threshold:.3f}")

# Predict với custom threshold
y_pred_custom = (y_proba >= optimal_threshold).astype(int)
```

7. Use Ensemble Methods

```
from imblearn.ensemble import BalancedBaggingClassifier, BalancedRando
mForestClassifier

# Balanced methods handle imbalance internally
balanced_rf = BalancedRandomForestClassifier(n_estimators=100, random
_state=42)
balanced_rf.fit(X_train, y_train)

balanced_bag = BalancedBaggingClassifier(n_estimators=100, random_stat
e=42)
balanced_bag.fit(X_train, y_train)
```