# Model Evaluation and Selection

## Cross-Validation: k-fold, Leave-One-Out, and Stratified Sampling

Sarwan Ali

Department of Computer Science
Georgia State University

Understanding Model Evaluation

# Today's Learning Journey

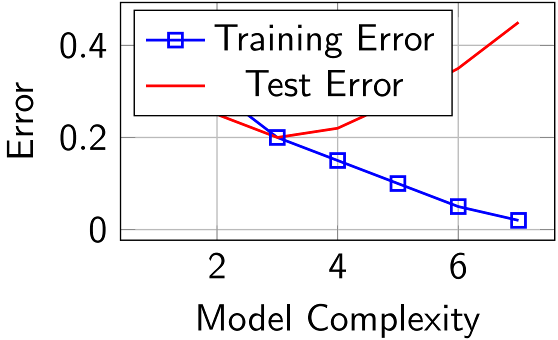
# Why Model Evaluation Matters

**The Challenge:**

- How do we know if our model is good?
- Training accuracy can be misleading
- Need to estimate generalization performance

**The Solution:**

- Cross-validation techniques
- Robust performance estimation
- Model selection and comparison




Overfitting Problem

# What is Cross-Validation?

**Definition**

Cross-validation is a statistical method for estimating the performance of machine learning models by partitioning data into subsets, training on some subsets, and validating on others.

**Key Benefits:**

- Robust estimation of model performance
- Model selection - choose best hyperparameters
- Model comparison - compare different algorithms
- Variance reduction in performance estimates

⚠ **Remember:** Never use test data for cross-validation!

# k-Fold Cross-Validation: The Concept

**How it works:**

1. Divide dataset into $k$ equal-sized folds
2. For each fold $i = 1, 2, ..., k$:
   - Use fold $i$ as validation set
   - Use remaining $k - 1$ folds as training set
   - Train model and evaluate on validation fold
3. Average the $k$ performance scores

**Mathematical Formula:**

$$CV_{(k)} = \frac{1}{k} \sum_{i=1}^{k} L(f_i, D_i)$$

where $L$ is the loss function, $f_i$ is the model trained on fold $i$, and $D_i$ is the validation data for fold $i$.

| | | | | | |
|---|---|---|---|---|---|
| Fold 1 | V | T | T | T | T |
| Fold 2 | T | V | T | T | T |
| Fold 3 | T | T | V | T | T |
| Fold 4 | T | T | T | V | T |
| Fold 5 | T | T | T | T | V |

T = Training, V = Validation

5-Fold Cross-Validation

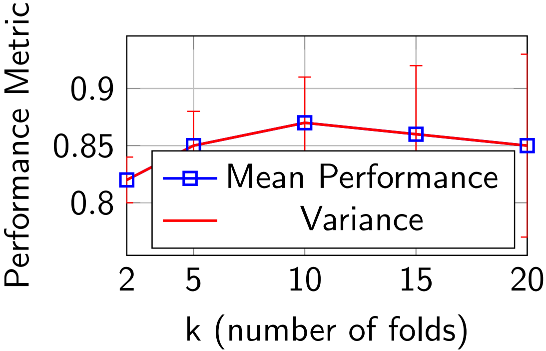
# k-Fold: Choosing the Right k

**Common choices:**

- k = 5: Good balance, computationally efficient
- k = 10: Most popular choice, good bias-variance trade-off
- k = n: Leave-one-out (special case)

**Trade-offs:**

- Higher k: Lower bias, higher variance
- Lower k: Higher bias, lower variance
- Computational cost: Increases with k




Bias-Variance Trade-off

# k-Fold Implementation Example

```
from sklearn.model_selection import KFold
from sklearn.linear_model import LogisticRegression
from sklearn.metrics import accuracy_score
import numpy as np

# Initialize k-fold cross-validator
kf = KFold(n_splits=5, shuffle=True, random_state=42)

# Initialize model
model = LogisticRegression()

# Store scores
scores = []

# Perform k-fold cross-validation
for train_idx, val_idx in kf.split(X):
    X_train, X_val = X[train_idx], X[val_idx]
    y_train, y_val = y[train_idx], y[val_idx]

    # Train model
    model.fit(X_train, y_train)
    # Predict and evaluate
    y_pred = model.predict(X_val)
    score = accuracy_score(y_val, y_pred)
    scores.append(score)
# Calculate final CV score
cv_score = np.mean(scores)
cv_std = np.std(scores)
```

## Key Points:

- shuffle=True: Randomize data order
- random_state: Reproducible results
- Store all scores: Calculate mean and standard deviation

## Output Example:

| Fold | Accuracy |
|---|---|
| 1 | 0.87 |
| 2 | 0.85 |
| 3 | 0.89 |
| 4 | 0.86 |
| 5 | 0.88 |
| **Mean** | **0.87 +- 0.014** |

# Leave-One-Out Cross-Validation (LOOCV)

## Definition

LOOCV is a special case of k-fold cross-validation where $k = n$ (number of samples). Each sample is used once as validation data while the remaining $n-1$ samples form the training set.

**Characteristics:**

- Maximum data usage: $n-1$ samples for training
- Deterministic: No randomness in splits
- Unbiased estimate: Nearly unbiased performance estimate
- High variance: Can be unstable
- Computationally expensive: $n$ model trainings

| | | | | | | |
|---|---|---|---|---|---|---|
| Run 1 | V | T | T | T | T | T |
| Run 2 | T | V | T | T | T | T |
| Run 3 | T | T | V | T | T | T |
| Run 4 | T | T | T | V | T | T |
| Run 5 | T | T | T | T | V | T |
| Run 6 | T | T | T | T | T | V |

Each sample used once for validation

LOOCV with n=6 samples

**Mathematical Formula:** $CV_{LOO} = \frac{1}{n}\sum_{i=1}^{n} L(f_{-i}, x_i, y_i)$, where $f_{-i}$ is the model trained on all data except sample $i$.

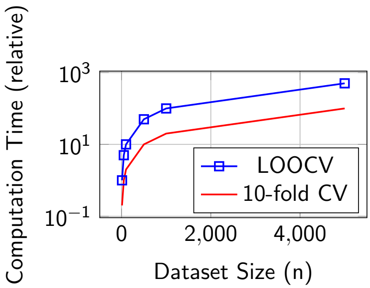
# LOOCV: When to Use It

**Good for:**

- Small datasets (n ¡ 100)
- When maximum training data is needed
- Deterministic evaluation required
- Linear models (computationally efficient)

**Avoid when:**

- Large datasets (computational cost)
- Complex models (deep learning)
- High variance is problematic
- k-fold gives similar results




Computational Complexity

**Rule of Thumb:** Use LOOCV for n ¡ 100, otherwise use k-fold CV

# LOOCV Implementation

```python
from sklearn.model_selection import LeaveOneOut
from sklearn.linear_model import LinearRegression
from sklearn.metrics import mean_squared_error
import numpy as np

# Initialize LOOCV
loo = LeaveOneOut()

# Initialize model
model = LinearRegression()

# Store scores
scores = []
# Perform LOOCV
for train_idx, test_idx in loo.split(X):
    X_train, X_test = X[train_idx], X[test_idx]
    y_train, y_test = y[train_idx], y[test_idx]
    # Train model on n-1 samples
    model.fit(X_train, y_train)
    # Predict on single test sample
    y_pred = model.predict(X_test)
    # Calculate error for this sample
    error = mean_squared_error(y_test, y_pred)
    scores.append(error)
# Calculate LOOCV score
loocv_score = np.mean(scores)
print(f"LOOCV MSE: {loocv_score:.4f}")
```

**Alternative: Efficient LOOCV**

- For linear regression: analytical formula exists
- No need to retrain n times
- Leverage scores can speed up computation

**Analytical LOOCV for Linear Regression:**

$$CV_{LOO} = \frac{1}{n}\sum_{i=1}^{n}\left(\frac{y_i - \hat{y}_i}{1 - h_{ii}}\right)^2$$

where $h_{ii}$ is the $i$-th diagonal element of the hat matrix $H = X(X^TX)^{-1}X^T$.

💡 This reduces complexity from $O(n \cdot p^3)$ to $O(p^3)$!

# Stratified Cross-Validation: Motivation

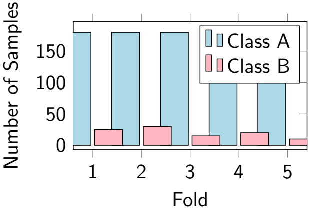
**The Problem:**

- Imbalanced datasets
- Random splits may not preserve class distribution
- Some folds might have very few (or no) samples from minority classes

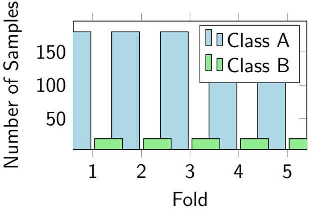
**Example Dataset:**

- Class A: 900 samples (90%)
- Class B: 100 samples (10%)

Risk: Random 5-fold CV might create a fold with only Class A samples!




Random CV: Uneven distribution




Stratified CV: Balanced distribution

# Stratified Cross-Validation: How It Works

**Definition**

Stratified cross-validation ensures that each fold maintains approximately the same percentage of samples from each target class as the complete dataset.

**Algorithm:**

1. Calculate class proportions in the full dataset
2. For each class, divide samples into k groups
3. Combine corresponding groups from each class to form folds
4. Each fold maintains original class proportions

Original Dataset

Class A: 80%, Class B: 20%

↓

| Fold 1 | Fold 2 | Fold 3 | Fold 4 | Fold 5 |
|---|---|---|---|---|
| 80%/20% | 80%/20% | 80%/20% | 80%/20% | 80%/20% |

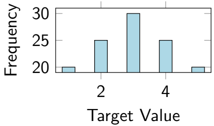
Each fold preserves original class distribution

# Types of Stratified Sampling

1. **Classification Tasks:**
   - Stratify by target class labels
   - Maintain class proportions
   - Essential for imbalanced datasets
2. **Regression Tasks:**
   - Stratify by target value bins
   - Create quantile-based bins
   - Ensure even distribution of target values
3. **Multi-label Classification:**
   - More complex stratification
   - Consider label combinations
   - Use iterative stratification

**Binary Classification Example:**

| Fold | Class 0 | Class 1 | Ratio |
|---|---|---|---|
| 1 | 72 | 18 | 80:20 |
| 2 | 72 | 18 | 80:20 |
| 3 | 72 | 18 | 80:20 |
| 4 | 72 | 18 | 80:20 |
| 5 | 72 | 18 | 80:20 |
| **Total** | **360** | **90** | **80:20** |




**Regression Binning:**

Equal-sized bins for stratification

# Stratified CV Implementation

```python
from sklearn.model_selection import StratifiedKFold
from sklearn.ensemble import RandomForestClassifier
from sklearn.metrics import classification_report
import numpy as np
# Initialize Stratified k-fold
skf = StratifiedKFold(n_splits=5, shuffle=True, random_state=42)
# Initialize model
model = RandomForestClassifier(random_state=42)
fold_scores = []
all_predictions = []
all_true_labels = []
# Perform stratified cross-validation
for fold, (train_idx, val_idx) in enumerate(skf.split(X, y)):
    X_train, X_val = X[train_idx], X[val_idx]
    y_train, y_val = y[train_idx], y[val_idx]
    # Check class distribution in this fold
    print(f"Fold {fold+1} - Class distribution:")
    unique, counts = np.unique(y_val, return_counts=True)
    print(dict(zip(unique, counts)))
    # Train and evaluate
    model.fit(X_train, y_train)
    y_pred = model.predict(X_val)
    fold_scores.append(accuracy_score(y_val, y_pred))
    all_predictions.extend(y_pred)
    all_true_labels.extend(y_val)
print(f"CV Score Mean: {np.mean(fold_scores):.4f}")
print(f"CV Score +- std: {np.std(fold_scores):.4f}")
```

## For Regression:

```python
from sklearn.model_selection import KFold
import pandas as pd
# Create bins for stratification
def create_bins(y, n_bins=5):
    return pd.cut(y, bins=n_bins, labels=False)
# Create target bins
y_binned = create_bins(y, n_bins=5)
# Use stratified CV with bins
skf = StratifiedKFold(n_splits=5)
for train_idx, val_idx in skf.split(X, y_binned):
    pass # Your CV code here
```

## Benefits:

- Consistent evaluation: Each fold is representative
- Reduced variance: More stable CV scores
- Better for imbalanced data: Fair evaluation across classes

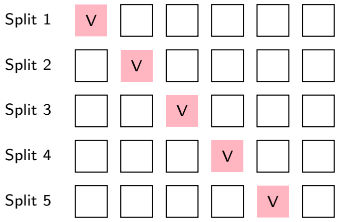
# Time Series Cross-Validation

**The Challenge:**

- Temporal dependencies in data
- Cannot use future to predict past
- Standard CV violates temporal order

**Time Series CV:**

- Forward chaining: Use past to predict future
- Expanding window: Training set grows over time
- Rolling window: Fixed-size training window




Forward Chaining CV

Respecting temporal order

**Key Principle:** Never use future information to predict the past!

# Group-Based Cross-Validation

**When to Use:**

- Multiple samples from same subject/group
- Spatial data with geographic clusters
- Multiple measurements per patient
- Image patches from same image

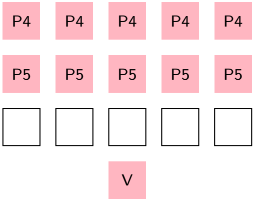
**The Problem:**

- Data leakage between train/validation
- Overoptimistic performance estimates
- Poor generalization to new groups

**Solution: GroupKFold**

- Ensure groups don't span train/validation
- Split by groups, not individual samples
- More realistic performance estimates

**Example: Medical Study**




P = Patient, T = Train, V = Validation

Group-based splits preserve independence

# Common Pitfalls and How to Avoid Them

**Pitfall 1: Data Leakage**

- Feature scaling on entire dataset
- Feature selection before CV
- Using test data in CV

**Solution:** Preprocessing inside CV loop

**Pitfall 2: Wrong CV for Time Series**

- Using random splits
- Ignoring temporal dependencies

**Solution:** Time series CV methods

**Pitfall 3: Ignoring Class Imbalance**

- Random splits with imbalanced data
- Inconsistent evaluation metrics

**Solution:** Stratified CV

**Pitfall 4: Hyperparameter Tuning**

- Using same data for CV and hyperparameter search
- Information leakage through repeated CV

**Solution:** Nested CV

**Pitfall 5: Statistical Significance**

- Comparing models on single CV run
- Ignoring variance in estimates

**Solution:** Multiple CV runs + statistical tests

| Incorrect | Correct |
|---|---|
| ↓ | ↓ |
| Scale $\rightarrow$ CV | CV $\rightarrow$ Scale |

# Nested Cross-Validation

## The Problem:

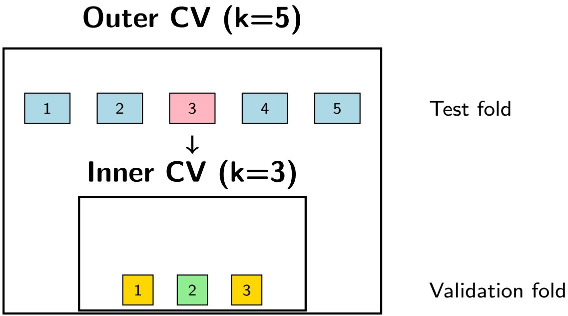
- Hyperparameter tuning needs validation data
- Using same CV for both tuning and evaluation
- Leads to optimistic bias

## Nested CV Solution:

- Outer loop: Model evaluation
- Inner loop: Hyperparameter tuning
- Each outer fold is completely independent

## Algorithm:

1. Split data into k outer folds
   - Use training data for inner CV
   - Find best hyperparameters
   - Train final model with best params
   - Evaluate on outer validation fold
2. Average outer CV scores




## Computational Cost:

$$O(k_{outer} \times k_{inner} \times \text{training time})$$

⚠ Can be expensive but gives unbiased estimates!

# Nested CV Implementation

```python
from sklearn.model_selection import GridSearchCV, cross_val_score
from sklearn.ensemble import RandomForestClassifier
from sklearn.model_selection import StratifiedKFold
# Define hyperparameter grid
param_grid = {'n_estimators': [50, 100, 200],'max_depth': [3, 5, 7, None],'min_samples_split': [2, 5, 10]}
# Initialize model
rf = RandomForestClassifier(random_state=42)
# Outer CV for unbiased evaluation
outer_cv = StratifiedKFold(n_splits=5, shuffle=True, random_state=42)
# Store outer CV scores
nested_scores = []
for train_idx, test_idx in outer_cv.split(X, y):
    X_train_outer, X_test_outer = X[train_idx], X[test_idx]
    y_train_outer, y_test_outer = y[train_idx], y[test_idx]
    # Inner CV for hyperparameter tuning
    inner_cv = StratifiedKFold(n_splits=3, shuffle=True, random_state=42)
    # Grid search with inner CV
    grid_search = GridSearchCV(
        rf, param_grid, cv=inner_cv,
        scoring='accuracy', n_jobs=-1)
    # Fit on outer training data
    grid_search.fit(X_train_outer, y_train_outer)
    # Get best model and evaluate on outer test data
    best_model = grid_search.best_estimator_
    outer_score = best_model.score(X_test_outer, y_test_outer)
    nested_scores.append(outer_score)
    print(f"Best params: {grid_search.best_params_}")
    print(f"Outer CV score: {outer_score:.4f}")
print(f"\nNested CV Score: {np.mean(nested_scores):.4f} +- {np.std(nested_scores):.4f}")
```

# Model Selection and Comparison

**Comparing Multiple Models:**

- Use same CV splits for all models
- Calculate statistical significance
- Consider computational costs
- Report confidence intervals

**Statistical Tests:**

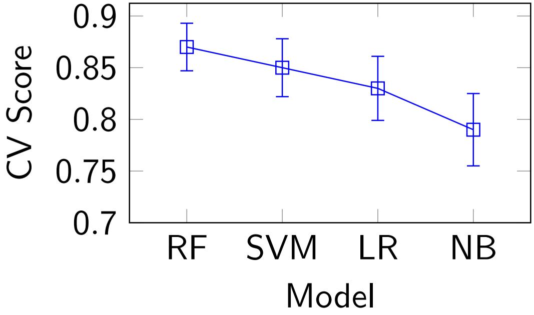
- Paired t-test: Compare two models
- McNemar's test: Classification tasks
- Wilcoxon signed-rank: Non-parametric alternative
- Friedman test: Multiple models

**Effect Size:**

- Don't just look at p-values
- Consider practical significance
- Cohen's d for effect size

**Model Comparison Results:**

| Model | CV Score | Std Dev | p-value |
|---|---|---|---|
| Random Forest | 0.87 | 0.023 | - |
| SVM | 0.85 | 0.028 | 0.045 |
| Logistic Reg. | 0.83 | 0.031 | 0.012 |
| Naive Bayes | 0.79 | 0.035 | 0.001 |




Error bars show confidence intervals

# Practical Guidelines for CV

**Choosing the Right CV Method:**

- Standard datasets: 5 or 10-fold CV
- Small datasets: LOOCV or higher k
- Imbalanced data: Stratified CV
- Time series: Temporal CV methods
- Grouped data: Group-based CV

**Computational Considerations:**

- Balance between bias and variance
- Consider training time
- Use parallel processing when possible
- Cache intermediate results

**Reporting CV Results:**

- Mean +- standard deviation
- Confidence intervals
- Individual fold results
- Statistical significance tests
- Computational time

**Best Practices Checklist:**

- ✓ Preprocessing inside CV
- ✓ Appropriate CV method for data type
- ✓ Nested CV for hyperparameter tuning
- ✓ Statistical significance testing
- ✓ Reproducible random seeds

**Remember:** CV estimates performance, not the final model!

# Cross-Validation: Key Takeaways

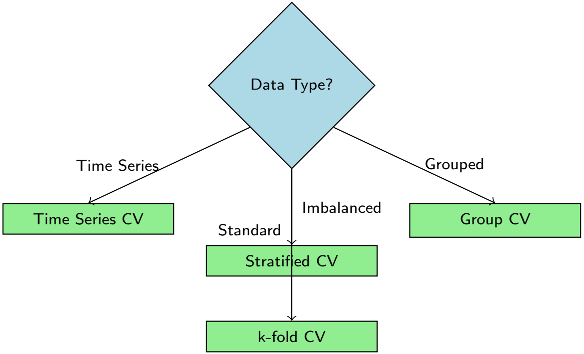
## What We Learned:

- k-fold CV: Balanced approach for most tasks
- LOOCV: Maximum data usage, high variance
- Stratified CV: Essential for imbalanced data
- Specialized methods: Time series, grouped data
- Nested CV: Unbiased hyperparameter tuning

## Critical Principles:

- No data leakage between folds
- Preprocessing inside CV loop
- Choose appropriate CV for your data
- Report uncertainty in estimates
- Statistical significance matters

## CV Method Selection Guide:




## Performance Estimation Hierarchy:

1. Training accuracy (Worst)
2. Simple train/validation split
3. k-fold cross-validation
4. Nested cross-validation (Best)

# Questions?

sali85@student.gsu.edu

**Next Topic:** Bias-Variance Tradeoff: Overfitting, underfitting, model complexity

"Cross-validation is not just a technique—it's a mindset for honest model evaluation."