# Cross-Validation Complete Guide

## 1. What is Cross-Validation?

Cross-validation (CV) is a **resampling technique** used to evaluate machine learning models by training and testing on different portions of the data multiple times.

**Purpose:**

- Estimate how well a model generalizes to unseen data
- Reduce variance in performance estimates
- Detect overfitting
- Compare different models or hyperparameters

**Key Idea:** Use your available data more efficiently by testing on multiple different subsets.

---

## 2. The Problem: Single Train/Test Split

### Issues with Simple Split

```
Dataset → [Training 80%][Testing 20%]
           Train model    Evaluate
```

**Problems:**

1. **High variance** - performance depends on which samples ended up in test set
2. **Waste of data** - 20% never used for training
3. **Unreliable** - single estimate may not reflect true performance
4. **Lucky/unlucky splits** - test set might be easier or harder than typical

### Example of Variance

```
Split 1: Test Accuracy = 92%
Split 2: Test Accuracy = 85%
Split 3: Test Accuracy = 88%
```

Which is the true performance? We don't know from a single split.

---

## 3. K-Fold Cross-Validation

### How It Works

**Steps:**

1. Split data into K equal-sized folds (subsets)
2. For each fold i = 1 to K:
    - Use fold i as test set
    - Use remaining K-1 folds as training set
    - Train model and evaluate on test fold
    - Record performance metric (accuracy, F1, etc.)
3. Average the K performance scores

### Visual Representation (K=5)

```
Original Data: [████████████████████████████████████████]

Fold 1: [TEST][TRAIN----------------------------]
Fold 2: [TRAIN][TEST][TRAIN---------------------]
Fold 3: [TRAIN------][TEST][TRAIN---------------]
Fold 4: [TRAIN-----------][TEST][TRAIN----------]
Fold 5: [TRAIN----------------][TEST][TRAIN-----]
        Round 1 Round 2 Round 3 Round 4 Round 5
```

**Result:** Each sample is tested exactly once, trained on K-1 times.

### Mathematical Formula

**Cross-Validation Score:**

```
CV_score = (1/K) × Σ score_i  for i=1 to K
```

**Standard Deviation:**

```
σ = √[(1/K) × Σ(score_i - CV_score)²]
```

**Standard Error:**

```
SE = σ / √K
```

**Confidence Interval (95%):**

```
CI = CV_score ± 1.96 × SE
```

### Example Calculation

```
5-Fold CV Results:
Fold 1: 0.85
Fold 2: 0.88
Fold 3: 0.82
Fold 4: 0.87
Fold 5: 0.86

Mean = (0.85 + 0.88 + 0.82 + 0.87 + 0.86) / 5 = 0.856
Std  = 0.0219
SE   = 0.0219 / √5 = 0.0098

Result: 85.6% ± 0.98%
```

---

# 4. Types of Cross-Validation

## 4.1 Standard K-Fold CV

**Use Case:** General purpose, most common

**Characteristics:**

- K typically 5 or 10
- Random split (shuffled data)
- Each fold roughly equal size

**Pros:**

- Good balance of bias and variance
- Computationally reasonable

**Cons:**

- May not preserve class distribution
- Random splits can vary

```python
from sklearn.model_selection import KFold

kfold = KFold(n_splits=5, shuffle=True, random_state=42)
for train_idx, test_idx in kfold.split(X):
    X_train, X_test = X[train_idx], X[test_idx]
    # Train and evaluate
```

---

## 4.2 Stratified K-Fold CV ★

**Use Case:** Classification with imbalanced classes

**Characteristics:**

- Maintains class distribution in each fold
- Each fold has same proportion of each class as original dataset

**Example:**

```
Original: 70% Class A, 30% Class B

Each fold will also have:
- 70% Class A
- 30% Class B
```

**Why Important:**

- Ensures representative testing
- Critical for imbalanced datasets
- More stable estimates

```python
from sklearn.model_selection import StratifiedKFold

skfold = StratifiedKFold(n_splits=5, shuffle=True, random_state=42)
for train_idx, test_idx in skfold.split(X, y):
    X_train, X_test = X[train_idx], X[test_idx]
    y_train, y_test = y[train_idx], y[test_idx]
```

**Rule:** Always use Stratified K-Fold for classification!

---

### 4.3 Leave-One-Out CV (LOOCV)

**Use Case:** Very small datasets

**Characteristics:**

- K = n (number of samples)
- Each iteration uses 1 sample for testing
- Train on n-1 samples

**Visual:**

```
n=5 samples
Fold 1: [TEST][TRAIN--------------]
Fold 2: [TRAIN][TEST][TRAIN-------]
Fold 3: [TRAIN------][TEST][TRAIN]
Fold 4: [TRAIN-----------][TEST]---
Fold 5: [TRAIN----------------][TEST]
```

**Formula:**

```
LOOCV_score = (1/n) × Σ score_i  for i=1 to n
```

**Pros:**

- Maximum use of data (train on n-1)
- No randomness (deterministic)
- Lowest bias

**Cons:**

- Very expensive (n model trainings)
- High variance in estimates
- No stratification possible

```
from sklearn.model_selection import LeaveOneOut

loo = LeaveOneOut()
for train_idx, test_idx in loo.split(X):
    X_train, X_test = X[train_idx], X[test_idx]
```

**When to use:** n < 100 and computational cost acceptable

---

### 4.4 Repeated K-Fold CV

**Use Case:** Need more robust estimates

**Characteristics:**

- Repeat K-fold multiple times with different random seeds
- Averages over multiple K-fold runs

**Example:** 5-Fold repeated 3 times = 15 total evaluations

**Formula:**

```
Repeated_CV = (1/(K×R)) × ΣΣ score_{k,r}
```

where R = number of repeats

**Pros:**

- More robust estimate
- Lower variance than single K-fold
- Good for hyperparameter tuning

**Cons:**

- More computationally expensive
- Diminishing returns after ~3-5 repeats

```python
from sklearn.model_selection import RepeatedKFold

rkfold = RepeatedKFold(n_splits=5, n_repeats=3, random_state=42)
```

---

### 4.5 Time Series CV (Temporal)

**Use Case:** Time-dependent data (stock prices, sales, etc.)

**Characteristics:**

- Always train on past data, test on future
- Preserves temporal order
- No shuffling!

**Visual:**

```
Timeline: [████████████████████████████████████]

Split 1:  [Train][Test]----------------------
Split 2:  [Train-----][Test]-----------------
Split 3:  [Train----------][Test]------------
Split 4:  [Train---------------][Test]-------
Split 5:  [Train--------------------][Test]--
```

**Why Different:**

- Cannot shuffle (breaks temporal dependency)
- Training set always comes before test set
- Training set grows over time (or slides)

**Types:**

**1. Expanding Window (Growing)**

```python
from sklearn.model_selection import TimeSeriesSplit

tscv = TimeSeriesSplit(n_splits=5)
for train_idx, test_idx in tscv.split(X):
    # Training set grows each iteration
```

**2. Rolling Window (Sliding)**

```
Fixed window size, slides forward
Split 1: [Train-----][Test]------------------
Split 2: -----[Train-----][Test]-------------
Split 3: ----------[Train-----][Test]--------
```

**Critical:** Never use regular K-Fold for time series!

---

### 4.6 Group K-Fold CV

**Use Case:** Data has natural groups (patients, users, sessions)

**Characteristics:**

- Ensures groups don't appear in both train and test
- Prevents data leakage

**Example:**

```
Patient A: [samples 1, 2, 3]
Patient B: [samples 4, 5]
Patient C: [samples 6, 7, 8]

Group CV ensures:
- All Patient A samples in train OR test, not both
```

**Why Important:**

- Medical data (multiple samples per patient)
- User data (multiple sessions per user)
- Temporal groups (data from same day)

```python
from sklearn.model_selection import GroupKFold

groups = [1, 1, 1, 2, 2, 3, 3, 3]  # Group labels
gkfold = GroupKFold(n_splits=3)
for train_idx, test_idx in gkfold.split(X, y, groups):
    # Groups don't overlap
```

---

### 4.7 Stratified Group K-Fold

**Use Case:** Classification + grouped data + imbalanced classes

**Combines:**

- Group K-Fold (no group leakage)

- Stratified K-Fold (preserve class distribution)

```
from sklearn.model_selection import StratifiedGroupKFold

sgkfold = StratifiedGroupKFold(n_splits=5)
for train_idx, test_idx in sgkfold.split(X, y, groups):
    # Both stratified AND grouped
```

---

## 5. Choosing K

### Guidelines

| K Value | Use Case | Pros | Cons |
|---|---|---|---|
| K=3 | Large datasets, quick experiments | Fast | Higher variance |
| K=5 | **Default choice** | Good balance | Standard |
| K=10 | More reliable estimates | Lower variance | Slower |
| K=n (LOOCV) | Small datasets (n<100) | Maximum data use | Very slow, high variance |

### Trade-offs

**Bias-Variance Trade-off:**

```
Small K (e.g., 3)  → Higher variance, lower bias
Large K (e.g., 10) → Lower variance, higher bias
```

**Computational Trade-off:**

```
Training time = K × (time to train one model)
```

### Recommendations

**Dataset Size:**

- n < 100: Use K=n (LOOCV) or K=10
- 100 < n < 1000: Use K=10
- 1000 < n < 10000: Use K=5
- n > 10000: Use K=3 or single train/test split

**Resource Constraints:**

- Limited time: K=3
- GPU training: Smaller K

- Need precision: Larger K or repeated CV

---

## 6. Cross-Validation for Model Evaluation

### Simple Example

```python
from sklearn.model_selection import cross_val_score
from sklearn.ensemble import RandomForestClassifier

model = RandomForestClassifier()
scores = cross_val_score(model, X, y, cv=5, scoring='accuracy')

print(f"Accuracy: {scores.mean():.3f} (+/- {scores.std() * 2:.3f})")
print(f"Individual folds: {scores}")
```

### Multiple Metrics

```python
from sklearn.model_selection import cross_validate

scoring = ['accuracy', 'precision', 'recall', 'f1']
scores = cross_validate(model, X, y, cv=5, scoring=scoring)

for metric in scoring:
    print(f"{metric}: {scores[f'test_{metric}'].mean():.3f}")
```

### Custom Scoring

```python
from sklearn.metrics import make_scorer, f1_score

# Custom scorer
custom_scorer = make_scorer(f1_score, average='weighted')
scores = cross_val_score(model, X, y, cv=5, scoring=custom_scorer)
```

---

## 7. Cross-Validation for Hyperparameter Tuning

### Grid Search CV

**Exhaustive search over parameter grid**

```python
from sklearn.model_selection import GridSearchCV
from sklearn.svm import SVC

# Define parameter grid
```

```
param_grid = {
    'C': [0.1, 1, 10, 100],
    'gamma': [0.001, 0.01, 0.1, 1],
    'kernel': ['rbf', 'poly']
}

# Grid search with 5-fold CV
grid = GridSearchCV(
    SVC(),
    param_grid,
    cv=5,
    scoring='accuracy',
    n_jobs=-1,  # Use all CPUs
    verbose=1
)

grid.fit(X_train, y_train)

print(f"Best parameters: {grid.best_params_}")
print(f"Best CV score: {grid.best_score_:.3f}")
print(f"Test score: {grid.score(X_test, y_test):.3f}")
```

**Total fits:** K × (number of parameter combinations)

Example: 5-fold CV × 32 combinations = 160 model fits

---

## Random Search CV

**Random sampling from parameter distributions**

```
from sklearn.model_selection import RandomizedSearchCV
from scipy.stats import uniform, randint

# Define parameter distributions
param_dist = {
    'C': uniform(0.1, 100),
    'gamma': uniform(0.001, 1),
    'kernel': ['rbf', 'poly']
}

# Random search
random_search = RandomizedSearchCV(
    SVC(),
    param_distributions=param_dist,
    n_iter=50,  # Number of random combinations
    cv=5,
    random_state=42,
    n_jobs=-1
)

random_search.fit(X_train, y_train)
```

**Advantages over Grid Search:**

- Much faster for large parameter spaces
- Can search continuous distributions
- Often finds good parameters with fewer fits

---

### Nested Cross-Validation

**For unbiased performance estimation**

```
Outer loop (K1 folds): Estimate model performance
Inner loop (K2 folds): Tune hyperparameters
```

**Why Needed:**

- Grid search CV gives optimistic estimates
- Hyperparameters tuned on same data used for evaluation
- Nested CV provides unbiased estimate

```
from sklearn.model_selection import cross_val_score, GridSearchCV

# Inner CV: Hyperparameter tuning
inner_cv = GridSearchCV(
    SVC(),
    param_grid={'C': [0.1, 1, 10], 'gamma': [0.01, 0.1, 1]},
    cv=3  # Inner folds
)

# Outer CV: Performance estimation
outer_scores = cross_val_score(
    inner_cv,
    X, y,
    cv=5  # Outer folds
)

print(f"Nested CV score: {outer_scores.mean():.3f}")
```

**Total fits:** K_outer × K_inner × n_params
Example: 5 × 3 × 9 = 135 fits

---

## 8. Complete Workflow

### Proper Train/Val/Test Strategy

```
Full Dataset (100%)
│
├── Training Set (60-80%)
│   │
│   └── Use for Cross-Validation
│       ├── Fold 1 (train/val)
│       ├── Fold 2 (train/val)
│       ├── Fold 3 (train/val)
│       └── ...
│
│   Purpose:
│   - Model training
│   - Hyperparameter tuning
│   - Model selection
│
└── Test Set (20-40%)
    │
    └── Touch ONLY at the very end

    Purpose:
    - Final performance estimate
    - Simulate real-world deployment
```

### Step-by-Step Process

```python
from sklearn.model_selection import train_test_split, GridSearchCV
from sklearn.preprocessing import StandardScaler
from sklearn.svm import SVC

# Step 1: Initial split (hold out test set)
X_train, X_test, y_train, y_test = train_test_split(
    X, y, test_size=0.2, random_state=42, stratify=y
)

# Step 2: Feature scaling (fit on train only!)
scaler = StandardScaler()
X_train_scaled = scaler.fit_transform(X_train)
X_test_scaled = scaler.transform(X_test)  # Use train statistics

# Step 3: Hyperparameter tuning with CV on training set
param_grid = {'C': [0.1, 1, 10], 'gamma': [0.01, 0.1, 1]}
grid = GridSearchCV(SVC(), param_grid, cv=5)
grid.fit(X_train_scaled, y_train)

print(f"Best params: {grid.best_params_}")
print(f"Best CV score: {grid.best_score_:.3f}")

# Step 4: Final evaluation on test set
test_score = grid.score(X_test_scaled, y_test)
print(f"Test score: {test_score:.3f}")
```

**Critical:** Test set is touched only once at the end!

---

## 9. Common Mistakes & How to Avoid Them

### ❌ Mistake 1: Data Leakage via Scaling

**Wrong:**

```
# DON'T DO THIS!
X_scaled = scaler.fit_transform(X)  # Scale all data
for train_idx, test_idx in kfold.split(X_scaled):
    # Test data statistics already leaked into training
```

**Correct:**

```
for train_idx, test_idx in kfold.split(X):
    X_train, X_test = X[train_idx], X[test_idx]

    # Fit scaler on training data only
    scaler = StandardScaler()
    X_train_scaled = scaler.fit_transform(X_train)
    X_test_scaled = scaler.transform(X_test)  # Use train statistics
```

---

### ❌ Mistake 2: Using Test Set for Hyperparameter Tuning

**Wrong:**

```
# DON'T DO THIS!
for C in [0.1, 1, 10]:
    model = SVC(C=C)
    model.fit(X_train, y_train)
    score = model.score(X_test, y_test)  # Testing on test set!
    # Pick best C based on test scores
```

**Correct:**

```
# Use cross-validation on training set
grid = GridSearchCV(SVC(), {'C': [0.1, 1, 10]}, cv=5)
grid.fit(X_train, y_train)

# Test set touched only once at the end
final_score = grid.score(X_test, y_test)
```

---

### ❌ Mistake 3: Shuffling Time Series Data

**Wrong:**

```
# DON'T DO THIS for time series!
kfold = KFold(n_splits=5, shuffle=True)  # Breaks temporal order
```

**Correct:**

```
tscv = TimeSeriesSplit(n_splits=5)  # Preserves temporal order
```

---

### ❌ Mistake 4: Not Stratifying Classification Data

**Wrong:**

```
# Regular K-Fold for imbalanced classes
kfold = KFold(n_splits=5)
```

**Correct:**

```
# Use Stratified K-Fold
skfold = StratifiedKFold(n_splits=5)
```

---

### ❌ Mistake 5: Ignoring Groups

**Wrong:**

```
# Multiple samples per patient, but using regular CV
kfold = KFold(n_splits=5)  # Patient samples can leak!
```

**Correct:**

```
# Ensure patient groups don't leak
gkfold = GroupKFold(n_splits=5)
for train_idx, test_idx in gkfold.split(X, y, groups=patient_ids):
    # No patient appears in both train and test
```

---

### ❌ Mistake 6: Feature Selection Before Split

**Wrong:**

```
# Feature selection on all data first
X_selected = select_features(X, y)  # Uses all data including test!
X_train, X_test = train_test_split(X_selected)
```

**Correct:**

```
# Split first, then select features
X_train, X_test, y_train, y_test = train_test_split(X, y)
X_train_selected = select_features(X_train, y_train)  # Fit on train only
X_test_selected = apply_selection(X_test)  # Apply to test
```

## 10. Interpreting Results

### Understanding CV Scores

```
scores = cross_val_score(model, X, y, cv=5)
# Output: [0.85, 0.88, 0.82, 0.87, 0.86]
```

**Mean:** Average performance (85.6%)
**Std:** Variability across folds (0.022)
**Min/Max:** Range of performance (82% - 88%)

### What Different Patterns Mean

**Pattern 1: Low mean, low std**

```
Scores: [0.60, 0.62, 0.61, 0.59, 0.60]
Mean: 0.604, Std: 0.010
```

→ Consistently poor performance. Model is underfitting.

**Pattern 2: High mean, low std**

```
Scores: [0.91, 0.92, 0.91, 0.92, 0.91]
Mean: 0.914, Std: 0.006
```

→ Consistently good performance. Model is well-tuned!

**Pattern 3: High mean, high std**

```
Scores: [0.75, 0.95, 0.70, 0.92, 0.78]
Mean: 0.820, Std: 0.106
```

→ Unstable performance. Model is overfitting or data has high variance.

**Pattern 4: Low mean, high std**

```
Scores: [0.45, 0.70, 0.40, 0.65, 0.50]
Mean: 0.540, Std: 0.124
```

→ Unreliable model. Poor and inconsistent.

### Comparing Models

```
model1_scores = [0.85, 0.88, 0.82, 0.87, 0.86]  # Mean: 0.856
model2_scores = [0.84, 0.87, 0.83, 0.86, 0.85]  # Mean: 0.850
```

**Statistical Test (Paired t-test):**

```
from scipy.stats import ttest_rel

statistic, pvalue = ttest_rel(model1_scores, model2_scores)
if pvalue < 0.05:
    print("Significant difference")
else:
    print("No significant difference")
```

---

## 11. Computational Considerations

### Time Complexity

**K-Fold CV:**

```
Time = K × (training_time + evaluation_time)
```

**Grid Search CV:**

```
Time = K × n_params × (training_time + evaluation_time)
```

**Nested CV:**

```
Time = K_outer × K_inner × n_params × (training_time + evaluation_time)
```

### Speeding Up CV

**1. Use fewer folds (K=3 instead of K=10)**

```
cv=3  # 3.3x faster than K=10
```

**2. Parallel processing**

```
cross_val_score(model, X, y, cv=5, n_jobs=-1)  # Use all CPUs
```

**3. Random search instead of grid search**

```
RandomizedSearchCV(model, param_dist, n_iter=20)  # Sample 20 instead of all
```

**4. Early stopping**

```
# For neural networks, tree ensembles
model = XGBClassifier(early_stopping_rounds=10)
```

**5. Use faster models for initial exploration**

```
# Start with linear models, then try complex models
LogisticRegression() → RandomForest() → XGBoost()
```

---

## 12. Advanced Topics

### 12.1 Cross-Validation for Imbalanced Data

**Combine with Stratification:**

```
from sklearn.model_selection import StratifiedKFold

# Ensures minority class in all folds
skfold = StratifiedKFold(n_splits=5)
```

**With SMOTE (Synthetic Minority Oversampling):**

```
from imblearn.over_sampling import SMOTE
from imblearn.pipeline import Pipeline

pipeline = Pipeline([
    ('smote', SMOTE()),
    ('classifier', SVC())
])

# SMOTE applied within each fold (no leakage)
scores = cross_val_score(pipeline, X, y, cv=5)
```

---

### 12.2 Monte Carlo Cross-Validation

**Random repeated train/test splits**

```
from sklearn.model_selection import ShuffleSplit

mccv = ShuffleSplit(n_splits=10, test_size=0.2, random_state=42)
scores = cross_val_score(model, X, y, cv=mccv)
```

**Difference from K-Fold:**

- Samples can appear in multiple test sets
- Samples might not appear in any test set
- More randomness, useful for uncertainty estimation

---

### 12.3 Bootstrap Cross-Validation

**Sample with replacement**

```
from sklearn.utils import resample

n_iterations = 100
scores = []

for i in range(n_iterations):
    # Bootstrap sample
    X_boot, y_boot = resample(X, y, n_samples=len(X))

    # Out-of-bag samples as test set
    oob_indices = set(range(len(X))) - set(X_boot.index)
    X_test = X.iloc[list(oob_indices)]
    y_test = y.iloc[list(oob_indices)]

    model.fit(X_boot, y_boot)
    scores.append(model.score(X_test, y_test))
```

---

### 12.4 Learning Curves

**Diagnose bias/variance with CV**

```
from sklearn.model_selection import learning_curve

train_sizes, train_scores, val_scores = learning_curve(
    model, X, y,
    cv=5,
    train_sizes=np.linspace(0.1, 1.0, 10),
    scoring='accuracy'
)

# Plot to diagnose overfitting/underfitting
```

**Interpretation:**

- Large gap: Overfitting (high variance)
- Both low: Underfitting (high bias)
- Both high, close: Well-fitted

---

## 13. Practical Examples

### Example 1: Binary Classification

```
from sklearn.datasets import make_classification
from sklearn.model_selection import cross_val_score, StratifiedKFold
from sklearn.ensemble import RandomForestClassifier
from sklearn.preprocessing import StandardScaler
from sklearn.pipeline import Pipeline

# Generate data
X, y = make_classification(n_samples=1000, n_features=20, random_state=42)

# Create pipeline
pipeline = Pipeline([
    ('scaler', StandardScaler()),
    ('classifier', RandomForestClassifier(random_state=42))
])

# Stratified 5-fold CV
cv = StratifiedKFold(n_splits=5, shuffle=True, random_state=42)
scores = cross_val_score(pipeline, X, y, cv=cv, scoring='f1')

print(f"F1 Score: {scores.mean():.3f} (+/- {scores.std() * 2:.3f})")
```

---

### Example 2: Regression

```
from sklearn.datasets import make_regression
from sklearn.model_selection import cross_val_score
from sklearn.ensemble import GradientBoostingRegressor
from sklearn.metrics import make_scorer, mean_squared_error
import numpy as np

# Generate data
X, y = make_regression(n_samples=1000, n_features=10, random_state=42)

# Custom scorer (RMSE)
rmse_scorer = make_scorer(
    lambda y_true, y_pred: np.sqrt(mean_squared_error(y_true, y_pred)),
    greater_is_better=False
)

# 10-fold CV
model = GradientBoostingRegressor(random_state=42)
scores = cross_val_score(model, X, y, cv=10, scoring=rmse_scorer)

print(f"RMSE: {-scores.mean():.3f} (+/- {scores.std() * 2:.3f})")
```

---

### Example 3: Time Series

```
from sklearn.model_selection import TimeSeriesSplit
import pandas as pd

# Time series data
dates = pd.date_range('2020-01-01', periods=1000)
X = pd.DataFrame({'date': dates, 'feature1': np.random.rand(1000)})
y = np.random.rand(1000)

# Time series CV
tscv = TimeSeriesSplit(n_splits=5)

for fold, (train_idx, test_idx) in enumerate(tscv.split(X), 1):
    X_train, X_test = X.iloc[train_idx], X.iloc[test_idx]
    y_train, y_test = y[train_idx], y[test_idx]

    print(f"Fold {fold}:")
    print(f"  Train: {X_train['date'].min()} to {X_train['date'].max()}")
    print(f"  Test:  {X_test['date'].min()} to {X_test['date'].max()}")
```

---

### Example 4: Hyperparameter Tuning

```
from sklearn.model_selection import RandomizedSearchCV
from sklearn.ensemble import RandomForestClassifier
from scipy.stats import randint, uniform

# Parameter distributions
param_dist = {
    'n_estimators': randint(50, 500),
    'max_depth': randint(3, 20),
    'min_samples_split': randint(2, 20),
    'min_samples_leaf': randint(1, 10),
    'max_features': uniform(0.1, 0.9)
}

# Random search
random_search = RandomizedSearchCV(
    RandomForestClassifier(random_state=42),
    param_distributions=param_dist,
    n_iter=50,
    cv=5,
    scoring='accuracy',
    n_jobs=-1,
    random_state=42,
    verbose=2
)

random_search.fit(X_train, y_train)

print(f"Best parameters: {random_search.best_params_}")
print(f"Best CV score: {random_search.best_score_:.3f}")
print(f"Test score: {random_search.score(X_test, y_test):.3f}")
```

---

## 14. Quick Reference

### Decision Tree: Which CV to Use?

```
Start
  │
  ├─ Time series data?
  │    └─ Yes → TimeSeriesSplit
```

```
    │
    ├─ Grouped data (patients/users)?
    │   ├─ Yes + Classification → StratifiedGroupKFold
    │   └─ Yes + Regression → GroupKFold
    │
    ├─ Classification?
    │   └─ Yes → StratifiedKFold
    │
    ├─ Very small dataset (n<100)?
    │   └─ Yes → LeaveOneOut or K=10
    │
    └─ Default → KFold (K=5)
```

### Common CV Methods Comparison

| Method | Use Case | K Value | Shuffle | Pros | Cons |
|---|---|---|---|---|---|
| KFold | General | 5-10 | Yes | Simple, fast | May not stratify |
| StratifiedKFold | Classification | 5-10 | Yes | Preserves class distribution | Classification only |
| TimeSeriesSplit | Time series | 5 | No | Respects temporal order | Can't shuffle |
| LeaveOneOut | Small data | n | No | Max data use | Very slow |
| GroupKFold | Grouped data | 5-10 | Optional | No group leakage | Needs group info |
| RepeatedKFold | Need precision | 5×3 | Yes | Lower variance | More expensive |

### Scoring Metrics

**Classification:**

- `accuracy`: Overall correctness
- `precision`: Positive prediction quality
- `recall`: Positive class coverage
- `f1`: Harmonic mean of precision/recall
- `roc_auc`: Area under ROC curve

**Regression:**

- `neg_mean_squared_error`: MSE (negated)
- `neg_root_mean_squared_error`: RMSE (negated)
- `neg_mean_absolute_error`: MAE (negated)
- `r2`: Coefficient of determination

---

## 15. Best Practices Summary

### ☑ DO:

1. Always use cross-validation for model evaluation
2. Use StratifiedKFold for classification
3. Use TimeSeriesSplit for time series
4. Split data BEFORE any preprocessing
5. Fit preprocessing on training folds only
6. Use at least K=5 for reliable estimates
7. Report mean and standard deviation
8. Use nested CV for hyperparameter tuning
9. Keep test set completely separate
10. Use appropriate scoring metric for your problem

### ✕ DON'T:

1. Scale/preprocess before splitting
2. Use test set for hyperparameter tuning
3. Shuffle time series data
4. Ignore data leakage in grouped data
5. Use regular K-Fold for imbalanced classification
6. Touch test set multiple times
7. Select features on all data
8. Forget to set random_state for reproducibility
9. Use K=2 (too high variance)
10. Assume single train/test split is enough

---

## 16. Conclusion

Cross-validation is an **essential tool** for reliable model evaluation and selection. Key takeaways:

1. **Better estimates** - More reliable than single train/test split
2. **Detect overfitting** - Compare train and CV scores
3. **Choose the right method** - Stratified for classification, Time Series for temporal data
4. **Avoid data leakage** - Preprocess within folds
5. **Use for hyperparameter tuning** - Grid/Random search with CV
6. **Report properly** - Mean ± standard deviation

**The Golden Rule:**
Split once (train/test), never touch test set during development, use CV on training set only, evaluate on test set once at the very end.