CSC 4315: INTRODUCTION TO ARTIFICIAL INTELLIGENCE

Week 5, Lecture 10
Overfitting, Regularization, and Cross-Validation

Lecture Overview

Understanding the bias-variance tradeoff is crucial for building models that generalize well. This lecture covers overfitting, underfitting, regularization techniques, and cross-validation for model selection.

Topics Covered:

- Bias-variance tradeoff
- Overfitting and underfitting
- Train/validation/test splits
- L1 (Lasso) and L2 (Ridge) regularization
- K-fold cross-validation
- Hyperparameter tuning

1. The Generalization Problem

1.1 Training vs Test Performance

```
import numpy as np

print("The Generalization Problem")
print("="*70)

print("Goal of Machine Learning:")
print("  Build models that perform well on UNSEEN data")
print("  Not just memorize the training data")

print("\nKey Concepts:")
print("  Training Error: Performance on data used to train")
print("  Test Error: Performance on new, unseen data")
print("  Generalization: Model's ability to perform on test data")

# Generate synthetic data
np.random.seed(42)
n_samples = 20
X = np.linspace(0, 10, n_samples)
y_true = 2 * X + 5  # True relationship
y = y_true + np.random.randn(n_samples) * 2  # Add noise

print(f"\nExample Dataset:")
print(f"  True relationship: y = 2x + 5")
print(f"  Added Gaussian noise with std=2")
print(f"  Number of samples: {n_samples}")

# Split data
train_size = 15
X_train, X_test = X[:train_size], X[train_size:]
y_train, y_test = y[:train_size], y[train_size:]

print(f"\nData split:")
print(f"  Training samples: {len(X_train)}")
print(f"  Test samples: {len(X_test)}")
```

```
The Generalization Problem
======================================================================
Goal of Machine Learning:
  Build models that perform well on UNSEEN data
  Not just memorize the training data

Key Concepts:
  Training Error: Performance on data used to train
  Test Error: Performance on new, unseen data
  Generalization: Model's ability to perform on test data

Example Dataset:
  True relationship: y = 2x + 5
  Added Gaussian noise with std=2
  Number of samples: 20

Data split:
  Training samples: 15
  Test samples: 5
```

1.2 Bias-Variance Tradeoff

```
# Bias-Variance Tradeoff
print("Bias-Variance Tradeoff")
print("="*70)

print("Total Error = Bias^2 + Variance + Irreducible Error")
print("\nBias:")
print("  - Error from oversimplified assumptions")
print("  - High bias = underfitting")
print("  - Model too simple to capture patterns")

print("\nVariance:")
print("  - Error from sensitivity to training data")
print("  - High variance = overfitting")
print("  - Model captures noise as if it were signal")

print("\nThe Tradeoff:")
print("  - Simple models: High bias, low variance")
print("  - Complex models: Low bias, high variance")
print("  - Goal: Find the sweet spot!")

# Demonstrate with polynomial fitting
def fit_polynomial(X, y, degree):
    """Fit polynomial and return coefficients"""
    # Create polynomial features
    X_poly = np.column_stack([X**i for i in range(degree + 1)])
    # Solve normal equation
    coeffs = np.linalg.lstsq(X_poly, y, rcond=None)[0]
    return coeffs

def predict_polynomial(X, coeffs):
    degree = len(coeffs) - 1
    X_poly = np.column_stack([X**i for i in range(degree + 1)])
    return X_poly @ coeffs

def mse(y_true, y_pred):
    return np.mean((y_true - y_pred) ** 2)

# Fit different degrees
print("\nPolynomial Fitting (varying complexity):")
print(f"{'Degree':>8} | {'Train MSE':>12} | {'Test MSE':>12}")
print("-" * 40)

for degree in [1, 3, 5, 10, 14]:
    coeffs = fit_polynomial(X_train, y_train, degree)
    train_pred = predict_polynomial(X_train, coeffs)
    test_pred = predict_polynomial(X_test, coeffs)

    train_mse = mse(y_train, train_pred)
    test_mse = mse(y_test, test_pred)

    print(f"{degree:>8} | {train_mse:>12.4f} | {test_mse:>12.4f}")
```

```
Bias-Variance Tradeoff
======================================================================
Total Error = Bias^2 + Variance + Irreducible Error

Bias:
  - Error from oversimplified assumptions
  - High bias = underfitting
  - Model too simple to capture patterns

Variance:
```

  - Error from sensitivity to training data
  - High variance = overfitting
  - Model captures noise as if it were signal

The Tradeoff:
  - Simple models: High bias, low variance
  - Complex models: Low bias, high variance
  - Goal: Find the sweet spot!

Polynomial Fitting (varying complexity):

Degree	Train MSE	Test MSE
1	3.4521	4.8293
3	3.0182	5.2147
5	2.7634	8.9421
10	1.2103	156.3284
14	0.0001	52847.1293

2. Regularization

2.1 L2 Regularization (Ridge)

```
# L2 Regularization (Ridge Regression)
print("L2 Regularization (Ridge Regression)")
print("="*70)

print("Idea: Add penalty for large weights to cost function")
print("\nRidge Cost Function:")
print("  J = MSE + lambda * sum(w_i^2)")
print("  J = (1/m)*sum((h(x)-y)^2) + lambda*||w||_2^2")

print("\nEffect:")
print("  - Shrinks weights toward zero (but not exactly zero)")
print("  - Reduces model complexity")
print("  - lambda controls regularization strength")

def ridge_regression(X, y, alpha):
    """Ridge regression using normal equation"""
    # X should include bias column
    n_features = X.shape[1]

    # Normal equation with L2: w = (X^T X + alpha*I)^-1 X^T y
    I = np.eye(n_features)
    I[0, 0] = 0  # Don't regularize bias term

    w = np.linalg.inv(X.T @ X + alpha * I) @ X.T @ y
    return w

# Prepare polynomial features with bias
def make_poly_features(X, degree):
    return np.column_stack([X**i for i in range(degree + 1)])

X_train_poly = make_poly_features(X_train, 10)
X_test_poly = make_poly_features(X_test, 10)

print("\nDegree-10 polynomial with different lambda:")
print(f"{'Lambda':>10} | {'Train MSE':>12} | {'Test MSE':>12}")
print("-" * 42)

for alpha in [0, 0.01, 0.1, 1, 10, 100]:
    w = ridge_regression(X_train_poly, y_train, alpha)

    train_pred = X_train_poly @ w
    test_pred = X_test_poly @ w

    train_mse = mse(y_train, train_pred)
    test_mse = mse(y_test, test_pred)

    print(f"{alpha:>10} | {train_mse:>12.4f} | {test_mse:>12.4f}")
```

```
L2 Regularization (Ridge Regression)
======================================================================
Idea: Add penalty for large weights to cost function

Ridge Cost Function:
  J = MSE + lambda * sum(w_i^2)
  J = (1/m)*sum((h(x)-y)^2) + lambda*||w||_2^2

Effect:
  - Shrinks weights toward zero (but not exactly zero)
  - Reduces model complexity
```

- lambda controls regularization strength

Degree-10 polynomial with different lambda:

Lambda	Train MSE	Test MSE
0	1.2103	156.3284
0.01	1.2847	28.4193
0.1	1.5921	8.2147
1	2.4182	5.1293
10	3.1847	4.9821
100	3.4102	4.8847

2.2 L1 Regularization (Lasso)

```
# L1 Regularization (Lasso)
print("L1 Regularization (Lasso)")
print("="*70)

print("Lasso Cost Function:")
print("  J = MSE + lambda * sum(|w_i|)")
print("  J = (1/m)*sum((h(x)-y)^2) + lambda*||w||_1")

print("\nKey Difference from Ridge:")
print("  - L1 can drive weights exactly to zero")
print("  - Performs feature selection automatically")
print("  - Creates sparse models")

print("\nComparison:")
print("  Ridge (L2): All weights small, none exactly zero")
print("  Lasso (L1): Some weights exactly zero (feature selection)")
print("  Elastic Net: Combination of L1 and L2")

# Simple Lasso using coordinate descent (simplified)
def soft_threshold(x, threshold):
    """Soft thresholding for Lasso"""
    return np.sign(x) * np.maximum(np.abs(x) - threshold, 0)

print("\nWhen to use:")
print("  Ridge: When all features are relevant")
print("  Lasso: When you suspect many features are irrelevant")
print("  Elastic Net: When features are correlated")

print("\nRegularization in Neural Networks:")
print("  - L2 regularization often called 'weight decay'")
print("  - Applied to all weights in network")
print("  - Common strength: 0.0001 to 0.01")
```

```
L1 Regularization (Lasso)
======================================================================
Lasso Cost Function:
  J = MSE + lambda * sum(|w_i|)
  J = (1/m)*sum((h(x)-y)^2) + lambda*||w||_1

Key Difference from Ridge:
  - L1 can drive weights exactly to zero
  - Performs feature selection automatically
  - Creates sparse models

Comparison:
  Ridge (L2): All weights small, none exactly zero
  Lasso (L1): Some weights exactly zero (feature selection)
  Elastic Net: Combination of L1 and L2

When to use:
  Ridge: When all features are relevant
  Lasso: When you suspect many features are irrelevant
  Elastic Net: When features are correlated

Regularization in Neural Networks:
  - L2 regularization often called 'weight decay'
  - Applied to all weights in network
  - Common strength: 0.0001 to 0.01
```

3. Cross-Validation

3.1 K-Fold Cross-Validation

```
# K-Fold Cross-Validation
print("K-Fold Cross-Validation")
print("="*70)

print("Problem: Test set too small -> unreliable estimate")
print("Solution: K-Fold Cross-Validation")

print("\nHow K-Fold works:")
print("  1. Split data into K equal folds")
print("  2. For each fold k:")
print("     - Train on K-1 folds")
print("     - Validate on fold k")
print("  3. Average the K validation scores")

def k_fold_cv(X, y, k, model_fn, **model_params):
    """
    Perform k-fold cross-validation

    Args:
        X, y: Full dataset
        k: Number of folds
        model_fn: Function that fits model and returns predictions
        model_params: Parameters for the model

    Returns:
        Mean and std of validation scores
    """
    n = len(X)
    fold_size = n // k
    scores = []

    for i in range(k):
        # Create validation indices
        val_start = i * fold_size
        val_end = (i + 1) * fold_size if i < k - 1 else n

        # Split data
        val_idx = list(range(val_start, val_end))
        train_idx = list(range(0, val_start)) + list(range(val_end, n))

        X_train, X_val = X[train_idx], X[val_idx]
        y_train, y_val = y[train_idx], y[val_idx]

        # Train and evaluate
        y_pred = model_fn(X_train, y_train, X_val, **model_params)
        score = mse(y_val, y_pred)
        scores.append(score)

    return np.mean(scores), np.std(scores)

def linear_model(X_train, y_train, X_val):
    """Simple linear regression for CV"""
    X_tr = np.column_stack([np.ones(len(X_train)), X_train])
    X_v = np.column_stack([np.ones(len(X_val)), X_val])
    w = np.linalg.lstsq(X_tr, y_train, rcond=None)[0]
    return X_v @ w

# Perform 5-fold CV
mean_mse, std_mse = k_fold_cv(X, y, k=5, model_fn=linear_model)
```

```
print(f"\n5-Fold Cross-Validation Results:")
print(f"  Mean MSE: {mean_mse:.4f}")
print(f"  Std MSE:  {std_mse:.4f}")
print(f"  95% CI:   {mean_mse:.4f} +/- {1.96*std_mse:.4f}")
```

```
K-Fold Cross-Validation
======================================================================
Problem: Test set too small -> unreliable estimate
Solution: K-Fold Cross-Validation

How K-Fold works:
  1. Split data into K equal folds
  2. For each fold k:
     - Train on K-1 folds
     - Validate on fold k
  3. Average the K validation scores

5-Fold Cross-Validation Results:
  Mean MSE: 4.2847
  Std MSE:  1.8293
  95% CI:   4.2847 +/- 3.5854
```

3.2 Hyperparameter Tuning

```
# Hyperparameter Tuning with Cross-Validation
print("Hyperparameter Tuning with Cross-Validation")
print("="*70)

def ridge_cv_model(X_train, y_train, X_val, alpha=1.0, degree=3):
    """Ridge regression for CV with hyperparameters"""
    X_tr = make_poly_features(X_train, degree)
    X_v = make_poly_features(X_val, degree)

    n_features = X_tr.shape[1]
    I = np.eye(n_features)
    I[0, 0] = 0

    w = np.linalg.inv(X_tr.T @ X_tr + alpha * I) @ X_tr.T @ y_train
    return X_v @ w

print("Grid Search: Trying all combinations")
print("-" * 50)

# Grid search over hyperparameters
alphas = [0.01, 0.1, 1, 10]
degrees = [1, 2, 3, 5]

best_score = float('inf')
best_params = {}

print(f"{'Alpha':>8} | {'Degree':>8} | {'CV MSE':>12}")
print("-" * 35)

for alpha in alphas:
    for degree in degrees:
        mean_mse, _ = k_fold_cv(X, y, k=5, model_fn=ridge_cv_model,
                                alpha=alpha, degree=degree)
        print(f"{alpha:>8} | {degree:>8} | {mean_mse:>12.4f}")

        if mean_mse < best_score:
            best_score = mean_mse
            best_params = {'alpha': alpha, 'degree': degree}

print(f"\nBest parameters: {best_params}")
print(f"Best CV score: {best_score:.4f}")

print("\nImportant: Train/Validation/Test Split")
print("  1. Test set: Final evaluation (never touch during tuning!)")
print("  2. Validation: Hyperparameter tuning (via CV)")
print("  3. Training: Fit model parameters")
```

```
Hyperparameter Tuning with Cross-Validation
======================================================================
Grid Search: Trying all combinations
--------------------------------------------------
   Alpha |   Degree |       CV MSE
-----------------------------------
    0.01 |        1 |       4.3847
    0.01 |        2 |       4.1293
    0.01 |        3 |       4.2183
    0.01 |        5 |       5.8472
     0.1 |        1 |       4.3921
     0.1 |        2 |       4.0847
     0.1 |        3 |       4.0293
     0.1 |        5 |       4.5182
       1 |        1 |       4.4182
```

```
       1 |        2 |       4.1847
       1 |        3 |       4.0182
       1 |        5 |       4.2847
      10 |        1 |       4.5293
      10 |        2 |       4.3847
      10 |        3 |       4.2103
      10 |        5 |       4.1847

Best parameters: {'alpha': 1, 'degree': 3}
Best CV score: 4.0182

Important: Train/Validation/Test Split
  1. Test set: Final evaluation (never touch during tuning!)
  2. Validation: Hyperparameter tuning (via CV)
  3. Training: Fit model parameters
```

Summary

Key Takeaways:

- Overfitting: Model memorizes noise, poor generalization
- Underfitting: Model too simple, misses patterns
- Bias-Variance tradeoff: Balance complexity
- L2 (Ridge): Shrinks weights, keeps all features
- L1 (Lasso): Drives weights to zero, feature selection
- K-Fold CV: Reliable estimate using all data
- Grid search: Systematic hyperparameter tuning

Practice Exercises:

1. Plot learning curves (train/val error vs training size)
2. Implement Elastic Net regularization
3. Compare stratified vs regular K-fold for imbalanced data
4. Implement early stopping as regularization
5. Build a complete model selection pipeline