In-Class Activity: Machine Learning Basics
Bias-Variance Tradeoff with Basis Function Approximation

ECON 6343: Structural Econometrics
Prof. Tyler Ransom
University of Oklahoma

Overview

Today we'll explore fundamental machine learning concepts through hands-on coding with a high-dimensional problem. We'll generate data from a sparse nonlinear process using basis function approximation, then demonstrate how proper ML techniques (train/validation/test splitting, regularization) help us recover the true relationship while avoiding overfitting in a $p \approx n$ setting.

Learning Objectives

- Understand the bias-variance tradeoff empirically in high dimensions
- Practice train/validation/test data splitting
- Work with polynomial basis function approximation
- Compare OLS, Ridge, and LASSO regularization
- Evaluate out-of-sample prediction performance
- Visualize overfitting, underfitting, and sparsity selection

Required Packages

```
using Random, LinearAlgebra, Statistics, Optim
using DataFrames, Plots, GLM, StatsPlots
using MLJ, MLJLinearModels, Tables
```

1 Understanding the Problem Setup (10 minutes)

1.1 The High-Dimensional Challenge

We'll work with a problem that mimics many real-world econometric applications:

- **Raw features:** 2 variables (x_1, x_2)
- **Basis expansion:** Transform into 120 polynomial basis functions (up to degree 4, including interactions)
- **True sparsity:** Only 5 of the 120 basis coefficients are actually non-zero
- **Sample size:** 300 total observations (split into train/val/test)
- **Challenge:** $p \approx n$ after train split (120 features, ∼180 training obs)

This creates a realistic scenario where:

- The true relationship is sparse (most basis terms don't matter)
- We don't know *which* basis terms matter
- OLS will severely overfit
- Regularization is essential

1.2 Polynomial Basis Functions

A polynomial basis transforms raw features into higher-order terms:

$$\phi(x_1, x_2) = [x_1, x_2, x_1^2, x_2^2, x_1^3, x_2^3, x_1^4, x_2^4, x_1x_2, x_1^2x_2, \ldots]$$

The true model in our data generation is:

$$y = \sum_{j=1}^{5} \beta_j \phi_j(x_1, x_2) + \varepsilon$$

where only 5 of the 120 β coefficients are non-zero.

2 Utility Functions (15 minutes)

2.1 Basis Function Generation

```
"""
Generate polynomial basis for multivariate input
"""
function polynomial_basis(X::Matrix{Float64}, max_degree::Int)
    n, d = size(X)
    bases = []

    # Univariate polynomials for each feature
    for j in 1:d
        for deg in 1:max_degree
            push!(bases, X[:, j].^deg)
        end
    end

    # Interactions (if multiple features)
    if d > 1
        for j1 in 1:d, j2 in (j1+1):d
            for deg1 in 1:max_degree, deg2 in 1:max_degree
                if deg1 + deg2 <= max_degree
                    push!(bases, X[:, j1].^deg1 .* X[:, j2].^deg2)
                end
            end
        end
    end

    return hcat(bases...)
end
```

Task: What happens to the number of basis functions as we increase `max_degree`? Calculate for $d = 2$ and degrees 2, 3, 4.

2.2 Data Generation with Sparsity

```
"""
Generate data with sparse nonlinear structure
"""
function generate_data(n::Int, n_features::Int, n_basis::Int,
                      n_active::Int; noise_std=0.5)
    # Generate raw features
    X = randn(n, n_features)

    # Create basis expansion (up to 4th degree)
    Phi = polynomial_basis(X, 4)

    # Truncate/pad to exactly n_basis functions
    if size(Phi, 2) > n_basis
        Phi = Phi[:, 1:n_basis]
    elseif size(Phi, 2) < n_basis
        extra = n_basis - size(Phi, 2)
        Phi = hcat(Phi, randn(n, extra))
```

```julia
    end

    # True coefficients: only first n_active are non-zero
    beta_true = zeros(n_basis)
    beta_true[1:n_active] = randn(n_active) .* 6.0

    # Generate outcome
    y = Phi * beta_true + randn(n) * noise_std

    return X, Phi, y, beta_true
end
```

2.3 Train/Validation/Test Split

```julia
"""
Split data into train/val/test
"""
function split_data(Phi, y; train_frac=0.6, val_frac=0.2)
    n = length(y)
    n_train = Int(floor(train_frac * n))
    n_val = Int(floor(val_frac * n))

    idx = Random.shuffle(1:n)
    train_idx = idx[1:n_train]
    val_idx = idx[n_train+1:n_train+n_val]
    test_idx = idx[n_train+n_val+1:end]

    return (
        train = (Phi[train_idx, :], y[train_idx]),
        val = (Phi[val_idx, :], y[val_idx]),
        test = (Phi[test_idx, :], y[test_idx]),
        idx = (train=train_idx, val=val_idx, test=test_idx)
    )
end
```

Key principle: The test set remains locked away until final evaluation!

2.4 Regression Functions

```julia
"""
Fit OLS regression
"""
function fit_ols(X::Matrix{Float64}, y::Vector{Float64})
    X_design = hcat(ones(size(X, 1)), X)
    beta = (X_design' * X_design) \ (X_design' * y)
    return beta
end

"""
Predict with OLS coefficients
"""
function predict_ols(X::Matrix{Float64}, beta::Vector{Float64})
    X_design = hcat(ones(size(X, 1)), X)
    return X_design * beta
end

"""
Fit Ridge regression (L2 penalty)
"""
function fit_ridge(X::Matrix{Float64}, y::Vector{Float64}, lambda::Float64)
    n, p = size(X)
    X_design = hcat(ones(n), X)
    # Don't penalize intercept
    penalty = diagm([0.0; fill(lambda, p)])
    beta = (X_design' * X_design + penalty) \ (X_design' * y)
    return beta
end

"""
Compute MSE
"""
function compute_mse(y_true::Vector{Float64}, y_pred::Vector{Float64})
    return mean((y_true .- y_pred).^2)
end
```

3 Data Generation and Preprocessing (10 minutes)

```julia
Random.seed!(1234)

# Parameters
n_total = 300
n_features = 2      # Raw features
n_basis = 120       # Basis functions (p)
n_active = 5       # True non-zero coefficients

println("Data Generation:")
println("  Total observations: ", n_total)
println("  Raw features: ", n_features)
println("  Basis functions (p): ", n_basis)
println("  True active coefficients: ", n_active)

# Generate data
X, Phi, y, beta_true = generate_data(n_total, n_features, n_basis,
                                      n_active, noise_std=23)

# Split data
data = split_data(Phi, y)
Phi_train, y_train = data.train
Phi_val, y_val = data.val
Phi_test, y_test = data.test

println("\nData Split:")
println("  Training: ", length(y_train), " obs")
println("  Validation: ", length(y_val), " obs")
println("  Test: ", length(y_test), " obs")
println("  p/n ratio: ", round(n_basis/length(y_train), digits=2))
```

3.1 Standardization

Critical for regularization methods:

```
# Standardize based on training set
mu = mean(Phi_train, dims=1)
sigma = std(Phi_train, dims=1)
Phi_train = (Phi_train .- mu) ./ sigma
Phi_val = (Phi_val .- mu) ./ sigma
Phi_test = (Phi_test .- mu) ./ sigma

# Standardize y
y_mean = mean(y_train)
y_std = std(y_train)
y_train = (y_train .- y_mean) ./ y_std
y_val = (y_val .- y_mean) ./ y_std
y_test = (y_test .- y_mean) ./ y_std
```

Discussion: Why standardize? Why use training set statistics?

4 Ordinary Least Squares (10 minutes)

Let's see what happens with OLS in a high-dimensional setting:

```
# Fit OLS
beta_ols = fit_ols(Phi_train, y_train)
```

```
# Compute MSE
mse_train_ols = compute_mse(y_train, predict_ols(Phi_train, beta_ols))
mse_val_ols = compute_mse(y_val, predict_ols(Phi_val, beta_ols))

println("\nOLS Results:")
println("  Training MSE: ", round(mse_train_ols, digits=4))
println("  Validation MSE: ", round(mse_val_ols, digits=4))
println("  Val/Train ratio: ", round(mse_val_ols/mse_train_ols, digits=2), "x")
```

Expected outcome: Severe overfitting! The validation MSE should be much larger than training MSE.

Task:

- What does the val/train ratio tell us?
- Why does OLS perform poorly here?
- What's the relationship between p, n, and overfitting?

5 Ridge Regression (L2 Regularization) (15 minutes)

Ridge adds a penalty on the sum of squared coefficients:

$$\min_{\beta} \sum_{i=1}^{n} (y_i - \phi_i' \beta)^2 + \lambda \sum_{j=1}^{p} \beta_j^2$$

5.1 Cross-Validation for λ

```
"""
Cross-validate to find optimal lambda for Ridge
"""
function cv_ridge(Phi_train, y_train, Phi_val, y_val, lambda_grid)
    mse_train = zeros(length(lambda_grid))
    mse_val = zeros(length(lambda_grid))

    for (i, lambda) in enumerate(lambda_grid)
        beta = fit_ridge(Phi_train, y_train, lambda)
        mse_train[i] = compute_mse(y_train, predict_ols(Phi_train, beta))
        mse_val[i] = compute_mse(y_val, predict_ols(Phi_val, beta))
    end

    best_idx = argmin(mse_val)
    return lambda_grid[best_idx], mse_train, mse_val, best_idx
end
```

```julia
# Run cross-validation
lambda_grid = exp.(range(log(1e-6), log(1000), length=500))
lambda_best_ridge, mse_train_ridge, mse_val_ridge, best_idx_ridge =
    cv_ridge(Phi_train, y_train, Phi_val, y_val, lambda_grid)

println("\nRidge Regression Results:")
println("  Best lambda: ", round(lambda_best_ridge, digits=4))
println("  Training MSE: ", round(mse_train_ridge[best_idx_ridge], digits=4))
println("  Validation MSE: ", round(mse_val_ridge[best_idx_ridge], digits=4))
println("  Val/Train ratio: ",
        round(mse_val_ridge[best_idx_ridge]/mse_train_ridge[best_idx_ridge],
              digits=2), "x")
println("  Improvement over OLS: ",
        round(100*(1 - mse_val_ridge[best_idx_ridge]/mse_val_ols), digits=1), "%")
```

Discussion:

- What happens as $\lambda \to 0$? As $\lambda \to \infty$?
- Why does Ridge help with overfitting?
- Limitation: Ridge doesn't set coefficients exactly to zero

6 LASSO Regression (L1 Regularization) (15 minutes)

LASSO adds a penalty on the sum of absolute values:

$$\min_{\beta} \sum_{i=1}^{n} (y_i - \phi_i'\beta)^2 + \lambda \sum_{j=1}^{p} |\beta_j|$$

Key advantage: LASSO performs **variable selection** by setting some coefficients exactly to zero.

6.1 Cross-Validation with Sparsity Tracking

```julia
"""
Cross-validate LASSO
"""
function cv_lasso(Phi_train, y_train, Phi_val, y_val, lambda_grid)
    mse_train = zeros(length(lambda_grid))
    mse_val = zeros(length(lambda_grid))
    n_nonzero = zeros(Int, length(lambda_grid))

    Phi_train_tbl = Tables.table(Phi_train)
    Phi_val_tbl = Tables.table(Phi_val)

    for (i, lambda) in enumerate(lambda_grid)
        lasso = LassoRegressor(lambda=lambda, solver=ProxGrad(max_iter=10_000))
        mach = machine(lasso, Phi_train_tbl, y_train)
        fit!(mach, verbosity=0)

        y_train_pred = MLJ.predict(mach, Phi_train_tbl)
        y_val_pred = MLJ.predict(mach, Phi_val_tbl)

        mse_train[i] = compute_mse(y_train, y_train_pred)
        mse_val[i] = compute_mse(y_val, y_val_pred)

        beta = fitted_params(mach).coefs
        n_nonzero[i] = sum(abs.(last.(beta)) .> 1e-6)
    end

    best_idx = argmin(mse_val)
    return lambda_grid[best_idx], mse_train, mse_val, n_nonzero, best_idx
end

# Run cross-validation
lambda_grid_lasso = exp.(range(log(1e-6), log(1000), length=500))
lambda_best_lasso, mse_train_lasso, mse_val_lasso, n_nonzero, best_idx_lasso =
    cv_lasso(Phi_train, y_train, Phi_val, y_val, lambda_grid_lasso)

println("\nLASSO Regression Results:")
println("  Best lambda: ", round(lambda_best_lasso, digits=4))
println("  Training MSE: ", round(mse_train_lasso[best_idx_lasso], digits=4))
println("  Validation MSE: ", round(mse_val_lasso[best_idx_lasso], digits=4))
println("  Selected features: ", n_nonzero[best_idx_lasso], " of ", n_basis)
println("  True active features: ", n_active)
println("  Improvement over OLS: ",
        round(100*(1 - mse_val_lasso[best_idx_lasso]/mse_val_ols), digits=1), "%")
```

Discussion:

- How many features did LASSO select vs. the true number (5)?
- Why is sparsity desirable for interpretation?
- When would you prefer LASSO over Ridge?

7 Visualization (15 minutes)

7.1 Ridge Regularization Path

```
# Plot 1: Ridge regularization path
p1 = plot(log.(lambda_grid), [mse_train_ridge mse_val_ridge],
     label=["Training MSE" "Validation MSE"],
     xlabel="log(lambda)", ylabel="MSE",
     title="Ridge: Bias-Variance Tradeoff",
     linewidth=2, legend=:topleft,
     size=(800, 500))
vline!(p1, [log(lambda_best_ridge)], label="Optimal lambda",
       linestyle=:dash, linewidth=2, color=:red)
savefig(p1, "ridge_regularization_path.png")
```

What to look for:

- Training MSE increases with λ (more bias)
- Validation MSE is U-shaped (bias-variance tradeoff)
- Optimal λ minimizes validation error

7.2 LASSO Sparsity Path

```
# Plot 2: LASSO sparsity path
p2 = plot(n_nonzero, [mse_train_lasso mse_val_lasso],
     label=["Training MSE" "Validation MSE"],
     xlabel="Number of Selected Features", ylabel="MSE",
     title="LASSO: Model Complexity vs Performance",
     linewidth=2, legend=:topright,
     size=(800, 500))
vline!(p2, [n_nonzero[best_idx_lasso]], label="Optimal",
       linestyle=:dash, linewidth=2, color=:red)
vline!(p2, [n_active], label="True # Active",
       linestyle=:dot, linewidth=2, color=:green)
savefig(p2, "lasso_sparsity_path.png")
```

7.3 Method Comparison

```
# Create results table
results = DataFrame(
    Method = ["OLS", "Ridge", "LASSO"],
    Train_MSE = [mse_train_ols,
                 mse_train_ridge[best_idx_ridge],
                 mse_train_lasso[best_idx_lasso]],
    Val_MSE = [mse_val_ols,
               mse_val_ridge[best_idx_ridge],
               mse_val_lasso[best_idx_lasso]]
)

# Plot 3: Method comparison
p3 = groupedbar([results.Train_MSE results.Val_MSE],
           bar_position=:dodge,
           label=["Train" "Validation"],
           xlabel="Method", ylabel="MSE",
           title="Performance Comparison Across Data Splits",
           xticks=(1:3, results.Method),
           legend=:topright,
           size=(800, 500))
savefig(p3, "method_comparison.png")
```

8 Final Test Set Evaluation (10 minutes)

Only now do we evaluate on the test set!

```
# Refit with best hyperparameters
beta_ridge_final = fit_ridge(Phi_train, y_train, lambda_best_ridge)

lasso_final = LassoRegressor(lambda=lambda_best_lasso)
mach_final = machine(lasso_final, Tables.table(Phi_train), y_train)
fit!(mach_final, verbosity=0)

# Test predictions
mse_test_ols = compute_mse(y_test, predict_ols(Phi_test, beta_ols))
mse_test_ridge = compute_mse(y_test, predict_ols(Phi_test, beta_ridge_final))
mse_test_lasso = compute_mse(y_test,
                              MLJ.predict(mach_final, Tables.table(Phi_test)))

println("\n" * "="^50)
println("FINAL TEST SET RESULTS")
println("="^50)
println("OLS Test MSE:    ", round(mse_test_ols, digits=4))
println("Ridge Test MSE:  ", round(mse_test_ridge, digits=4))
println("LASSO Test MSE:  ", round(mse_test_lasso, digits=4))

# Add to results table
results.Test_MSE = [mse_test_ols, mse_test_ridge, mse_test_lasso]
results.Overfit_Ratio = [mse_val_ols/mse_train_ols,
                         mse_val_ridge[best_idx_ridge]/mse_train_ridge[best_idx_ridge],
                                mse_val_lasso[best_idx_lasso]/mse_train_lasso[best_idx_lasso]]

println("\n", results)
```

Critical question: Does test MSE match validation MSE? If not, what went wrong?

9 Key Takeaways (10 minutes)

9.1 Main Lessons

1. **High-Dimensional Problems ($p \approx n$):**
   - OLS fails catastrophically when p is close to n
   - Training error becomes misleading (can be arbitrarily small)
   - Regularization is not optional—it's essential
2. **Bias-Variance Tradeoff:**
   - No regularization ($\lambda = 0$): High variance, overfits training noise
   - Too much regularization ($\lambda \to \infty$): High bias, underfits signal
   - Optimal λ: Balances bias and variance for best generalization
3. **Train/Validation/Test Split:**
   - Training: Estimate model parameters (β)
   - Validation: Choose hyperparameters (λ)
   - Test: Assess final performance (never touch until end!)
4. **Regularization Methods:**
   - Ridge (L2): Shrinks all coefficients, works well with correlated features
   - LASSO (L1): Sparse selection, sets many coefficients to exactly zero
   - Choice depends on whether you believe in sparsity
5. **Sparsity and Interpretability:**
   - True model: Only 5 of 120 basis terms are active
   - LASSO automatically discovers relevant features
   - Sparse models are more interpretable and computationally efficient

9.2 Connection to Econometrics

When do economists face high-dimensional problems?

- Many control variables relative to sample size
- Flexible functional forms (polynomials, splines, interactions)
- Fixed effects with many categories
- Instrumental variables with many instruments (weak IV problem)
- Time series with many lags

9.3 Extensions to Explore

- **Elastic Net:** Combine L1 and L2 penalties: $\lambda_1|\beta|+\lambda_2\beta^2$
- ***k*-Fold Cross-Validation:** Use all data more efficiently for hyperparameter tuning
- **Different Basis Functions:** Try Chebyshev, B-splines, wavelets
- **Classification:** Binary outcomes with regularized logistic regression
- **Tree-Based Methods:** Random Forest, XGBoost for comparison
- **Neural Networks:** Deep learning as ultimate flexible basis expansion

9.4 Implementation Tips

- Always standardize features before applying regularization
- Use a fine grid for λ (logarithmic spacing works well)
- Monitor both training and validation error to diagnose bias vs. variance
- For LASSO, track sparsity to understand model selection
- Never tune hyperparameters using test set performance
- In practice, use k-fold CV instead of single validation set

9.5 Theoretical Foundations

For those interested in the theory:

- Ridge solution: $\hat{\beta}_{Ridge} = (X'X+\lambda I)^{-1}X'y$
- LASSO has no closed form, requires iterative algorithms (coordinate descent, LARS)
- Oracle inequality: Regularization achieves optimal prediction rate under sparsity