

Introduction to Regression

From OLS to the ML Perspective on Regression

Assoc. Prof. Dr. Onur Polat

Hacettepe University, Institute of Informatics
Fall 2026–2027

Week 4 Outline

From Classical Econometrics to the ML Regression Framework

01 OLS Review

Classical foundations

02 Regression as Supervised Learning

ML perspective on prediction

03 Loss Functions

MSE, MAE, Huber and beyond

04 Gradient Descent

Optimization for large data

05 Polynomial Regression

Non-linearity and overfitting

06 Cross-Validation

Honest model evaluation

07 Multiple Regression in ML

Feature engineering pipeline

08 Regularization Preview

Ridge, Lasso and Elastic Net

01

OLS Review

Classical Foundations for Regression Analysis

OLS: Core Concepts

The workhorse of econometrics

$$\mathbf{y} = \mathbf{X}\boldsymbol{\beta} + \boldsymbol{\varepsilon} \rightarrow \boldsymbol{\beta} = (\mathbf{X}^T\mathbf{X})^{-1}\mathbf{X}^T\mathbf{y}$$

Minimizes sum of squared residuals: $\sum (y_i - \hat{y}_i)^2$

- **Linearity assumption:** $E(\mathbf{y}|\mathbf{X}) = \mathbf{X}\boldsymbol{\beta}$
- Unbiased: $E(\boldsymbol{\beta}) = \boldsymbol{\beta}$ under Gauss–Markov conditions
- BLUE: Best Linear Unbiased Estimator when $E(\boldsymbol{\varepsilon}\boldsymbol{\varepsilon}^T) = \sigma^2\mathbf{I}$
- Closed-form solution — no iteration needed
- Standard errors, t-tests, F-tests all follow from normality of $\boldsymbol{\varepsilon}$

OLS: Assumptions and Diagnostics

Gauss–Markov Assumptions

A1. Linearity

$y = X\beta + \varepsilon$ — linear in parameters

A2. Full Rank

X has rank k (no perfect multicollinearity)

A3. Exogeneity

$E(\varepsilon | X) = 0$ — no omitted variable bias

A4. Homoscedasticity

$\text{Var}(\varepsilon | X) = \sigma^2 I$ — constant variance

A5. No autocorrelation

$\text{Cov}(\varepsilon_i, \varepsilon_j) = 0$ for $i \neq j$

Diagnostic Tools

- ▶ Residual plots → linearity
- ▶ Breusch–Pagan → heteroscedasticity
- ▶ Durbin–Watson → autocorrelation
- ▶ VIF > 10 → multicollinearity
- ▶ Jarque–Bera → normality
- ▶ RESET test → functional form
- ▶ Adjusted R^2 → goodness of fit

OLS in Python

statsmodels and scikit-learn side by side

Python: Two Paradigms

```
import statsmodels.api as sm
from sklearn.linear_model import LinearRegression

# --- statsmodels (inference) ---
X_sm = sm.add_constant(X)
model = sm.OLS(y, X_sm).fit()
print(model.summary()) # t-stats, p-values, CI

# --- scikit-learn (prediction) ---
lr = LinearRegression()
lr.fit(X_train, y_train)
y_pred = lr.predict(X_test)
print(f"R² = {lr.score(X_test, y_test):.4f}")
```

! statsmodels for inference (p-values, CIs) | scikit-learn for prediction (R², MSE)

OLS Limitations for ML

Where classical econometrics meets its boundaries

$$O(n \cdot k^2)$$

Matrix inversion
scaling cost

$$p > n$$

High-dimensional
failure

No Penalty

Overfits with
many features

- Computational cost: $(X^T X)^{-1}$ becomes infeasible for very large datasets
- Cannot handle $p > n$ (more features than observations)
- No built-in regularization $\rightarrow$ overfits when k is large relative to n
- Assumes linear functional form — misses non-linear patterns
- Prediction accuracy is secondary to unbiasedness in econometric tradition

02

Regression as Supervised Learning

Shifting from Explanation to Prediction

The Supervised Learning Framework

Training a model to predict unseen outcomes

$$\mathbf{f} = \operatorname{argmin} E[L(y, f(\mathbf{x}))]$$

Find the function f that minimizes expected loss over all (x, y) pairs

- **Goal: learn a mapping $f: X \rightarrow Y$ from labeled training data**
- Regression: y is continuous (e.g. GDP, price, wage)
- Training set $\{(x_i, y_i)\}_n$ used to estimate $\hat{f}$, then evaluated on unseen data
- Key difference from econometrics: focus on prediction accuracy, not β interpretation
- Bias–variance tradeoff replaces the unbiasedness criterion

Econometrics vs. Machine Learning

Two paradigms for the same regression equation

Aspect	Econometrics	Machine Learning
Goal	Estimate β (causal effect)	Minimize prediction error
Loss metric	Unbiasedness, efficiency	MSE, MAE on test set
Model selection	Theory-driven	Data-driven (CV, AIC)
Overfitting	Adjusted R^2 , parsimony	Train/test split, regularization
Interpretation	Central	Secondary
Sample size	Moderate (100–5,000)	Large (10,000–millions)
Feature engineering	Minimal	Extensive

→ ML does not replace econometrics — it complements it for prediction tasks

The Bias–Variance Tradeoff

The fundamental tension in predictive modeling

$$E[(y - f(x))^2] = \text{Bias}^2(f) + \text{Var}(f) + \sigma^2_e$$

Total Error = Systematic Error + Model Instability + Irreducible Noise



High Bias

Underfitting: model too simple,
misses patterns



High Variance

Overfitting: model too complex,
memorizes noise



Sweet Spot

Balance: just enough complexity to
generalize

Train/Test Split

The first step toward honest evaluation

Training Set (80%)

Test (20%)

Python: train_test_split

```
from sklearn.model_selection import train_test_split

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

model.fit(X_train, y_train)      # learn from training data
score = model.score(X_test, y_test)  # evaluate on unseen data
```

03

Loss Functions

Measuring How Wrong Our Predictions Are

Common Loss Functions

MSE, RMSE, and MAE for regression

Loss Functions for Regression

$$MSE = (1/n) \sum (y_i - \hat{y}_i)^2$$

Mean Squared Error — penalizes large errors quadratically, differentiable everywhere

$$RMSE = \sqrt{MSE}$$

Root MSE — same units as y, interpretable, standard reporting metric

$$MAE = (1/n) \sum |y_i - \hat{y}_i|$$

Mean Absolute Error — robust to outliers, non-differentiable at zero

$$R^2 = 1 - SS_{res}/SS_{tot}$$

Coefficient of determination — proportion of variance explained (0 to 1)



MSE/RMSE preferred for optimization (smooth gradient) | MAE for outlier-robust applications

Advanced Loss Functions

Huber loss and when to choose which

Huber Loss

$$L^{\delta}(y, f(x)) =$$

$$\begin{aligned} & \frac{1}{2}(y - f(x))^2 && \text{if } |y - f(x)| \leq \delta \\ & \delta |y - f(x)| - \frac{1}{2}\delta^2 && \text{otherwise} \end{aligned}$$

*Quadratic near zero, linear for large errors.
Hyperparameter δ controls the transition.*

Best of both worlds: smooth gradient + outlier robustness

Choosing a Loss Function

Clean data, no outliers?

MSE / RMSE

Outliers in y ?

MAE or Huber

Business cost asymmetric?

Custom loss

Need interpretable scale?

RMSE (same units as y)

Comparing models?

R^2 + RMSE together

Computing Loss in Python

sklearn.metrics for regression evaluation

Python: Evaluation Metrics

```
from sklearn.metrics import (mean_squared_error,
                             mean_absolute_error, r2_score)
import numpy as np

y_pred = model.predict(X_test)

mse = mean_squared_error(y_test, y_pred)
rmse = np.sqrt(mse)
mae = mean_absolute_error(y_test, y_pred)
r2 = r2_score(y_test, y_pred)

print(f"MSE: {mse:.4f}")
print(f"RMSE: {rmse:.4f}")
print(f"MAE: {mae:.4f}")
print(f"R2: {r2:.4f}")
```

04

Gradient Descent

Iterative Optimization for Large-Scale Regression

Gradient Descent: Intuition

Walking downhill on the loss surface

$$\theta_{n+1} = \theta_n - \alpha \cdot \nabla J(\theta_n)$$

Update = Current – Learning Rate × Gradient

- **Start with random parameters θ_0**
- Compute gradient $\nabla J(\theta)$: direction of steepest increase
- Move in the opposite direction (downhill) by step size α
- Repeat until convergence: $|\nabla J(\theta)| < \epsilon$ or max iterations
- **Key hyperparameter: learning rate α**

Learning Rate: Too Small vs. Too Large

$$\alpha = 0.001$$

Too small
Slow convergence

$$\alpha = 0.01$$

Just right
Smooth convergence

$$\alpha = 1.0$$

Too large
Diverges / oscillates

- Small α : takes many iterations, may get stuck in local minima
- Large α : overshoots minimum, loss oscillates or diverges
- Adaptive methods (Adam, RMSProp) adjust α per parameter automatically
- For convex problems (OLS): any $\alpha < 2/\lambda_{\max}$ guarantees convergence

Gradient Descent Variants

Batch, Stochastic, and Mini-Batch

Variant	Batch Size	Per-Step Cost	Convergence	When to Use
Batch GD	All n	$O(n \cdot k)$	Smooth, slow	Small data ($n < 10k$)
Stochastic (SGD)	1	$O(k)$	Noisy, fast	Online learning
Mini-Batch	32–256	$O(b \cdot k)$	Balanced	Standard for most ML

- **Mini-Batch GD is the industry standard — good GPU utilization + stable training**
- SGD with momentum helps escape shallow local minima
- Adam optimizer: adaptive α + momentum — default in most deep learning frameworks

GD vs. Normal Equations

When to iterate vs. when to solve directly

Normal Equations

$$\beta = (X^T X)^{-1} X^T y$$

- ✓ Exact solution (no hyperparameters)
- ✓ One-step computation
- ✗ $O(k^3)$ — slow when $k > 10,000$
- ✗ Requires invertible $X^T X$

Gradient Descent

$$\theta \leftarrow \theta - \alpha \nabla J(\theta)$$

- ✓ Scales to millions of samples
- ✓ Works with any differentiable loss
- ✗ Requires tuning α , epochs
- ✗ May converge to local minimum

Rule of thumb: $n < 10,000$ and $k < 1,000 \rightarrow$ Normal Equations | Otherwise $\rightarrow$ GD

Gradient Descent from Scratch

A simple Python implementation for linear regression

Python: Manual GD

```
def gradient_descent(X, y, lr=0.01, epochs=1000):
    n, k = X.shape
    theta = np.zeros(k)

    for epoch in range(epochs):
        y_pred = X @ theta
        gradient = (2/n) * X.T @ (y_pred - y)
        theta -= lr * gradient

        if epoch % 200 == 0:
            loss = np.mean((y_pred - y)**2)
            print(f"Epoch {epoch}: MSE = {loss:.4f}")

    return theta

# Usage
X_b = np.c_[np.ones(n), X] # add bias column
theta = gradient_descent(X_b, y, lr=0.01)
```

05

Polynomial Regression & Overfitting

Non-Linearity, Model Complexity, and the Danger of Overfitting

Polynomial Regression

Capturing non-linear relationships with linear algebra

$$y = \beta_0 + \beta_1 x + \beta_2 x^2 + \beta_3 x^3 + \dots + \beta_n x^n + \varepsilon$$

Still linear in parameters — OLS can solve this!

- **Key insight: polynomial regression is linear regression with engineered features**
- Transform $x \rightarrow [x, x^2, x^3, \dots]$ then fit ordinary linear regression
- Higher degree $\rightarrow$ more flexibility $\rightarrow$ risk of overfitting
- Degree 1 = linear, Degree 2 = quadratic, Degree 3 = cubic
- Economic examples: Kuznets curve (inequality vs. income), Phillips curve

Underfitting vs. Overfitting

Polynomial degree controls model complexity

Degree 1: Underfitting

Too simple — misses curvature in the data

Train $R^2 = 0.45$
Test $R^2 = 0.44$

Degree 3: Good Fit

Captures the true relationship well

Train $R^2 = 0.92$
Test $R^2 = 0.89$

Degree 15: Overfitting

Memorizes noise — fails on new data

Train $R^2 = 0.99$
Test $R^2 = 0.35$

The gap between Train R^2 and Test R^2 is the overfitting signal

Learning Curves

Diagnosing underfitting and overfitting with training size

- **Learning curve: plot training & validation error vs. training set size**
- High bias (underfitting): both errors converge at a HIGH value
- High variance (overfitting): large gap between training and validation error
- Adding more data helps overfitting but NOT underfitting
- Fixing underfitting: add features, increase degree, use a more complex model
- Fixing overfitting: more data, reduce features, regularize, cross-validate

Python: Learning Curves

```
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="neg_mean_squared_error"
```

Polynomial Regression in Python

Using sklearn Pipeline for clean code

Python: Polynomial Pipeline

```
from sklearn.preprocessing import PolynomialFeatures
from sklearn.pipeline import make_pipeline
from sklearn.linear_model import LinearRegression

# Compare degrees 1, 3, 10
for degree in [1, 3, 10]:
    model = make_pipeline(
        PolynomialFeatures(degree, include_bias=False),
        LinearRegression()
    )
    model.fit(X_train, y_train)

    train_r2 = model.score(X_train, y_train)
    test_r2 = model.score(X_test, y_test)
    print(f"Degree {degree:2d}: "
          f"Train R²={train_r2:.3f}, "
          f"Test R²={test_r2:.3f}")
```

06

Cross-Validation

Honest Model Evaluation and Selection

K-Fold Cross-Validation

Every observation gets to be in the test set exactly once



- Split data into K equal folds (typically K = 5 or 10)
- Train on K-1 folds, test on remaining fold — repeat K times
- Final score = average across all K test folds $\pm$ std
- Uses all data for both training and testing — no wasted samples

Cross-Validation in Python

Model selection with cross_val_score

Python: CV for Model Selection

```
from sklearn.model_selection import cross_val_score
from sklearn.preprocessing import PolynomialFeatures
from sklearn.pipeline import make_pipeline

# Compare polynomial degrees using 5-fold CV
for degree in [1, 2, 3, 5, 10]:
    model = make_pipeline(
        PolynomialFeatures(degree, include_bias=False),
        LinearRegression()
    )
    scores = cross_val_score(
        model, X, y, cv=5,
        scoring="neg_mean_squared_error"
    )
    rmse = np.sqrt(-scores.mean())
    std = np.sqrt(-scores).std()
    print(f"Degree {degree:2d}: "
          f"CV RMSE = {rmse:.4f} ± {std:.4f}")
```

07

Multiple Regression in ML

Feature Engineering, Scaling, and the Full Pipeline

The ML Regression Pipeline

From raw data to predictions



- **Feature engineering: create new variables from existing data**
- Log transforms for skewed variables (income, house prices)
- Interaction terms: $x_1 \cdot x_2$ captures joint effects
- Dummy encoding for categorical variables (one-hot encoding)
- Domain knowledge from economics guides feature creation

Feature Scaling

Why scale matters for gradient-based and distance-based models

Scaling Methods

StandardScaler: $z = (x - \mu) / \sigma$

Zero mean, unit variance. Best for normally distributed features. Required for regularization.

MinMaxScaler: $x' = (x - \min) / (\max - \min)$

Scales to [0, 1]. Preserves zeros. Sensitive to outliers.

RobustScaler: $x' = (x - \text{median}) / \text{IQR}$

Uses median and IQR. Robust to outliers. Good for skewed data.



Always fit the scaler on training data only, then transform both train and test

Data leakage = fitting scaler on full data before splitting → overly optimistic scores

Feature Importance in Linear Models

Which variables drive predictions?

- **After scaling, coefficient magnitudes indicate feature importance**
- $|\beta_j|$ ranks variables by their contribution to predictions
- Warning: only meaningful after standardization (z-scoring)
- Complements p-values from econometrics with a predictive perspective

Python: Feature Importance

```
from sklearn.preprocessing import StandardScaler
from sklearn.linear_model import LinearRegression

scaler = StandardScaler()
X_scaled = scaler.fit_transform(X_train)

lr = LinearRegression().fit(X_scaled, y_train)

# Feature importance (standardized coefficients)
importance = pd.Series(
```

Complete sklearn Pipeline

Putting it all together with Pipeline + GridSearchCV

**Python: Full Pipeline +
GridSearchCV**

```
from sklearn.pipeline import Pipeline
from sklearn.preprocessing import StandardScaler, PolynomialFeatures
from sklearn.linear_model import LinearRegression
from sklearn.model_selection import GridSearchCV
```

```
pipe = Pipeline([
    ("poly", PolynomialFeatures(include_bias=False)),
    ("scaler", StandardScaler()),
    ("reg", LinearRegression())
])
```

```
param_grid = {"poly__degree": [1, 2, 3, 4, 5]}
grid = GridSearchCV(pipe, param_grid, cv=5,
                    scoring="neg_mean_squared_error")
grid.fit(X_train, y_train)
```

```
print(f"Best degree: {grid.best_params_}")
print(f"Test R2: {grid.score(X_test, y_test):.4f}")
```

08

Regularization Preview

Controlling Complexity: Ridge, Lasso, and Elastic Net

Why Regularize?

Adding a penalty term to prevent overfitting

$$J(\beta) = \sum (y_i - \hat{y}_i)^2 + \lambda \cdot \text{Penalty}(\beta)$$

Loss = Data Fit + Complexity Penalty

- **λ (lambda) controls the strength of regularization**
- $\lambda = 0$: no penalty $\rightarrow$ ordinary OLS (risk of overfitting)
- $\lambda \rightarrow \infty$: all coefficients shrunk to zero (underfitting)
- Cross-validation selects the optimal λ
- Regularization introduces bias to reduce variance $\rightarrow$ better generalization
- Required when features > observations or features are correlated

Three Regularization Methods

Method	Penalty	Effect	Best For
Ridge (L2)	$\lambda \sum \beta_j^2$	Shrinks all β toward zero but none exactly to zero	Multicollinearity, all features relevant
Lasso (L1)	$\lambda \sum \beta_j $	Shrinks some β exactly to zero (feature selection)	Sparse models, feature selection
Elastic Net	$\lambda_1 \sum \beta_j + \lambda_2 \sum \beta_j^2$	Combines L1 and L2: shrinkage + selection	Correlated features, best of both worlds

We will dive deep into regularization in Week 5 — this is the preview!

Quick Python Preview

Ridge, Lasso, ElasticNet in scikit-learn

Python: Regularized Regression

```
from sklearn.linear_model import Ridge, Lasso, ElasticNet

# Ridge (L2)
ridge = Ridge(alpha=1.0)
ridge.fit(X_train_scaled, y_train)

# Lasso (L1)
lasso = Lasso(alpha=0.1)
lasso.fit(X_train_scaled, y_train)
print(f"Non-zero coefficients: "
      f"{np.sum(lasso.coef_ != 0)} / {len(lasso.coef_)}")

# Elastic Net (L1 + L2)
enet = ElasticNet(alpha=0.1, l1_ratio=0.5)
enet.fit(X_train_scaled, y_train)

# Compare on test set
for name, m in [("Ridge", ridge), ("Lasso", lasso),
               ("ElasticNet", enet)]:
    r2 = m.score(X_test_scaled, y_test)
```

Week 4 Summary

Key takeaways from OLS to the ML Regression Framework

01

OLS Review

Classical foundation:
BLUE under Gauss–Markov

02

Supervised Learning

Shift from β interpretation
to prediction accuracy

03

Loss Functions

MSE, MAE, Huber:
choose by data quality

04

Gradient Descent

Iterative optimization
scales to big data

05

Polynomial Regression

Non-linearity via
feature engineering

06

Cross-Validation

Honest evaluation:
K-fold CV for selection

07

ML Pipeline

Scale $\rightarrow$ engineer $\rightarrow$
train $\rightarrow$ evaluate

08

Regularization

Ridge, Lasso, ElasticNet:
coming in Week 5

$\rightarrow$ Next Week: Regularization in Depth — Ridge, Lasso, Elastic Net, and Model Selection