IMEN574/IDSC574 Programming for Data Science

9. **Regression**

Spring 2026

Instructor: **Sangwoo Mo**

Today’s Agenda

01

What Is Regression?

Definition, examples, and where it fits in ML

02

Linear Regression

OLS, gradient descent, closed-form,

03

Regularization

Ridge (L2), Lasso (L1) and why they matter

04

Kernel Regression

Going non-linear with the kernel trick

05

Tree-Based & Boosting

Decision Tree, Random Forest, XGBoost, LightGBM

06

Evaluation & Practice

Metrics, CV, full example, and best practices

What Is Regression?

Given input features X, predict a continuous numerical output y.

$$y = f(X) + \varepsilon$$

X = input features (square footage, # bedrooms, ...)

y = continuous target (house price, temperature, ...)

f = the mapping function we want to learn from data

ε = irreducible noise

Classification → discrete categories (spam / not spam) | Regression → continuous values ($350K, 22.5°C)

Finding the Best f in Function Space

Regression is fundamentally an optimization problem in a function space F.

$$f^* = \underset{f \in F}{\operatorname{argmin}} \; E[\, L(\, y,\, f(X)\,)\,]$$

Different methods define F differently:

Kernel Methods
F = RKHS defined by kernel K

Gaussian Processes
F = distribution over functions

Decision Trees
F = piecewise constant functions

All regression methods are searching for the best f — they just define the search space F differently. Every model you learn in this course is a different way to solve this optimization.

Parametric Models: From f* to θ*

When f is parameterized by θ (i.e., $f(X) = f_\theta(X)$), searching over F becomes searching over θ:

$$\theta^* = \text{argmin } E[\, L(\, y, f_\theta(X)\,)\,]$$

↓

Linear Regression: $f(X) = \beta_0 + \beta_1 x_1 + \ldots + \beta_n x_n$, so $\theta = \beta$

$$\beta^* = \text{argmin } MSE(\beta)$$

This is why the next section focuses on finding β* — via closed-form (Normal Eq.) or gradient descent.

Regression Is Everywhere

Housing Prices

Predict sale price from area, location, features

Finance

Forecast stock returns, risk, or revenue

Healthcare

Estimate dosage, recovery time, or lab values

Advertising

Predict sales lift from ad spend allocation

Energy

Forecast electricity demand or solar output

Retail

Predict demand, pricing or customer LTV

Linear Regression

The foundation of regression modeling

Linear Regression — Core Idea

Model output as a weighted sum of input features:

$$\hat{y} = \beta_0 + \beta_1 x_1 + \beta_2 x_2 + \ldots + \beta_n x_n$$

- β_0 = intercept (bias term)
- $\beta_1 \ldots \beta_n$ = coefficients (weights) — how much each feature matters
- Learning = finding the β values that best fit the training data

How do we define “best fit”?

Minimize the Mean Squared Error (MSE) — the average of squared differences:

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

Squaring penalizes large errors more heavily, and makes the math nice (differentiable, convex).

Gradient Descent — How Models Learn

The most general optimization algorithm in ML. Works for any differentiable loss function.

Idea: Iteratively adjust β in the direction that reduces the loss.

1. Start with random (or zero) weights β
2. Compute the gradient: ∇L = how much loss changes with each β
3. Update: $\beta \leftarrow \beta - \eta \cdot \nabla L$ (η = learning rate)
4. Repeat until convergence (gradient ≈ 0)

Learning Rate η

Too small
Converges but extremely slow

Just right
Smooth convergence to minimum

Too large
Oscillates or diverges!

Gradient Descent — Update Rule

For linear regression with MSE loss:

$$\beta_j \leftarrow \beta_j - \eta \cdot (2/n) \sum (y_i - \hat{y}_i) \cdot (-x_{ij})$$

Gradient Descent Variants:

Batch GD

Uses all N samples per step.
Exact gradient. Slow on big data.

Stochastic GD (SGD)

Uses 1 random sample per step.
Noisy but very fast.

Mini-Batch GD

Uses a batch of B samples.
Best of both worlds. Most common.

In practice, most frameworks (PyTorch, TensorFlow) use mini-batch SGD by default.

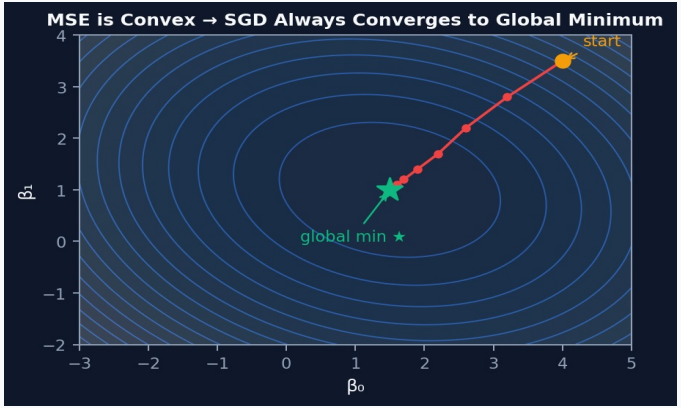
💡 Intuition: Each β_j is adjusted by how much its feature x_j contributed to the prediction error.

MSE is Convex — Why This Matters

MSE is convex → only one minimum!

Whether you use GD or the formula below,
you always reach the same global optimum.
No local minima to worry about.

→ So SGD and closed-form give the same β !



This is special to linear regression with MSE loss. Other losses (like cross-entropy) can have non-convex landscapes.

Convex (MSE)

- One global minimum
- GD always converges
- Closed-form exists

Non-Convex (e.g., Neural Nets)

- Multiple local minima
- GD can get stuck
- No closed-form solution

Closed-Form Solution (Normal Equation)

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

Gradient Descent vs. Closed-Form:

	Gradient Descent	Normal Equation
How	Iterative optimization	One matrix computation
Large p (features)	Works well	Slow — $O(p^3)$ matrix inverse
Other losses	Any differentiable loss	Only MSE

sklearn LinearRegression = closed-form | SGDRegressor = gradient descent | Both converge to the same β

When to Use Which?

- **p < 10,000 features** → LinearRegression (closed-form) is instant
- **p > 100,000 or streaming data** → SGDRegressor scales to millions of rows
- **Non-MSE loss (Huber, ε-insensitive)** → Only gradient descent supports these

Linear Regression — Code

```
from sklearn.linear_model import LinearRegression
from sklearn.model_selection import train_test_split
from sklearn.metrics import mean_squared_error, r2_score
import numpy as np
X_train, X_test, y_train, y_test = train_test_split(
    X, y, test_size=0.2, random_state=42)
model = LinearRegression()        # uses closed-form (Normal Eq)
model.fit(X_train, y_train)
y_pred = model.predict(X_test)
rmse = np.sqrt(mean_squared_error(y_test, y_pred))
r2 = r2_score(y_test, y_pred)
print(f"RMSE: {rmse:.3f},  R\u00B2: {r2:.3f}")
print(f"Coefficients: {model.coef_}")
print(f"Intercept:    {model.intercept_:.3f}")
```

Gradient Descent Version — SGDRegressor

When dataset is too large for closed-form, or you want a different loss:

```
from sklearn.linear_model import SGDRegressor
from sklearn.preprocessing import StandardScaler
from sklearn.pipeline import make_pipeline
# SGD requires feature scaling!
sgd_model = make_pipeline(
    StandardScaler(),
    SGDRegressor(loss='squared_error',   # = MSE
                 learning_rate='invscaling',
                 eta0=0.01,              # initial learning rate
                 max_iter=1000,
                 random_state=42)
)
sgd_model.fit(X_train, y_train)
print(f"R\u00B2: {sgd_model.score(X_test, y_test):.3f}")
```

LinearRegression (closed-form) is preferred for most tabular problems. SGD is for very large datasets or custom loss functions.

Interpreting Coefficients

Each coefficient β_j tells you:

"When feature x_j increases by 1 unit (holding others constant), the predicted y changes by β_j."

- Positive β → feature increases prediction
- Negative β → feature decreases prediction
- Larger |β| → stronger effect (but depends on feature scale!)

```
# Example: California Housing
# After fitting, inspect what the model learned
import pandas as pd
coef_df = pd.DataFrame({
    'feature': data.feature_names,
    'coefficient': model.coef_
}).sort_values('coefficient', key=abs, ascending=False)
print(coef_df.to_string(index=False))
# Tip: scale features first for comparable coefficients
from sklearn.preprocessing import StandardScaler
scaler = StandardScaler()
X_scaled = scaler.fit_transform(X_train)
```

Linear Regression — Limitations

Linearity

Assumes a linear relationship between X and y. Non-linear patterns? Need kernel methods or tree-based models.

Multicollinearity

Highly correlated features → unstable, unreliable coefficients. Solution: regularization or feature selection.

Outlier Sensitivity

Squared error amplifies outlier influence. A single extreme point can skew the entire model.

Overfitting (many features)

When p (features) is large relative to n (samples), the model memorizes noise. Solution: regularization.

Most of these limitations are addressed by regularization — our next topic!

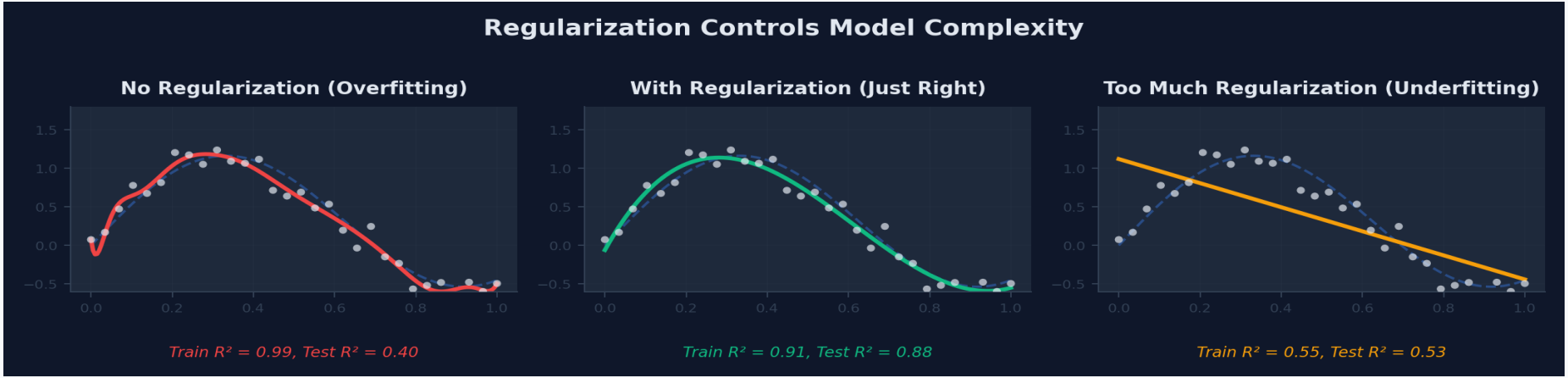
Regularization

Controlling model complexity to prevent overfitting

Why Regularization?

Problem: With many features, linear regression can memorize noise instead of learning patterns.

Solution: Loss = MSE + α · Penalty(β) — penalizes large coefficients to prevent overfitting.



α (alpha) controls regularization strength:

α = 0	No penalty → standard LR (overfit risk)
α optimal	Balanced bias-variance → best generalization
α too large	All β → 0 → underfitting

Ridge Regression (L2 Penalty)

$$\text{Loss} = \text{MSE} + \alpha \sum \beta_j^2$$

- Adds the sum of squared coefficients as penalty
- Shrinks all coefficients toward zero, but never exactly zero
- All features stay in the model (no feature selection)
- Handles multicollinearity well — distributes weight among correlated features

```
from sklearn.linear_model import Ridge, LinearRegression
from sklearn.preprocessing import StandardScaler
from sklearn.pipeline import make_pipeline
import numpy as np
# Ridge requires scaled features for fair penalization
ridge = make_pipeline(StandardScaler(), Ridge(alpha=1.0))
ridge.fit(X_train, y_train)
print(f"Ridge R\u00B2: {ridge.score(X_test, y_test):.3f}")
# Compare coefficient sizes: Ridge vs plain LR
lr = LinearRegression().fit(X_train, y_train)
print(f"LR   coef norm: {np.linalg.norm(lr.coef_):.2f}")
print(f"Ridge coef norm: {np.linalg.norm(ridge[1].coef_):.2f}")
```

Why penalize large β? Large coefficients = model over-relies on specific features = unstable on new data.

Ridge — Effect of α

As α increases, coefficients shrink smoothly toward zero:

```
from sklearn.linear_model import Ridge
from sklearn.preprocessing import StandardScaler
from sklearn.model_selection import cross_val_score
import numpy as np
scaler = StandardScaler()
X_scaled = scaler.fit_transform(X)
alphas = [0.001, 0.01, 0.1, 1, 10, 100, 1000]
for a in alphas:
    ridge = Ridge(alpha=a)
    scores = cross_val_score(ridge, X_scaled, y, cv=5, scoring='r2')
    coef_norm = np.linalg.norm(
        Ridge(alpha=a).fit(X_scaled, y).coef_)
    print(f"alpha={a:7.3f}  R\u00B2={scores.mean():.3f}  "
          f"|\u03B2|={coef_norm:.2f}")
# Output shows: as alpha increases,
#   - |beta| decreases (coefficients shrink)
#   - R\u00B2 first improves then degrades (sweet spot)
```

Lasso Regression (L1 Penalty)

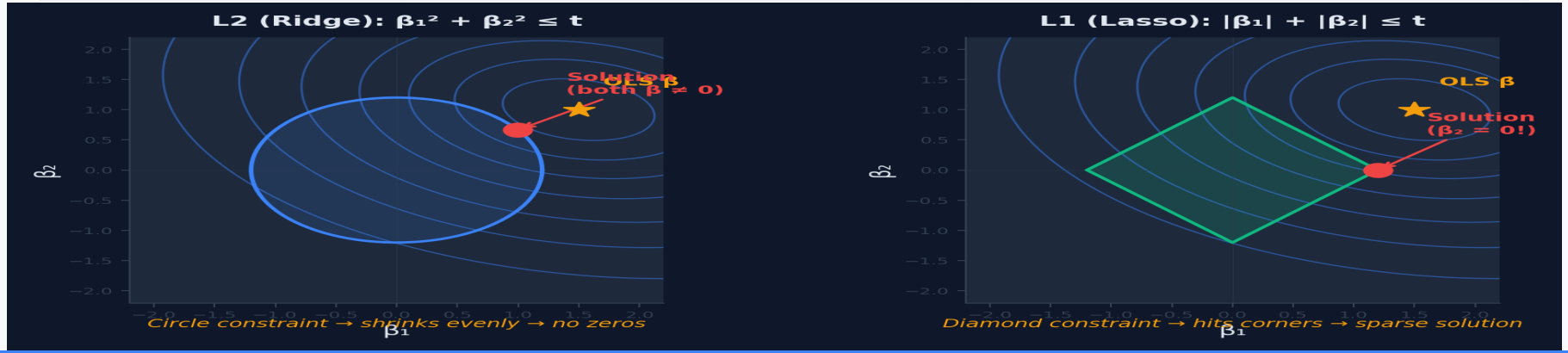
$$\text{Loss} = \text{MSE} + \alpha \sum |\beta_j|$$

- Adds the sum of absolute values of coefficients
- **Can shrink coefficients to exactly zero → automatic feature selection!**
- Great when you suspect only a few features truly matter
- Produces sparse models (fewer non-zero coefficients)

```
from sklearn.linear_model import Lasso
lasso = make_pipeline(StandardScaler(), Lasso(alpha=0.1))
lasso.fit(X_train, y_train)
print(f"Lasso R\u00B2: {lasso.score(X_test, y_test):.3f}")
# Feature selection: which coefficients survived?
coefs = lasso[1].coef_
n_features = len(coefs)
n_used = (coefs != 0).sum()
print(f"Features used: {n_used} / {n_features}")
for name, c in zip(feature_names, coefs):
    if c != 0:
        print(f"  {name}: {c:.4f}")
```

Ridge vs. Lasso — When to Use Which?

	Ridge (L2)	Lasso (L1)
Penalty	$\sum\beta^2$	$\sum\lvert\beta\rvert$
Coefficient behavior	Shrinks toward 0, never exactly 0	Can be exactly 0 (sparse)
Feature selection	No — keeps all features	Yes — drops irrelevant features
Correlated features	Spreads weight among them	Picks one, zeros others
Best when	Many features, all contribute	Many features, few matter



Kernel Regression

Going non-linear without leaving the linear framework

The Kernel Trick — Why We Need It

Linear models are limited to straight lines. What if the true relationship is curved?

The Problem

Linear regression fits $y = \beta X$, but many real-world relationships are non-linear.

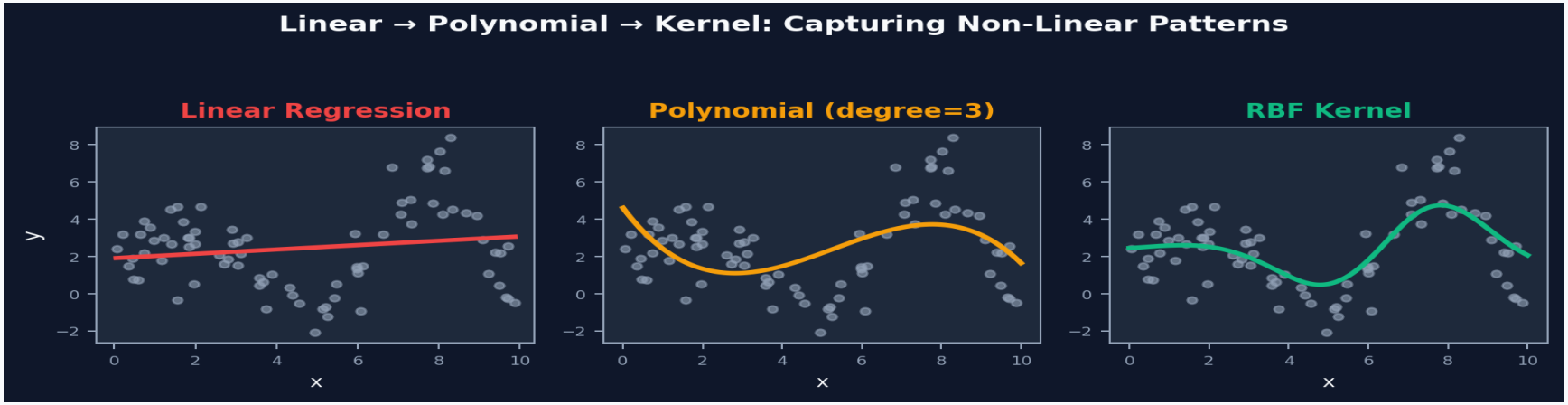
One option: manually create polynomial features (x^2, x^3, x_1x_2...). But this explodes in high dimensions.

The Kernel Trick

Instead of explicitly creating new features, use a kernel function $K(x_i, x_j)$ that implicitly computes similarity in a higher-dimensional space.

The model stays efficient, but captures non-linear patterns.

Common Kernel Functions



Common Kernel Functions:

Linear

$K(x,z) = x^T z$

No transformation.
Same as standard linear.

RBF (Gaussian)

$K(x,z) = \exp(-\gamma\|x-z\|^2)$

Most popular. Captures local, smooth patterns.

Polynomial

$K(x,z) = (\gamma x^T z + r)^d$

Captures polynomial interactions of degree d.

γ controls "closeness": small γ = smooth fit (many points influence), large γ = wiggly fit (only very nearby points matter).

Kernel Regression in Practice

```python
from sklearn.kernel_ridge import KernelRidge
from sklearn.preprocessing import StandardScaler
from sklearn.pipeline import make_pipeline
from sklearn.model_selection import cross_val_score
# RBF kernel: captures non-linear patterns
kr_rbf = make_pipeline(StandardScaler(),
    KernelRidge(kernel='rbf', alpha=1.0, gamma=0.1))
scores = cross_val_score(kr_rbf, X_train, y_train, cv=5, scoring='r2')
print(f"KernelRidge (RBF)  CV R²: {scores.mean():.3f}")
# Compare different kernels
for kern in ['linear', 'poly', 'rbf']:
    kr = make_pipeline(StandardScaler(),
        KernelRidge(kernel=kern, alpha=1.0, gamma=0.1))
    sc = cross_val_score(kr, X_train, y_train, cv=5, scoring='r2')
    print(f"  {kern:6s}  CV R²: {sc.mean():.3f}")
```

Key params: kernel ('rbf','poly','linear') | alpha (regularization strength) | gamma (RBF width: small→smooth, large→wiggly)

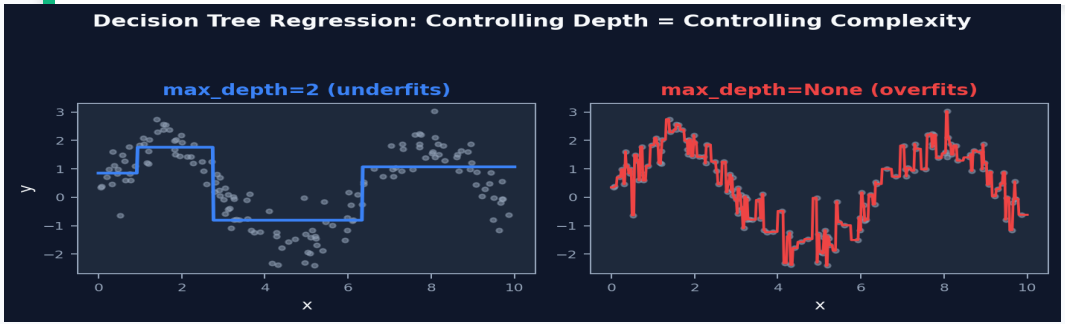
Tree-Based Models & Boosting

From decision trees to XGBoost

Decision Tree Regression — Quick Overview

Core Idea

- Split feature space into rectangular regions using if-then rules
- Each leaf predicts the mean of its training samples
- Handles non-linearity naturally, no feature scaling needed
- But: overfits easily (deep trees memorize noise)




```python
from sklearn.tree import DecisionTreeRegressor
tree = DecisionTreeRegressor(
    max_depth=5, min_samples_leaf=10)
tree.fit(X_train, y_train)
print(f"Train R²: {tree.score(X_train, y_train):.3f}")
print(f"Test  R²: {tree.score(X_test, y_test):.3f}")
```

Key: control max_depth to balance overfitting vs underfitting.

Random Forest — Quick Overview

Idea: Train many trees on random data subsets, average their predictions.

- Each tree sees a bootstrap sample + random feature subset per split
- Averaging reduces variance (overfitting) of individual trees
- Very robust, hard to mess up — great default for non-linear problems
- But: not the best accuracy, and can be slow with many trees

```
from sklearn.ensemble import RandomForestRegressor
rf = RandomForestRegressor(
    n_estimators=200, max_depth=15,
    min_samples_leaf=5, n_jobs=-1, random_state=42
)
rf.fit(X_train, y_train)
print(f"RF R\u00B2: {rf.score(X_test, y_test):.3f}")
```

Random Forest = Bagging (parallel, independent trees). Next: Boosting (sequential, error-correcting trees).

From Bagging to Boosting

Bagging (Random Forest)

- Trees trained independently
- Each on a random data subset
- Predictions averaged
- **Reduces variance (overfitting)**
- Can run in parallel

Boosting (XGBoost)

- Trees trained sequentially
- Each corrects predecessor’s errors
- Predictions summed (weighted)
- **Reduces bias (underfitting)**
- Inherently sequential



Gradient Boosting — Step by Step

1

$F0(x) = \text{mean}(y)$

Start with average

2

$r1 = y - F0(x)$

Compute residuals

3

$h1 = \text{fit tree to } r1$

Fit weak learner to errors

4

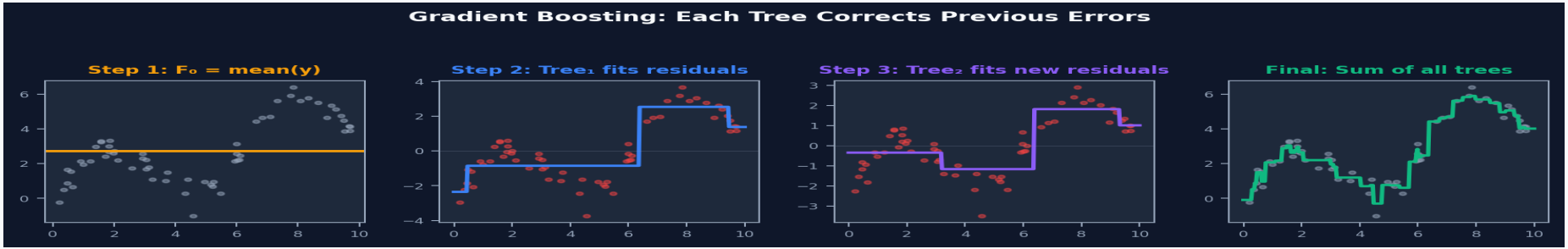
$F1 = F0 + \eta \cdot h1$

Update (learning rate η)

...

Repeat N times

Each step reduces error



Key Hyperparameters:

`n_estimators`	50–500 trees	`learning_rate (eta)`	0.01–0.1	`max_depth`	3–6 (shallow = weak learner)

XGBoost — Why It Dominates



eXtreme Gradient Boosting

- Built-in L1 & L2 regularization on tree weights
- Handles missing values natively (learns optimal split direction)
- Column subsampling (like Random Forest) + row subsampling
- Early stopping: monitors validation loss, stops when no improvement
- GPU support for massive speed-up on large datasets

"When in doubt, XGBoost." — Virtually every Kaggle tabular data competition winner uses it (or LightGBM).

```
import xgboost as xgb  # pip install xgboost
xgb_model = xgb.XGBRegressor(
    n_estimators=500, learning_rate=0.05,
    max_depth=6, reg_alpha=0.1, reg_lambda=1.0,
    subsample=0.8, colsample_bytree=0.8,
    random_state=42
)
xgb_model.fit(X_train, y_train,
              eval_set=[(X_test, y_test)], verbose=False, callbacks=[xgb.callback.EarlyStopping(rounds=20)])
print(f"R\u00B2: {xgb_model.score(X_test, y_test):.3f}")
```

XGBoost — Key Hyperparameters

Parameter	Description	Typical Range
`n_estimators`	Number of trees. Start with 500+, use early_stopping_rounds to find optimum.	100 – 5000
`learning_rate (eta)`	Step shrinkage. Lower = more trees needed but better generalization.	0.01 – 0.3
`max_depth`	Maximum tree depth. Controls complexity of each tree.	3 – 10
`subsample`	Fraction of rows used per tree. Adds randomness, reduces overfitting.	0.6 – 1.0
`colsample_bytree`	Fraction of features used per tree. Like Random Forest's max_features.	0.5 – 1.0
`reg_alpha (L1)`	L1 regularization on leaf weights. Encourages sparsity.	0 – 10
`reg_lambda (L2)`	L2 regularization on leaf weights. Smooths predictions.	0 – 10
`early_stopping_rounds`	Stop if no improvement for N rounds on validation set.	10 – 50

Tip: Start with defaults, then tune learning_rate + max_depth first. Use RandomizedSearchCV for efficiency.

XGBoost — Hyperparameter Tuning

```
from sklearn.model_selection import RandomizedSearchCV
from scipy.stats import uniform, randint
import xgboost as xgb
param_dist = {
    'n_estimators': randint(100, 1000),
    'learning_rate': uniform(0.01, 0.29),
    'max_depth': randint(3, 10),
    'subsample': uniform(0.6, 0.4),
    'colsample_bytree': uniform(0.5, 0.5),
    'reg_alpha': uniform(0, 5),
    'reg_lambda': uniform(0, 5),
}
search = RandomizedSearchCV(
    xgb.XGBRegressor(random_state=42),
    param_dist, n_iter=50, cv=5, scoring='r2',
    random_state=42, n_jobs=-1
)
search.fit(X_train, y_train,
           eval_set=[(X_test, y_test)], verbose=False, callbacks=[xgb.callback.EarlyStopping(rounds=20)])
print(f"Best R\u00B2: {search.best_score_:.3f}")
print(f"Best params: {search.best_params_}")
```

LightGBM — Faster Alternative

LightGBM vs XGBoost

- Leaf-wise tree growth (vs level-wise): faster convergence
- Histogram-based splitting: much faster on large data
- Native categorical feature support (no one-hot encoding!)
- Often faster than XGBoost with similar accuracy

```
import lightgbm as lgb  # pip install lightgbm
lgb_model = lgb.LGBMRegressor(
    n_estimators=500, learning_rate=0.05,
    max_depth=-1, num_leaves=31,
    min_child_samples=20,
    reg_alpha=0.1, reg_lambda=1.0,
    random_state=42
)
lgb_model.fit(X_train, y_train,
              eval_set=[(X_test, y_test)],
              callbacks=[lgb.early_stopping(20, verbose=False)])
print(f"R\u00B2: {lgb_model.score(X_test, y_test):.3f}")
```

Evaluation & Metrics

Measuring how good your model really is

Regression Metrics

MSE $1/n \sum(y_i - \hat{y}_i)^2$ — Penalizes large errors. Training objective.

RMSE $\sqrt{MSE}$ — Same units as target. Most interpretable.

MAE $1/n \sum|y_i - \hat{y}_i|$ — Robust to outliers. Equal error weight.

R^2 $1-SS_res/SS_tot$ — Variance explained. 1=perfect, 0=mean.

```
from sklearn.metrics import mean_squared_error, mean_absolute_error, r2_score
rmse = mean_squared_error(y_test, y_pred, squared=False)
```

Cross-Validation

A single train/test split can be misleading. K-Fold CV gives a reliable estimate.




```python
from sklearn.model_selection import cross_val_score
# 5-fold CV with R\u00B2 scoring
scores = cross_val_score(model, X, y, cv=5, scoring='r2')
print(f"R\u00B2 per fold: {scores.round(3)}")
print(f"Mean R\u00B2: {scores.mean():.3f} (+/- {scores.std():.3f})")
# You can also use negative MSE (sklearn convention)
mse_scores = cross_val_score(model, X, y, cv=5,
                             scoring='neg_mean_squared_error')
rmse_scores = (-mse_scores) ** 0.5
print(f"Mean RMSE: {rmse_scores.mean():.3f}")
```

Why negative MSE? sklearn always maximizes; since MSE should be minimized, it returns −MSE. We negate it back.

Putting It All Together

Comparison, full example, and best practices

Algorithm Comparison

Algorithm	Interpretable	Non-linear	Scaling?	Best for
Linear Reg.	⭐⭐⭐	✗	No	Baseline, interpretability
Ridge (L2)	⭐⭐⭐	✗	Yes	Many correlated features
Lasso (L1)	⭐⭐⭐	✗	Yes	Feature selection
Kernel Ridge	⭐	✓	Yes	Non-linear + small/medium data
Decision Tree	⭐⭐⭐	✓	No	Quick prototyping
Random Forest	⭐⭐	✓	No	Robust non-linear
XGBoost	⭐	✓	No	**Best accuracy (tabular)**
LightGBM	⭐	✓	No	Large data + categoricals

Start with Linear → Ridge/Lasso → Kernel Ridge or XGBoost. Compare with cross-validation.

How to Choose Your Algorithm

Need interpretable model?	→ Linear / Ridge / Lasso
Many irrelevant features?	→ Lasso (automatic feature selection)
Non-linear relationships?	→ Kernel Ridge, RF, or XGBoost
Want best accuracy (tabular)?	→ XGBoost / LightGBM + tune hyperparams
Very large dataset?	→ LightGBM (fastest) or SGD-based linear

Golden rule: Always start with a simple baseline (Linear Regression), then compare more complex models.

scikit-learn Pipelines

Always bundle preprocessing + model into a Pipeline to prevent data leakage.

```
from sklearn.pipeline import make_pipeline
from sklearn.preprocessing import StandardScaler
from sklearn.linear_model import Ridge
# BAD: data leakage! Scaler sees test data
scaler = StandardScaler()
X_all_scaled = scaler.fit_transform(X)  # leaks test info!
# GOOD: pipeline keeps preprocessing inside CV
pipe = make_pipeline(StandardScaler(), Ridge(alpha=1.0))
scores = cross_val_score(pipe, X, y, cv=5, scoring='r2')
print(f"R\u00B2: {scores.mean():.3f}")
# Pipeline works like a single model
pipe.fit(X_train, y_train)
pipe.predict(X_test)
pipe.score(X_test, y_test)
```

Data leakage: when information from the test set leaks into training. Pipelines prevent this by ensuring preprocessing is fit only on training folds during CV.

Full Example: California Housing (1/3)



```python
import pandas as pd
import numpy as np
from sklearn.datasets import fetch_california_housing
from sklearn.model_selection import train_test_split, cross_val_score
from sklearn.preprocessing import StandardScaler
from sklearn.pipeline import make_pipeline
from sklearn.metrics import mean_squared_error, r2_score
# 1. Load data
data = fetch_california_housing()
X = pd.DataFrame(data.data, columns=data.feature_names)
y = data.target
print(f"Samples: {X.shape[0]}, Features: {X.shape[1]}")
print(f"Target: median house value (x $100K)")
print(f"Target range: {y.min():.2f} - {y.max():.2f}")
# 2. Split
X_train, X_test, y_train, y_test = train_test_split(
    X, y, test_size=0.2, random_state=42)
```

Full Example: California Housing (2/3)



```
from sklearn.linear_model import LinearRegression, Ridge, Lasso
from sklearn.kernel_ridge import KernelRidge
from sklearn.ensemble import RandomForestRegressor
import xgboost as xgb
# 3. Compare models
models = {
    'Linear':  make_pipeline(StandardScaler(), LinearRegression()),
    'Ridge':   make_pipeline(StandardScaler(), Ridge(alpha=1.0)),
    'Lasso':   make_pipeline(StandardScaler(), Lasso(alpha=0.01)),
    'KernelR': make_pipeline(StandardScaler(),
                   KernelRidge(kernel='rbf', alpha=1.0, gamma=0.1)),
    'RF':      RandomForestRegressor(n_estimators=200, random_state=42),
    'XGBoost': xgb.XGBRegressor(n_estimators=300, learning_rate=0.1,
                                max_depth=6, random_state=42),
}
for name, model in models.items():
    scores = cross_val_score(model, X_train, y_train,
                             cv=5, scoring='r2')
    print(f"{name:8s}  CV R\u00B2 = {scores.mean():.3f} "
          f"(+/- {scores.std():.3f})")
```

Expected: XGBoost > KernelRidge > RF > Ridge ≈ Lasso > Linear. Non-linear models win on this dataset!

Full Example: California Housing (3/3)



```
# 4. Tune the best model (XGBoost)
from sklearn.model_selection import RandomizedSearchCV
from scipy.stats import uniform, randint
param_dist = {
    'learning_rate': uniform(0.01, 0.19),
    'max_depth': randint(3, 10),
    'n_estimators': randint(200, 1000),
    'subsample': uniform(0.6, 0.4),
}
search = RandomizedSearchCV(
    xgb.XGBRegressor(random_state=42),
    param_dist, n_iter=30, cv=5, scoring='r2', n_jobs=-1)
search.fit(X_train, y_train)
# 5. Final evaluation on held-out test set
best = search.best_estimator_
y_pred = best.predict(X_test)
print(f"Test RMSE: {np.sqrt(mean_squared_error(y_test,y_pred)):.3f}")
print(f"Test R\u00B2:  {r2_score(y_test, y_pred):.3f}")
```

Key Takeaways

- Linear Regression is your foundation — understand it deeply (OLS, GD)
- Regularization (Ridge/Lasso) is essential when dealing with many features
- Kernel trick extends linear models to capture non-linear relationships
- XGBoost/LightGBM are the go-to for best accuracy on tabular data
- Always use cross-validation — never trust a single train/test split
- Pipelines prevent data leakage and keep your code clean
- Start simple, add complexity only when justified by validation metrics

```
Consistent API:  pipe.fit(X_train, y_train)  >>  pipe.predict(X_test)  >>  pipe.score(X_test, y_test)
```