Machine Learning with R and Python

Don Hale

2025-09-22

Table of contents

About

This book is a companion to the Machine Learning Class. It will be a repository of R and Python Code. The chapters will be updated as we progress through the class.

1 Introduction

1.1 Introduction to Machine Learning

Machine Learning (ML) is a branch of artificial intelligence (AI) that focuses on building systems that **learn patterns from data** and make predictions or decisions without being explicitly programmed to perform specific tasks. Instead of writing rules by hand, a machine learning algorithm "learns" from examples in the data.

1.2 What is Machine Learning?

Formally, machine learning is the process of using data to **train a model** to make accurate predictions or decisions. The model identifies patterns in the training data and generalizes these patterns to new, unseen data.

ML can be thought of as:

- **Predictive modeling** – learning a function that maps inputs (features) to outputs (targets).
- **Automated decision-making** – systems that improve their performance over time without explicit reprogramming.

1.3 Major Types of Machine Learning

Machine learning is commonly categorized into several types based on the **nature of the task** and **availability of labeled data**.

1.3.1 1. Supervised Learning

- Uses **labeled data**, meaning that each training example has an input and a known output.
- The goal is to **learn a function** that maps inputs to outputs accurately.
- Typical tasks:
  - **Regression**: Predict a continuous variable (e.g., house prices).
  - **Classification**: Predict a categorical variable (e.g., spam vs. non-spam email).

1.3.2 2. Unsupervised Learning

- Works with **unlabeled data**, where the output is unknown.
- The goal is to **discover patterns or structure** in the data.
- Typical tasks:
  - **Clustering**: Group similar observations together (e.g., customer segmentation).
  - **Dimensionality reduction**: Reduce the number of features while retaining important information (e.g., PCA).

1.3.3 3. Semi-Supervised Learning

- Combines a small amount of labeled data with a large amount of unlabeled data.
- Useful when labeling data is expensive or time-consuming.

1.3.4 4. Reinforcement Learning

- Focuses on **learning through trial and error**.
- An agent learns to take actions in an environment to **maximize cumulative reward**.
- Examples: Robotics, game AI, recommendation systems.

1.4 Objectives of Machine Learning

The main objective of machine learning is to **build models that generalize well** from historical data to make accurate predictions or decisions on **new, unseen data**. Key considerations include:

1. **Accuracy** – How well does the model predict outcomes?

2. **Generalization** – Does the model perform well on unseen data, or is it overfitting the training data?
3. **Interpretability** – Can humans understand how the model makes predictions?

1.5 Bias-Variance Tradeoff

A fundamental concept in machine learning is the **bias-variance tradeoff**, which explains the balance between:

- **Bias**: Error due to overly simplistic models that **underfit** the data.
- **Variance**: Error due to overly complex models that **overfit** the training data.

The goal is to find a model that **minimizes the total prediction error**:

[Total Error = Bias^2 + Variance + Irreducible Error]

- **Underfitting** → High bias, low variance
- **Overfitting** → Low bias, high variance

The optimal model balances bias and variance for the best predictive performance.

1.6 The Machine Learning Process

A typical workflow in machine learning consists of the following steps:

1. **Data Collection**
    - Gather data from experiments, databases, or external sources.
2. **Data Preprocessing**
    - Handle missing values, outliers, and feature engineering.
3. **Model Selection**
    - Choose an appropriate algorithm (e.g., linear regression, random forest, neural networks).
4. **Model Training**

- Fit the model to the training data.

5. **Model Evaluation**
   - Assess performance using metrics such as RMSE, accuracy, precision, recall, or AUC.
6. **Model Tuning**
   - Adjust hyperparameters to improve performance.
7. **Model Deployment**
   - Use the trained model to make predictions on new data.
8. **Monitoring and Maintenance**
   - Continuously track model performance and retrain if necessary.

1.7 Summary

Machine learning allows us to build predictive models that **learn from data**. Understanding the different types of ML, their objectives, and the tradeoffs between bias and variance is critical to applying ML effectively. The remainder of this book explores practical algorithms, their implementation in **R and Python**, and strategies for model selection and interpretation.

2 Model Selection and Regularization - R Version

2.1 Overview

In this section, we'll use the **Ames Housing** dataset to demonstrate model selection and regularization.

We'll cover:

1. Splitting data into training and testing sets
2. Performing **stepwise regression** with cross-validation
3. Running **Ridge** and **Lasso** regression, and visualizing how affects RMSE and model complexity
4. Exploring **Elastic Net**, varying to balance Ridge and Lasso
5. Comparing all models on the test set

2.2 Step 0: Setup and Data Preparation

We'll use a reduced set of variables for speed and clarity.

```
# Suppress messages and warnings globally
knitr::opts_chunk$set(echo = TRUE, message = FALSE, warning = FALSE)

library(tidyverse)
library(caret)
library(glmnet)
library(AmesHousing)
```

```r
library(GGally)

set.seed(123)

# Load data and split into 70% training, 30% testing
ames <- make_ordinal_ames() %>% mutate(id = row_number())
train <- ames %>% sample_frac(0.7)
test  <- anti_join(ames, train, by = "id")

# Select a manageable subset of predictors
keep <- c("Sale_Price", "Bedroom_AbvGr", "Year_Built", "Mo_Sold", "Lot_Area",
          "Street", "Central_Air", "First_Flr_SF", "Second_Flr_SF", "Full_Bath",
          "Half_Bath", "Fireplaces", "Garage_Area", "Gr_Liv_Area", "TotRms_AbvGrd")

train <- train %>% select(all_of(keep))
test  <- test  %>% select(all_of(keep))

# Convert categorical variables to factors
train <- train %>% mutate(across(c(Street, Central_Air), as.factor))
test  <- test %>% mutate(across(c(Street, Central_Air), as.factor))
```

2.3 Step 1: Stepwise Regression with 10-Fold Cross Validation

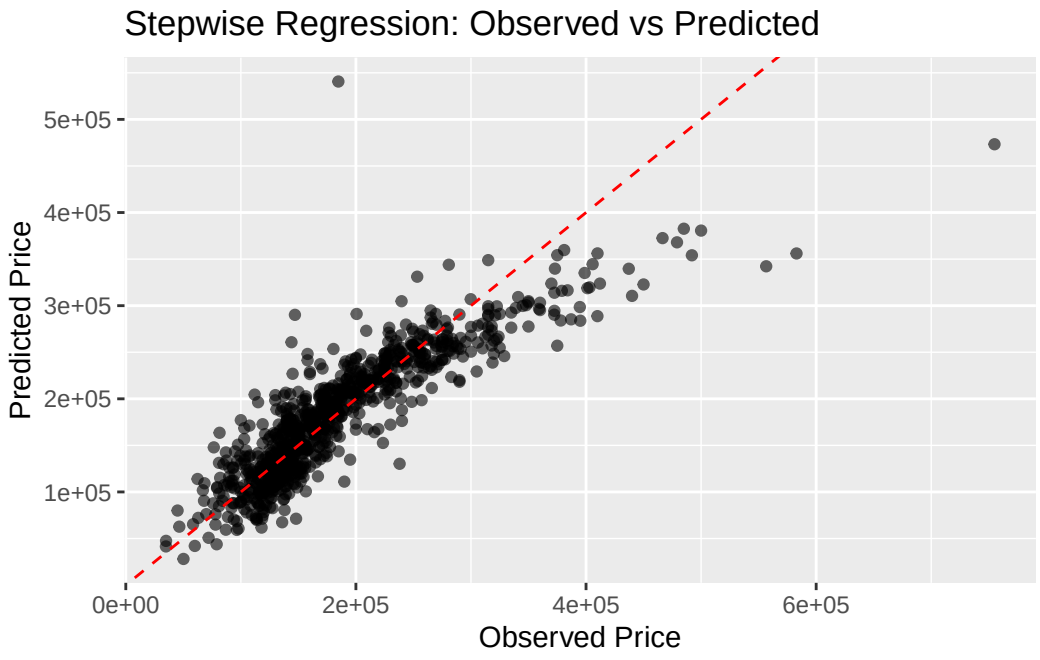
We'll use backward stepwise regression as a traditional benchmark.

```r
ctrl <- trainControl(method = "cv", number = 10)

step_model <- train(
  Sale_Price ~ ., data = train,
  method = "lmStepAIC",
  trControl = ctrl,
  trace = FALSE,
  direction = "backward"
)

# Evaluate on the test set
step_pred <- predict(step_model, newdata = test)
```

```r
step_perf <- postResample(step_pred, test$Sale_Price)
step_perf
```

```
        RMSE     Rsquared          MAE
3.820776e+04 7.670269e-01 2.613098e+04
```

```r
# Visualize predicted vs observed
ggplot(data.frame(obs = test$Sale_Price, pred = step_pred), aes(obs, pred)) +
  geom_point(alpha = 0.6) +
  geom_abline(linetype = "dashed", color = "red") +
  labs(title = "Stepwise Regression: Observed vs Predicted",
       x = "Observed Price", y = "Predicted Price")
```



Note:
Stepwise regression is intuitive and fast, but it can be unstable.

Stepwise regression (forward, backward, or both) selects predictors one at a time based on how much they improve a criterion (like AIC or adjusted R^2).

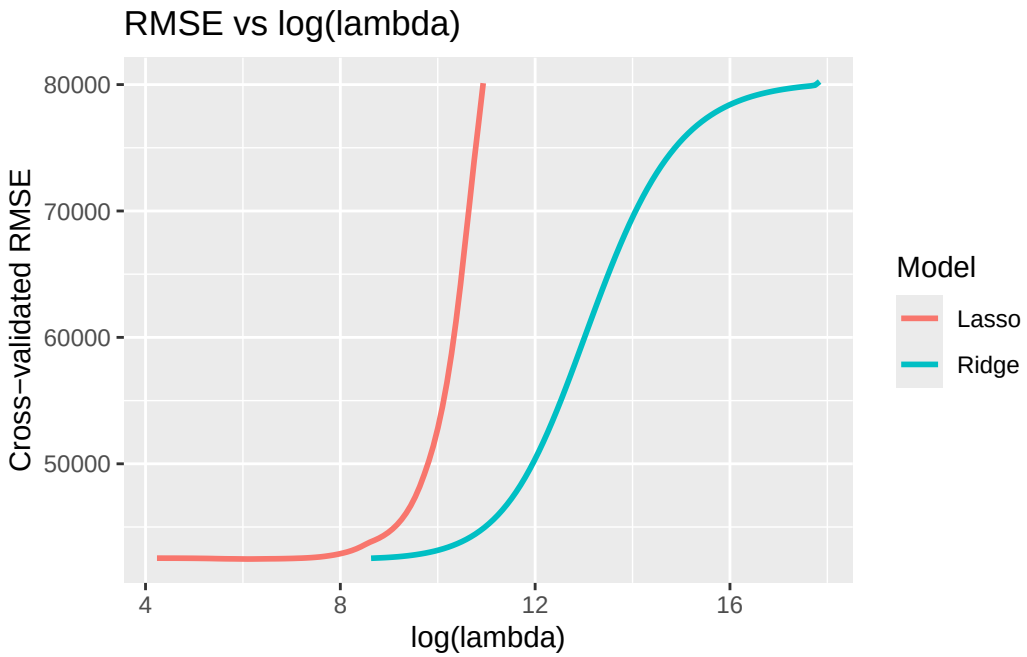
2.4 Step 2: Ridge and Lasso Regression

Ridge and Lasso both shrink coefficients, but in different ways: - Ridge shrinks all coefficients toward zero. - Lasso can set some coefficients *exactly* to zero, performing variable selection.

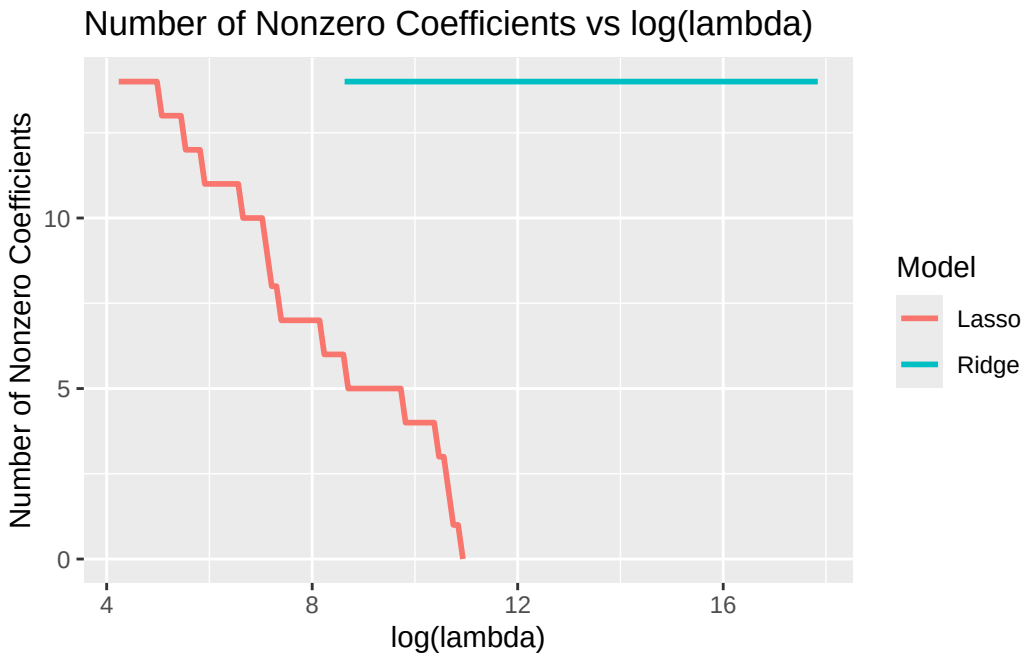
```
x_train <- model.matrix(Sale_Price ~ ., train)[, -1]
y_train <- train$Sale_Price
x_test  <- model.matrix(Sale_Price ~ ., test)[, -1]
y_test  <- test$Sale_Price

set.seed(123)
cv_ridge <- cv.glmnet(x_train, y_train, alpha = 0, nfolds = 10)
cv_lasso <- cv.glmnet(x_train, y_train, alpha = 1, nfolds = 10)
```

2.4.1 Visualizing RMSE and Model Complexity

```
build_path_df <- function(cvfit, label) {
  fit <- cvfit$glmnet.fit
  tibble(
    lambda = fit$lambda,
    log_lambda = log(fit$lambda),
    RMSE = sqrt(cvfit$cvm),
    nonzero = colSums(abs(fit$beta) > 0),
    Model = label
  )
}

ridge_df <- build_path_df(cv_ridge, "Ridge")
lasso_df <- build_path_df(cv_lasso, "Lasso")
df <- bind_rows(ridge_df, lasso_df)

ggplot(df, aes(log_lambda, RMSE, color = Model)) +
  geom_line(size = 1) +
  labs(title = "RMSE vs log(lambda)", y = "Cross-validated RMSE", x = "log(lambda)")
```




```
ggplot(df, aes(log_lambda, nonzero, color = Model)) +
  geom_line(size = 1) +
  labs(title = "Number of Nonzero Coefficients vs log(lambda)",
       y = "Number of Nonzero Coefficients", x = "log(lambda)")
```



Note:
- Increasing increases regularization.
- Ridge never eliminates variables, Lasso can.
- There's a sweet spot where RMSE is minimized.

2.5 Step 3: Elastic Net (Balancing Ridge and Lasso)

Elastic Net introduces to control the mix between Ridge (= 0) and Lasso (= 1).

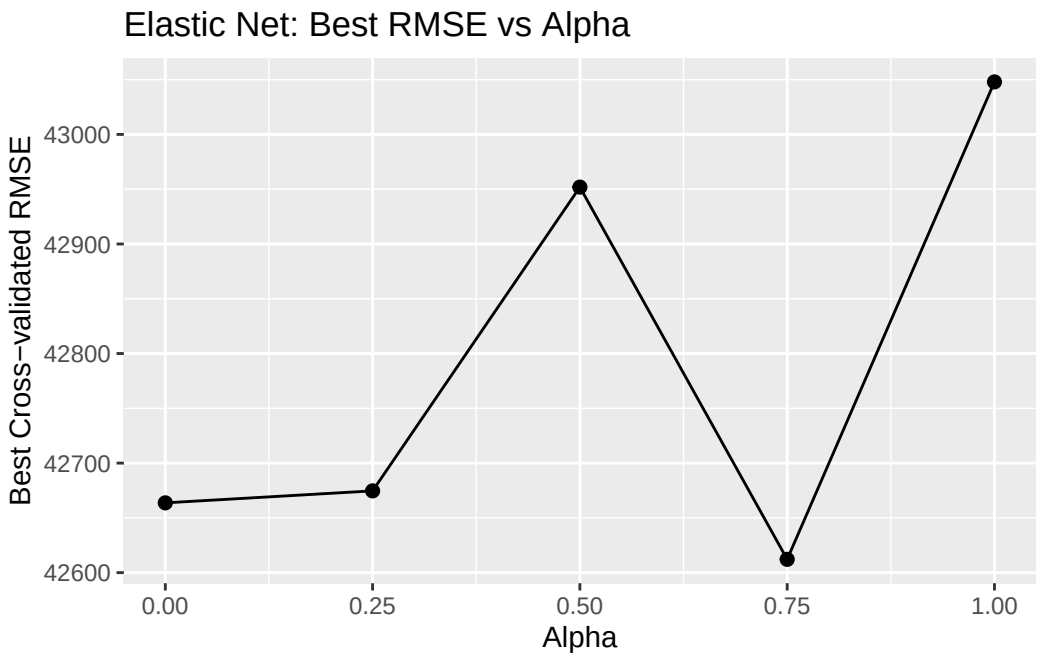
```
alpha_grid <- seq(0, 1, by = 0.25)

elastic_results <- map_df(alpha_grid, function(a) {
  cv_fit <- cv.glmnet(x_train, y_train, alpha = a, nfolds = 10)
  tibble(
    alpha = a,
    best_lambda = cv_fit$lambda.min,
    best_RMSE = sqrt(min(cv_fit$cvm)),
    nonzero = sum(abs(coef(cv_fit, s = "lambda.min")[-1]) > 0)
  )
```

```
})

elastic_results
```

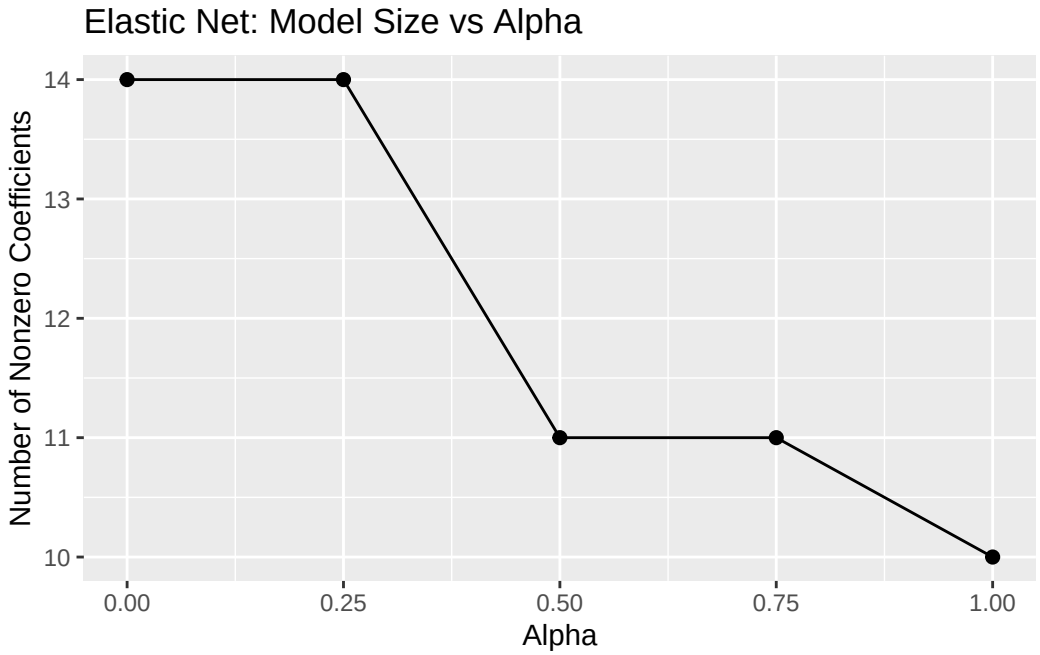
```
# A tibble: 5 x 4
  alpha best_lambda best_RMSE nonzero
  <dbl>       <dbl>     <dbl>   <int>
1  0          5603.    42664.      14
2  0.25       1224.    42675.      14
3  0.5        1552.    42952.      11
4  0.75        650.    42612.      11
5  1          1126.    43048.      10
```

2.5.1 Visualizing Effects

```
ggplot(elastic_results, aes(alpha, best_RMSE)) +
  geom_line() + geom_point(size = 2) +
  labs(title = "Elastic Net: Best RMSE vs Alpha",
       y = "Best Cross-validated RMSE", x = "Alpha")
```



```r
ggplot(elastic_results, aes(alpha, nonzero)) +
  geom_line() + geom_point(size = 2) +
  labs(title = "Elastic Net: Model Size vs Alpha",
       y = "Number of Nonzero Coefficients", x = "Alpha")
```



Note:
Elastic Net combines the strengths of Ridge and Lasso: - Ridge for stability. - Lasso for sparsity. - Often performs best at intermediate values.

2.6 Step 4: Comparing All Models on the Test Set

We'll compare Stepwise, Ridge, Lasso, and the best Elastic Net.

```r
ridge_pred <- predict(cv_ridge, newx = x_test, s = "lambda.min")
lasso_pred <- predict(cv_lasso, newx = x_test, s = "lambda.min")

best_alpha <- elastic_results %>% arrange(best_RMSE) %>% slice(1) %>% pull(alpha)
cv_en_best <- cv.glmnet(x_train, y_train, alpha = best_alpha, nfolds = 10)
elastic_pred <- predict(cv_en_best, newx = x_test, s = "lambda.min")
```

```r
compare_tbl <- bind_rows(
  Stepwise = as_tibble_row(postResample(step_pred, test$Sale_Price)),
  Ridge = as_tibble_row(postResample(ridge_pred, y_test)),
  Lasso = as_tibble_row(postResample(lasso_pred, y_test)),
  Elastic = as_tibble_row(postResample(elastic_pred, y_test)),
  .id = "Model"
)

colnames(compare_tbl) <- c("Model", "RMSE", "Rsquared", "MAE")
compare_tbl <- compare_tbl %>% arrange(RMSE)
compare_tbl
```

```
# A tibble: 4 x 4
  Model      RMSE Rsquared    MAE
  <chr>     <dbl>    <dbl>  <dbl>
1 Stepwise 38208.    0.767 26131.
2 Lasso    38297.    0.767 26045.
3 Elastic  38301.    0.767 26039.
4 Ridge    38412.    0.766 26022.
```

```r
# ------------------------------------------------------------

# Display the best cross-validated Lasso, Ridge, and Elastic Net models


# ------------------------------------------------------------

library(glmnet)
library(knitr)

# Helper function to summarize a fitted cv.glmnet model

summarize_glmnet <- function(cvfit, X_train, y_train, X_test, y_test, model_name) {
  cat("\n==========================================\n")
  cat("Model:", model_name, "\n")
  cat("==========================================\n")

  # Extract best lambda
  best_lambda <- cvfit$lambda.min
```

```r
  cat("Best Lambda (lambda.min):", round(best_lambda, 6), "\n")

  # Coefficients (non-zero)
  coefs <- as.matrix(coef(cvfit, s = "lambda.min"))
  nonzero <- coefs[coefs != 0, , drop = FALSE]

  # Print nonzero coefficients
  cat("\nNonzero Coefficients:\n")
  print(knitr::kable(as.data.frame(nonzero), digits = 4))

  # Predictions and performance
  pred_train <- predict(cvfit, newx = X_train, s = "lambda.min")
  pred_test  <- predict(cvfit, newx = X_test,  s = "lambda.min")

  rmse <- function(y, yhat) sqrt(mean((y - yhat)^2))
  rsq  <- function(y, yhat) 1 - sum((y - yhat)^2) / sum((y - mean(y))^2)

  train_rmse <- rmse(y_train, pred_train)
  test_rmse  <- rmse(y_test, pred_test)
  test_r2    <- rsq(y_test, pred_test)

  # Return one-row summary
  data.frame(
    Model = model_name,
    Lambda = round(best_lambda, 6),
    Train_RMSE = round(train_rmse, 2),
    Test_RMSE = round(test_rmse, 2),
    Test_R2 = round(test_r2, 3),
    Nonzero_Coeff = nrow(nonzero)
  )
}

# Summarize and display all three models
results <- rbind(
  summarize_glmnet(cv_lasso, x_train, y_train, x_test, y_test, "Lasso"),
  summarize_glmnet(cv_ridge, x_train, y_train, x_test, y_test, "Ridge"),
  summarize_glmnet(cv_en_best, x_train, y_train, x_test, y_test, "Elastic Net")
)
```

```
==========================================
Model: Lasso
```

```
==========================================
Best Lambda (lambda.min): 444.0297

Nonzero Coefficients:
```

	lambda.min
(Intercept)	-1536712.4502
Bedroom_AbvGr	-12871.7248
Year_Built	786.0097
Lot_Area	0.3320
StreetPave	14782.6929
Central_AirY	3463.4742
First_Flr_SF	22.9872
Half_Bath	-1509.7005
Fireplaces	11647.3233
Garage_Area	62.2549
Gr_Liv_Area	74.6167
TotRms_AbvGrd	1301.9029

```
==========================================
Model: Ridge
==========================================
Best Lambda (lambda.min): 5603.019

Nonzero Coefficients:
```

	lambda.min
(Intercept)	-1398127.9251
Bedroom_AbvGr	-13051.9562
Year_Built	706.9534
Mo_Sold	105.9352
Lot_Area	0.4001
StreetPave	22908.4375
Central_AirY	6951.0203
First_Flr_SF	45.3168
Second_Flr_SF	21.9030
Full_Bath	4203.4612
Half_Bath	-26.1950
Fireplaces	12666.1199

```
|Garage_Area   |       64.5695|
|Gr_Liv_Area   |       42.3845|
|TotRms_AbvGrd |     3067.4622|

==========================================
Model: Elastic Net
==========================================
Best Lambda (lambda.min): 592.0396

Nonzero Coefficients:


|              |    lambda.min|
|:-------------|-------------:|
|(Intercept)   | -1532932.2050|
|Bedroom_AbvGr |   -12808.7530|
|Year_Built    |      783.9250|
|Lot_Area      |        0.3319|
|StreetPave    |    14821.6671|
|Central_AirY  |     3497.9772|
|First_Flr_SF  |       23.2544|
|Half_Bath     |    -1319.3051|
|Fireplaces    |    11686.0135|
|Garage_Area   |       62.4230|
|Gr_Liv_Area   |       73.9778|
|TotRms_AbvGrd |     1388.0576|
```

```
# Combined performance table
cat("\n\n***Model Performance Comparison***\n")
```

```
***Model Performance Comparison***
```

```
kable(results, caption = "Performance of Cross-Validated Regularized Models")
```

Table 2.1: Performance of Cross-Validated Regularized Models

Model	Lambda	Train_RMSE	Test_RMSE	Test_R2	Nonzero_Coeff
Lasso	444.0297	41745.42	38296.94	0.764	12

Model	Lambda	Train_RMSE	Test_RMSE	Test_R2	Nonzero_Coeff
Ridge	5603.0194	41830.59	38411.78	0.763	15
Elastic Net	592.0396	41748.11	38300.92	0.764	12

2.7 Key Takeaways

- Stepwise selection is simple but can lead to unstable models.
- Ridge adds stability via coefficient shrinkage.
- Lasso enforces sparsity by removing weak predictors.
- Elastic Net balances both effects.
- Regularization often produces models that generalize better than purely stepwise ones.

3 Model Selection and Regularization - Python Version

3.1 Overview

This section parallels the R version of Model Selection and Regularization.
We will:

1. Load the Ames Housing data
2. Split into training and test sets
3. Perform Ridge, Lasso, and Elastic Net regressions with 10-fold CV
4. Visualize RMSE and coefficient shrinkage patterns
5. Compare test set performance

3.2 Step 0: Setup and Data

We'll use scikit-learn for modeling.
If the CSVs `ames_training.csv` and `ames_testing.csv` are not present, you can load Ames Housing via the `fetch_openml` function.

```
import warnings
warnings.filterwarnings("ignore")

import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
from sklearn.datasets import fetch_openml
from sklearn.model_selection import train_test_split, KFold, cross_val_score
```

```
from sklearn.preprocessing import OneHotEncoder, StandardScaler
from sklearn.compose import ColumnTransformer
from sklearn.pipeline import Pipeline
from sklearn.linear_model import RidgeCV, LassoCV, ElasticNetCV
from sklearn.metrics import mean_squared_error

# Load Ames dataset directly
ames = fetch_openml(name="house_prices", as_frame=True)
df = ames.frame

# Subset variables to match the R version
keep = [
    "SalePrice","BedroomAbvGr","YearBuilt","MoSold","LotArea","Street","CentralAir",
    "1stFlrSF","2ndFlrSF","FullBath","HalfBath","Fireplaces","GarageArea",
    "GrLivArea","TotRmsAbvGrd"
]
df = df[keep].copy()

# Clean up names to match Python variable rules
df.columns = ["Sale_Price","Bedroom_AbvGr","Year_Built","Mo_Sold","Lot_Area","Street",
              "Central_Air","First_Flr_SF","Second_Flr_SF","Full_Bath","Half_Bath",
              "Fireplaces","Garage_Area","Gr_Liv_Area","TotRms_AbvGrd"]

# Drop missing rows
df = df.dropna()

# Train/test split (70/30)
train, test = train_test_split(df, test_size=0.3, random_state=123)

# Identify predictors and target
y_train = train["Sale_Price"]
y_test  = test["Sale_Price"]
X_train = train.drop(columns="Sale_Price")
X_test  = test.drop(columns="Sale_Price")

cat_vars = ["Street","Central_Air"]
num_vars = [c for c in X_train.columns if c not in cat_vars]

# Preprocess: scale numeric, one-hot encode categorical
preprocessor = ColumnTransformer([
    ("num", StandardScaler(), num_vars),
    ("cat", OneHotEncoder(drop="first"), cat_vars)
```

```
])
```

3.3 Step 1: Ridge Regression

We'll use cross-validation to tune (alpha).
Then we visualize RMSE vs log(lambda) and the number of nonzero coefficients.

```
alphas = np.logspace(4, -4, 80)
ridge = Pipeline([
    ("prep", preprocessor),
    ("model", RidgeCV(alphas=alphas, scoring="neg_mean_squared_error", cv=10))
])
ridge.fit(X_train, y_train)

ridge_best = ridge.named_steps["model"].alpha_
print("Best Ridge alpha:", ridge_best)

# Evaluate RMSE on test set
ridge_pred = ridge.predict(X_test)
ridge_rmse = np.sqrt(mean_squared_error(y_test, ridge_pred))
print("Test RMSE (Ridge):", ridge_rmse)
```

```
Best Ridge alpha: 94.33732216299774
Test RMSE (Ridge): 37055.84511170759
```

3.4 Step 2: Lasso Regression

Lasso adds variable selection — some coefficients go exactly to zero.

```
lasso = Pipeline([
    ("prep", preprocessor),
    ("model", LassoCV(alphas=alphas, cv=10, max_iter=20000))
])
lasso.fit(X_train, y_train)
```

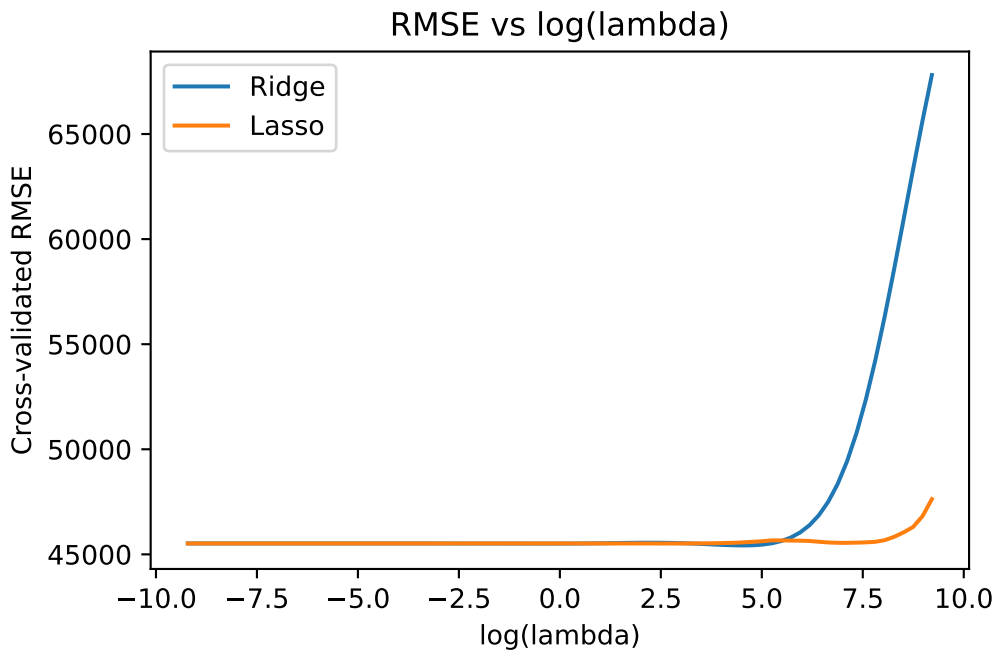
```
lasso_best = lasso.named_steps["model"].alpha_
print("Best Lasso alpha:", lasso_best)

lasso_pred = lasso.predict(X_test)
lasso_rmse = np.sqrt(mean_squared_error(y_test, lasso_pred))
print("Test RMSE (Lasso):", lasso_rmse)
```

```
Best Lasso alpha: 11.568875283162821
Test RMSE (Lasso): 37074.74978063246
```

3.5 Step 3: Visualize Coefficient Shrinkage and RMSE

Let's visualize how RMSE and model sparsity change with .

```
# Compute RMSE path manually
def cv_rmse(model_class, X, y, alphas):
    rmse = []
    for a in alphas:
        model = Pipeline([
            ("prep", preprocessor),
            ("model", model_class(alpha=a, max_iter=20000))
        ])
        scores = cross_val_score(model, X, y, scoring="neg_mean_squared_error", cv=KFold(10,
        rmse.append(np.sqrt(-scores.mean()))
    return np.array(rmse)

from sklearn.linear_model import Ridge, Lasso
ridge_rmse_path = cv_rmse(Ridge, X_train, y_train, alphas)
lasso_rmse_path = cv_rmse(Lasso, X_train, y_train, alphas)

plt.figure()
plt.plot(np.log(alphas), ridge_rmse_path, label="Ridge")
plt.plot(np.log(alphas), lasso_rmse_path, label="Lasso")
plt.xlabel("log(lambda)")
plt.ylabel("Cross-validated RMSE")
plt.title("RMSE vs log(lambda)")
plt.legend()
plt.show()
```



3.6 Step 4: Elastic Net

Elastic Net blends Ridge and Lasso.
We'll vary (the mix) and visualize the tradeoff.

```
l1_ratios = np.linspace(0, 1, 6)
en_cv = Pipeline([
    ("prep", preprocessor),
    ("model", ElasticNetCV(l1_ratio=l1_ratios, alphas=alphas, cv=10, max_iter=50000))
])
en_cv.fit(X_train, y_train)

print("Best Elastic Net alpha:", en_cv.named_steps["model"].alpha_)
print("Best Elastic Net l1_ratio:", en_cv.named_steps["model"].l1_ratio_)

en_pred = en_cv.predict(X_test)
en_rmse = np.sqrt(mean_squared_error(y_test, en_pred))
print("Test RMSE (Elastic Net):", en_rmse)
```

```
Best Elastic Net alpha: 0.55825862688627
Best Elastic Net l1_ratio: 0.8
Test RMSE (Elastic Net): 37134.497858434275
```

3.7 Step 5: Compare Models

```python
results = pd.DataFrame({
    "Model": ["Ridge", "Lasso", "Elastic Net"],
    "Test_RMSE": [ridge_rmse, lasso_rmse, en_rmse]
})
print(results.sort_values("Test_RMSE"))
```

```
         Model     Test_RMSE
0        Ridge  37055.845112
1        Lasso  37074.749781
2  Elastic Net  37134.497858
```

3.8 Summary

- Ridge stabilizes coefficients by shrinking them.
- Lasso enforces sparsity by zeroing weak predictors.
- Elastic Net blends both for balance.
- Regularization often beats pure stepwise regression on unseen data.

4 GAM – Piecewise, MARS, LOESS, and Splines - R Version

Goal: Predict `Sale_Price` from a focused set of Ames features using increasingly flexible methods in the GAM framework.

4.1 1. Setup and Data

```
set.seed(4321)

library(tidyverse)
library(dplyr)
library(tidyr)
library(AmesHousing)   # make_ordinal_ames()
library(earth)         # MARS
library(segmented)     # piecewise (segmented) regression
library(splines)       # regression splines (bs, ns) - kept for reference
library(mgcv)          # smoothing splines via GAM
library(caret)         # data splitting / utilities
library(Metrics)       # rmse, mse helpers (plus we'll compute R2 manually)
theme_set(theme_minimal())
```

```
# Limit to requested columns
cols <- c("Sale_Price","Bedroom_AbvGr","Year_Built","Mo_Sold","Lot_Area","Street",
          "Central_Air","First_Flr_SF","Second_Flr_SF","Full_Bath","Half_Bath",
          "Fireplaces","Garage_Area","Gr_Liv_Area","TotRms_AbvGrd")

ames <- make_ordinal_ames() |>
  dplyr::select(dplyr::all_of(cols)) |>
  tidyr::drop_na()

# Train/test split
set.seed(4321)
```

```r
idx <- sample.int(nrow(ames), size = floor(0.7*nrow(ames)))
train <- ames[idx,]
test  <- ames[-idx,]

glimpse(train)
```

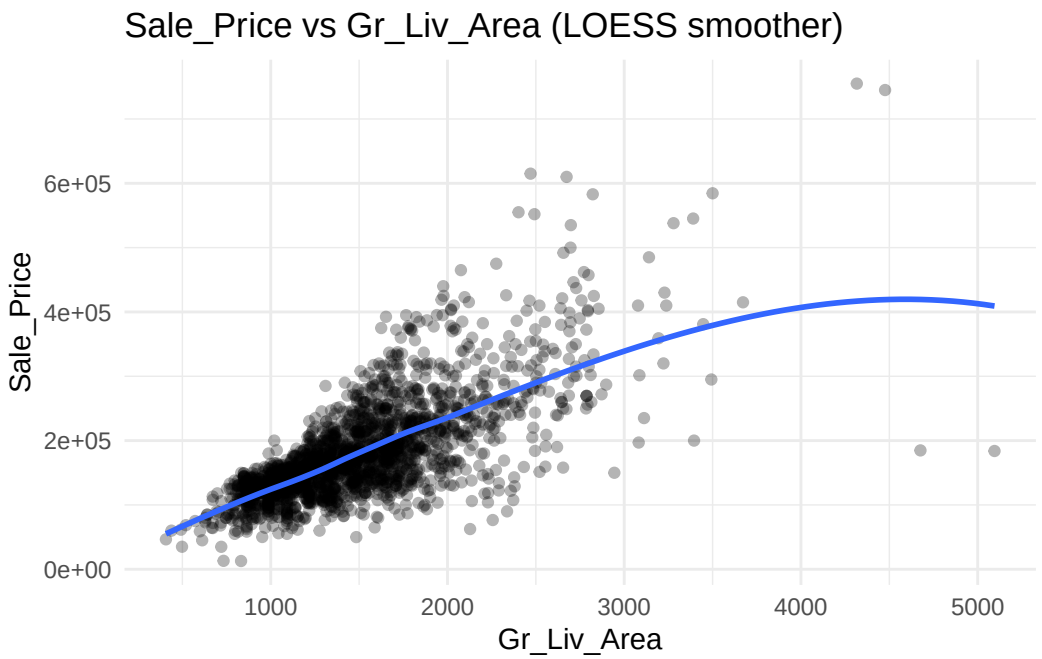
```
Rows: 2,051
Columns: 15
$ Sale_Price    <int> 274000, 75200, 329900, 145400, 108000, 184000, 176000, 1~
$ Bedroom_AbvGr <int> 3, 2, 4, 3, 2, 2, 3, 3, 3, 3, 3, 3, 3, 4, 3, 2, 3, 3, 2,~
$ Year_Built    <int> 2001, 1922, 2005, 1926, 1949, 1999, 1962, 1915, 1999, 19~
$ Mo_Sold       <int> 1, 9, 8, 5, 5, 6, 5, 6, 9, 6, 8, 5, 7, 7, 4, 8, 5, 11, 5~
$ Lot_Area      <int> 9720, 3672, 11643, 7000, 8777, 5858, 19296, 8094, 3768, ~
$ Street        <fct> Pave, Pave, Pave, Pave, Pave, Pave, Pave, Pave, Pave, Pa~
$ Central_Air   <fct> Y, Y, Y, Y, Y, Y, Y, Y, Y, Y, Y, Y, Y, Y, Y, Y, Y, Y, Y,~
$ First_Flr_SF  <int> 1366, 816, 1544, 861, 1126, 1337, 1382, 1048, 713, 792, ~
$ Second_Flr_SF <int> 581, 0, 814, 424, 0, 0, 0, 720, 739, 725, 0, 1151, 695, ~
$ Full_Bath     <int> 2, 1, 2, 1, 2, 2, 1, 2, 2, 1, 1, 2, 2, 2, 1, 2, 2, 1, 1,~
$ Half_Bath     <int> 1, 0, 1, 0, 0, 0, 0, 0, 1, 0, 1, 1, 1, 0, 1, 0, 1, 0, 0,~
$ Fireplaces    <int> 1, 0, 1, 0, 0, 1, 1, 0, 0, 2, 0, 1, 2, 1, 0, 1, 0, 0, 1,~
$ Garage_Area   <dbl> 725, 100, 784, 506, 520, 511, 884, 576, 506, 400, 0, 434~
$ Gr_Liv_Area   <int> 1947, 816, 2358, 1285, 1126, 1337, 1382, 1768, 1452, 151~
$ TotRms_AbvGrd <int> 7, 5, 10, 6, 5, 5, 6, 8, 6, 7, 5, 8, 7, 5, 6, 7, 6, 6, 5~
```

```r
summary(train$Sale_Price)
```

```
   Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
  12789  130000  161000  180897  215000  755000
```

4.2 2. Quick Visual: Nonlinearity is Common

```r
ggplot(train, aes(Gr_Liv_Area, Sale_Price)) +
  geom_point(alpha=.3) +
  geom_smooth(method="loess", se=FALSE) +
  labs(title="Sale_Price vs Gr_Liv_Area (LOESS smoother)")
```



Takeaway: The relationship bends—nonlinear methods can help.

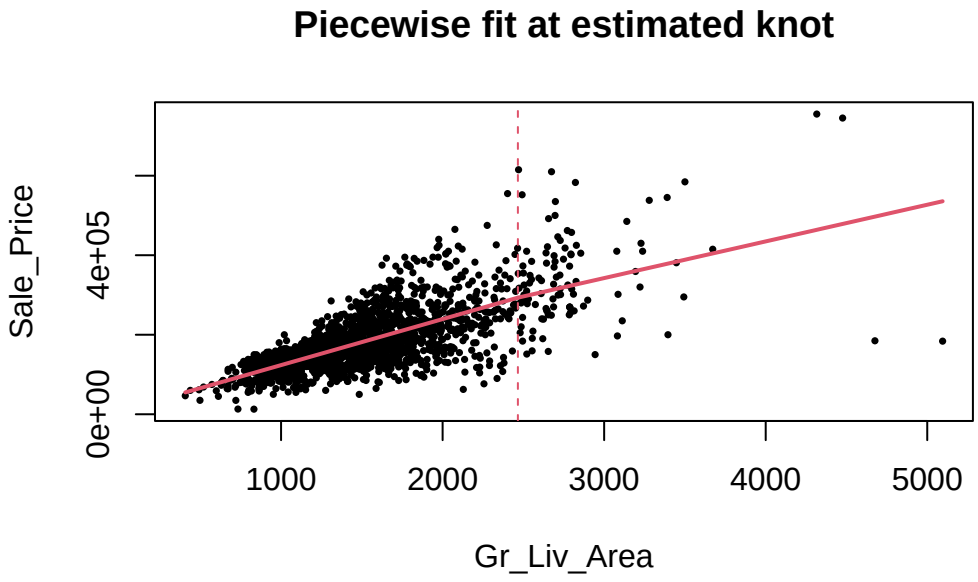
4.3 3. Piecewise Regression

```
m_lin <- lm(Sale_Price ~ Gr_Liv_Area, data=train)
m_seg_init <- lm(Sale_Price ~ Gr_Liv_Area, data=train)
m_seg <- segmented(m_seg_init, seg.Z = ~ Gr_Liv_Area, psi = list(Gr_Liv_Area = 2000))
summary(m_seg)
```

```
    ***Regression Model with Segmented Relationship(s)***

Call:
segmented.lm(obj = m_seg_init, seg.Z = ~Gr_Liv_Area, psi = list(Gr_Liv_Area = 2000))

Estimated Break-Point(s):
                  Est.  St.Err
psi1.Gr_Liv_Area 2466 332.881
```

```
Coefficients of the linear terms:
               Estimate Std. Error t value Pr(>|t|)
(Intercept)    7901.186   4698.641   1.682   0.0928 .
Gr_Liv_Area     115.705      3.171  36.488   <2e-16 ***
U1.Gr_Liv_Area  -23.479     12.436  -1.888       NA
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

Residual standard error: 55110 on 2047 degrees of freedom
Multiple R-Squared: 0.5124,  Adjusted R-squared: 0.5116

Boot restarting based on 6 samples. Last fit:
Convergence attained in 1 iterations (rel. change 0)
```

```r
plot(train$Gr_Liv_Area, train$Sale_Price, pch=16, cex=.5,
     xlab="Gr_Liv_Area", ylab="Sale_Price", main="Piecewise fit at estimated knot")
plot(m_seg, add=TRUE, col=2, lwd=2)
abline(v = m_seg$psi[,"Est."], lty=2, col=2)
```



Interpretation (piecewise): Two linear slopes before/after a knot. Useful when you expect a threshold effect.

4.4 4. MARS (earth) – From Simple to Interactions

MARS = *Multivariate Adaptive Regression Splines*. It builds **hinge** basis functions like `h(x - c) = max(0, x - c)` and can **interact** them.
- A term such as `h(Gr_Liv_Area - 1500)` means *once* `Gr_Liv_Area` exceeds 1500, the fitted line's slope can change.
- An interaction like `h(Gr_Liv_Area - 1500) * Central_AirY` means the slope change applies when `Central_Air == "Y"`.

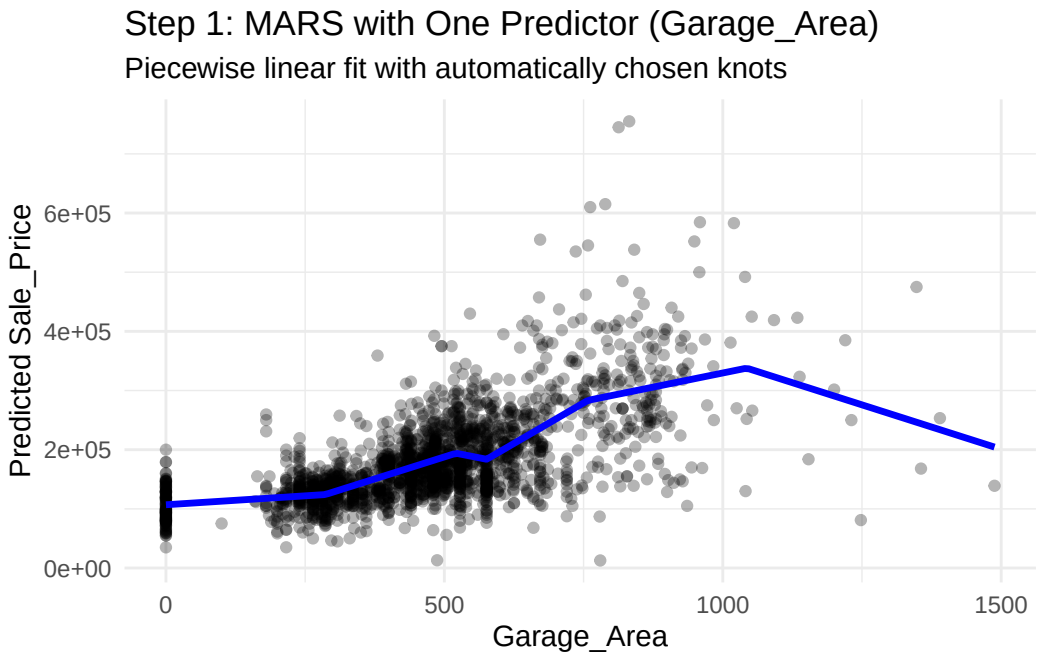
4.4.1 4A. Univariate MARS (Garage_Area)

```
mars1 <- earth(Sale_Price ~ Garage_Area, data=train)
summary(mars1)
```

```
Call: earth(formula=Sale_Price~Garage_Area, data=train)

                    coefficients
(Intercept)           124159.039
h(286-Garage_Area)       -60.257
h(Garage_Area-286)       297.277
h(Garage_Area-521)      -483.642
h(Garage_Area-576)       733.859
h(Garage_Area-758)      -356.460
h(Garage_Area-1043)     -490.873

Selected 7 of 7 terms, and 1 of 1 predictors
Termination condition: RSq changed by less than 0.001 at 7 terms
Importance: Garage_Area
Number of terms at each degree of interaction: 1 6 (additive model)
GCV 3427475346    RSS 6.94092e+12    GRSq 0.4492014    RSq 0.4556309
```

```
grid <- tibble(Garage_Area = seq(min(train$Garage_Area), max(train$Garage_Area), length.out=
grid$pred <- predict(mars1, newdata=grid)

ggplot(train, aes(Garage_Area, Sale_Price)) +
  geom_point(alpha=.3) +
  geom_line(data=grid, aes(Garage_Area, pred), color="blue", linewidth=1.2) +
```

```
  labs(title="Step 1: MARS with One Predictor (Garage_Area)",
       subtitle="Piecewise linear fit with automatically chosen knots",
       y="Predicted Sale_Price")
```



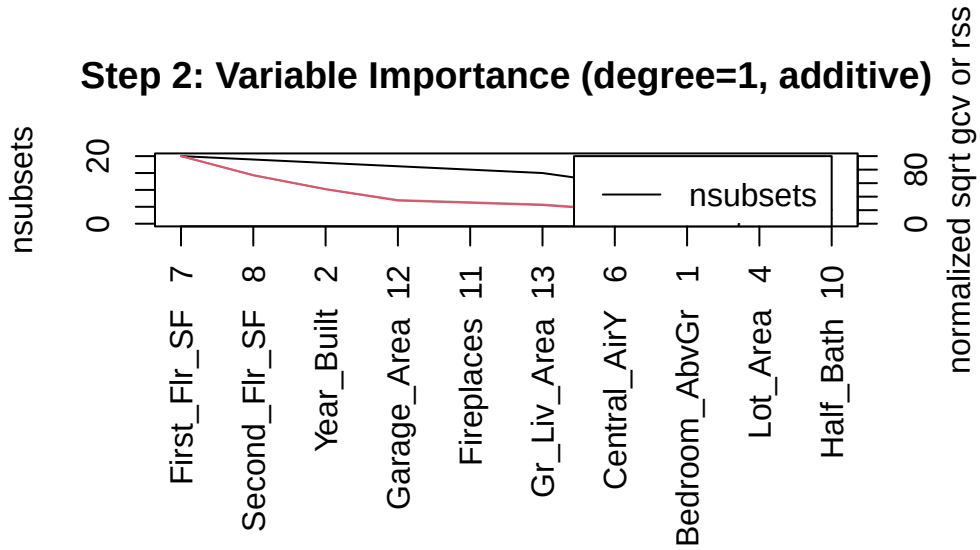
4.4.2 4B. Additive MARS (degree = 1; no interactions)

```
mars2 <- earth(Sale_Price ~ Bedroom_AbvGr + Year_Built + Mo_Sold + Lot_Area +
                 Street + Central_Air + First_Flr_SF + Second_Flr_SF + Full_Bath +
                 Half_Bath + Fireplaces + Garage_Area + Gr_Liv_Area + TotRms_AbvGrd,
               data=train, degree=1, nfold=5)
summary(mars2)
```

```
Call: earth(formula=Sale_Price~Bedroom_AbvGr+Year_Built+Mo_Sold+Lot_Ar...),
            data=train, degree=1, nfold=5)

                      coefficients
(Intercept)             319493.46
Central_AirY             20289.49
h(4-Bedroom_AbvGr)        9214.66
h(Bedroom_AbvGr-4)      -23009.05
```

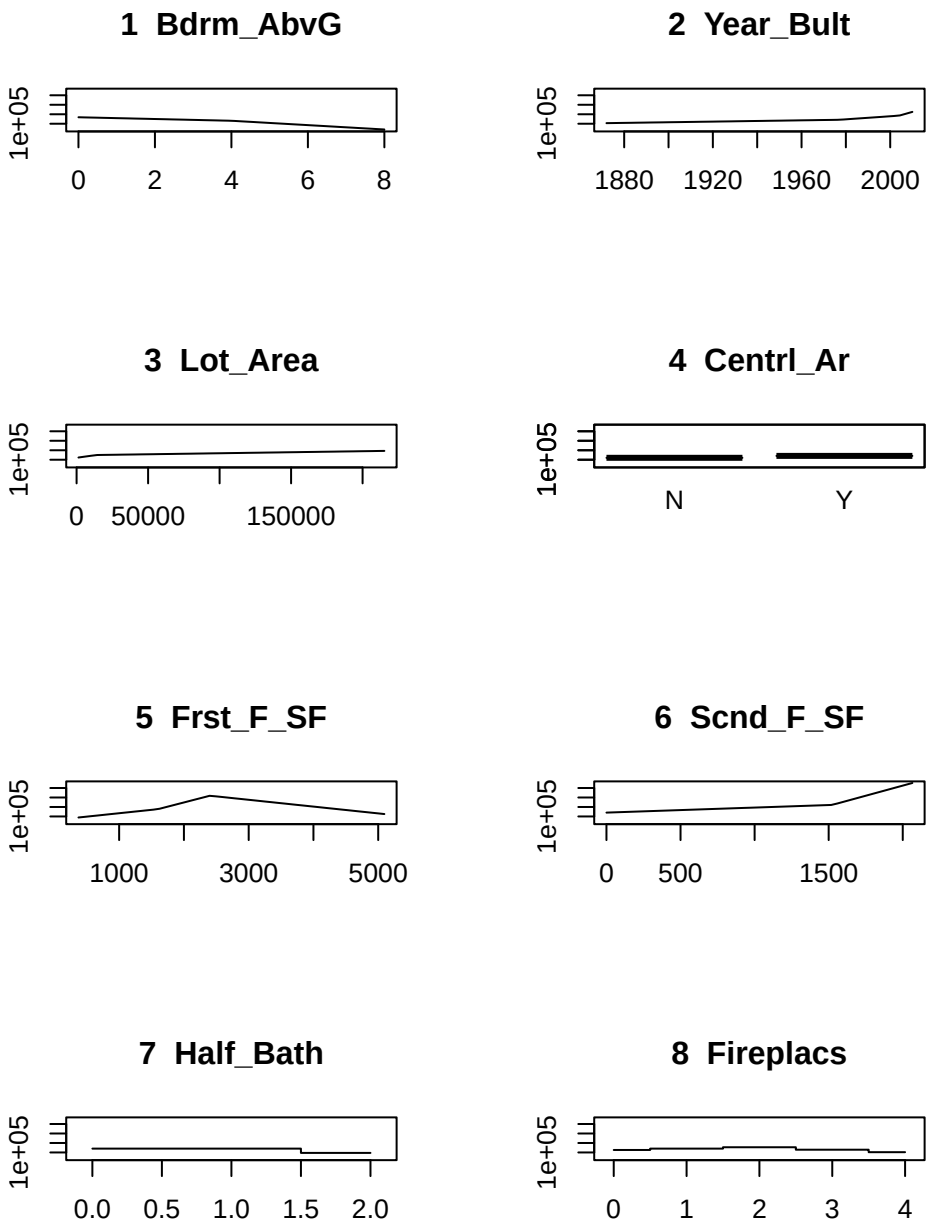
```
h(Year_Built-1977)         1275.57
h(2004-Year_Built)         -336.64
h(Year_Built-2004)         5315.57
h(13869-Lot_Area)            -2.09
h(Lot_Area-13869)             0.22
h(First_Flr_SF-1600)        104.91
h(2402-First_Flr_SF)        -71.56
h(First_Flr_SF-2402)       -176.61
h(1523-Second_Flr_SF)       -53.13
h(Second_Flr_SF-1523)       426.63
h(Half_Bath-1)           -45378.31
h(2-Fireplaces)          -14408.56
h(Fireplaces-2)          -26072.58
h(Garage_Area-539)           101.97
h(Garage_Area-1043)         -294.30
h(Gr_Liv_Area-2049)           65.21
h(Gr_Liv_Area-3194)         -159.79

Selected 21 of 24 terms, and 10 of 14 predictors
Termination condition: Reached nk 29
Importance: First_Flr_SF, Second_Flr_SF, Year_Built, Garage_Area, ...
Number of terms at each degree of interaction: 1 20 (additive model)
GCV 1033819964  RSS 2.036439e+12  GRSq 0.8338641  RSq 0.8402842  CVRSq 0.7978671

Note: the cross-validation sd's below are standard deviations across folds

Cross validation:   nterms 21.00 sd 0.71    nvars 9.20 sd 0.84

     CVRSq    sd     MaxErr     sd
     0.798 0.043    -428941 282429
```

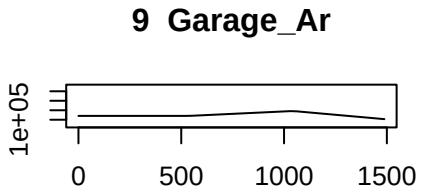
```r
ev2 <- evimp(mars2); plot(ev2, main="Step 2: Variable Importance (degree=1, additive)")
```

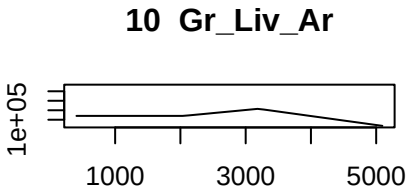



```
par(mfrow=c(2,2)); plotmo(mars2, type="response", nresponse=1, do.par=FALSE); par(mfrow=c(1,1
```

```
plotmo grid:    Bedroom_AbvGr Year_Built Mo_Sold Lot_Area Street Central_Air
                            3       1974       6     9480   Pave           Y
First_Flr_SF Second_Flr_SF Full_Bath Half_Bath Fireplaces Garage_Area
        1092             0         2         0          1         480
Gr_Liv_Area TotRms_AbvGrd
       1442             6
```


1 Bdrm_AbvG
1e+05
0
2
4
6
8
2 Year_Built
1e+05
1880
1920
1960
2000
3 Lot_Area
1e+05
0
50000
150000
4 Centrl_Ar
1e+05
N
Y
5 Frst_F_SF
1e+05
1000
3000
5000
6 Scnd_F_SF
1e+05
0
500
1500
7 Half_Bath
1e+05
0.0
0.5
1.0
1.5
2.0
8 Fireplacs
1e+05
0
1
2
3
4





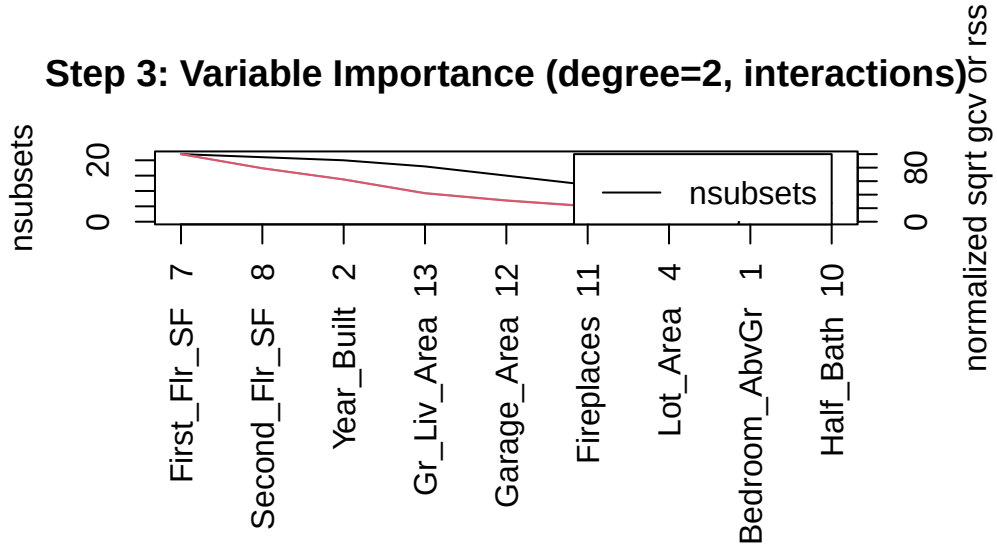
4.4.3 4C. MARS with 2-way Interactions (degree = 2)

```
mars3 <- earth(Sale_Price ~ Bedroom_AbvGr + Year_Built + Mo_Sold + Lot_Area +
                 Street + Central_Air + First_Flr_SF + Second_Flr_SF + Full_Bath +
                 Half_Bath + Fireplaces + Garage_Area + Gr_Liv_Area + TotRms_AbvGrd,
               data=train, degree=2, nfold=5)
summary(mars3)
```

```
Call: earth(formula=Sale_Price~Bedroom_AbvGr+Year_Built+Mo_Sold+Lot_Ar...),
            data=train, degree=2, nfold=5)

                                        coefficients
(Intercept)                               411949.27
h(Year_Built-2004)                        169869.37
h(14803-Lot_Area)                             -1.75
h(Lot_Area-14803)                              0.48
h(1570-First_Flr_SF)                        -150.79
h(First_Flr_SF-1570)                         224.90
h(1523-Second_Flr_SF)                        -86.91
h(Second_Flr_SF-1523)                        156.02
h(1-Half_Bath)                             -7743.19
```

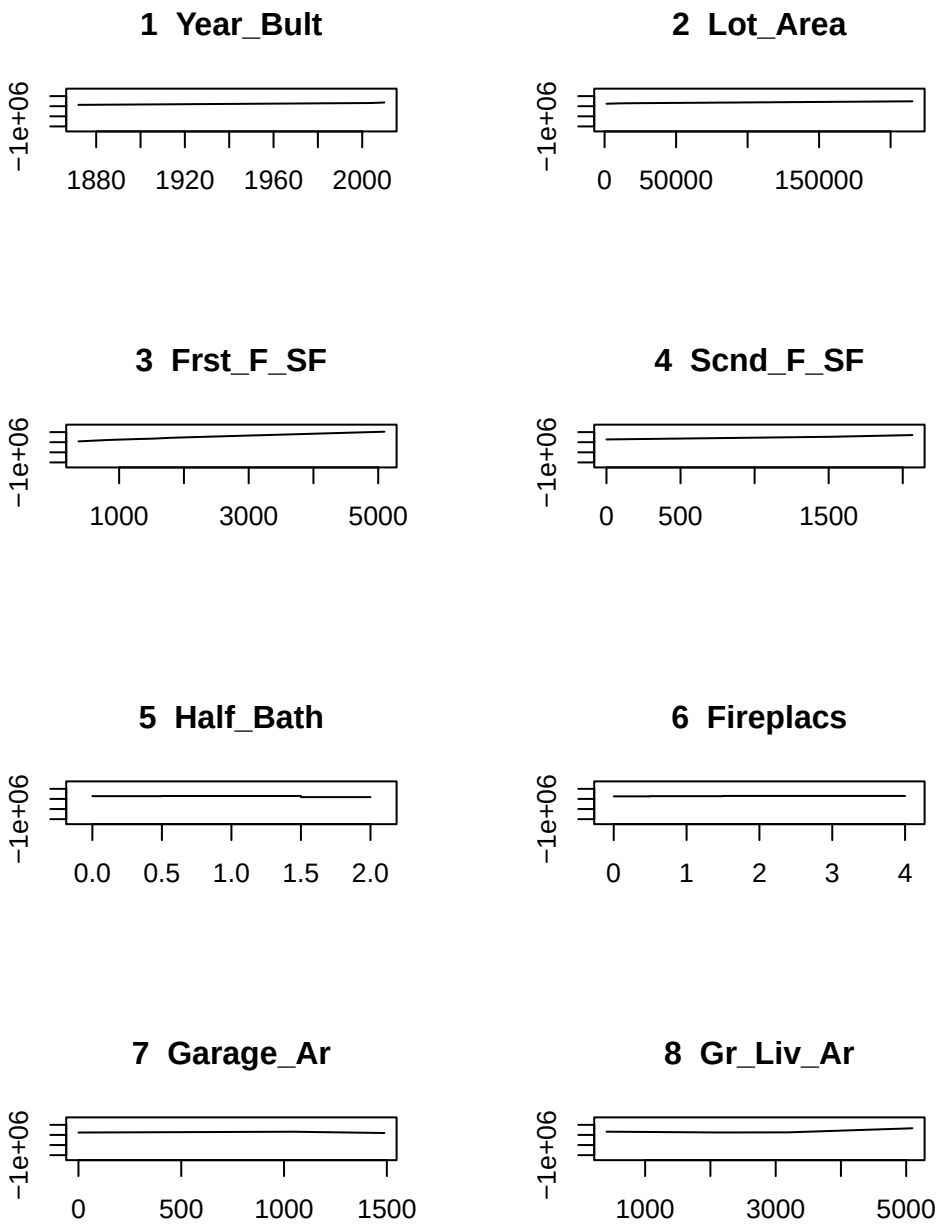
```
h(Half_Bath-1)                                  -54819.24
h(2-Fireplaces)                                 -10516.92
h(1043-Garage_Area)                                -70.97
h(Garage_Area-1043)                               -141.86
h(Gr_Liv_Area-3194)                                106.27
h(3-Bedroom_AbvGr) * h(1523-Second_Flr_SF)           7.01
h(Bedroom_AbvGr-3) * h(1523-Second_Flr_SF)          -9.61
h(2004-Year_Built) * h(First_Flr_SF-876)            -1.86
h(2004-Year_Built) * h(3194-Gr_Liv_Area)            -0.17
h(Year_Built-2004) * h(Gr_Liv_Area-2320)          -159.80
h(Year_Built-2004) * h(2320-Gr_Liv_Area)           156.64
h(Year_Built-2004) * h(3194-Gr_Liv_Area)          -173.05
h(First_Flr_SF-1778) * h(2-Fireplaces)             -78.19
h(1043-Garage_Area) * h(2122-Gr_Liv_Area)            0.05

Selected 23 of 29 terms, and 9 of 14 predictors
Termination condition: Reached nk 29
Importance: First_Flr_SF, Second_Flr_SF, Year_Built, Gr_Liv_Area, ...
Number of terms at each degree of interaction: 1 13 9
GCV 795873462  RSS 1.544416e+12  GRSq 0.8721024  RSq 0.8788731  CVRSq 0.7824362

Note: the cross-validation sd's below are standard deviations across folds

Cross validation:   nterms 22.60 sd 1.82    nvars 9.40 sd 1.14

     CVRSq    sd     MaxErr     sd
     0.782 0.114    -945502 422348
```

```r
ev3 <- evimp(mars3); plot(ev3, main="Step 3: Variable Importance (degree=2, interactions)")
```

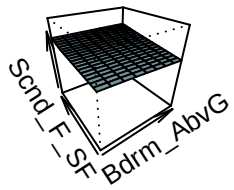



```
par(mfrow=c(2,2)); plotmo(mars3, type="response", nresponse=1, do.par=FALSE); par(mfrow=c(1,1
```

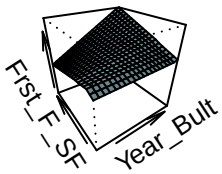
```
plotmo grid:    Bedroom_AbvGr Year_Built Mo_Sold Lot_Area Street Central_Air
                            3       1974       6     9480   Pave           Y
First_Flr_SF Second_Flr_SF Full_Bath Half_Bath Fireplaces Garage_Area
        1092             0         2         0          1         480
Gr_Liv_Area TotRms_AbvGrd
       1442             6
```


1 Year_Bult
−1e+06
1880
1920
1960
2000
2 Lot_Area
−1e+06
0
50000
150000
3 Frst_F_SF
−1e+06
1000
3000
5000
4 Scnd_F_SF
−1e+06
0
500
1500
5 Half_Bath
−1e+06
0.0
0.5
1.0
1.5
2.0
6 Fireplacs
−1e+06
0
1
2
3
4
7 Garage_Ar
−1e+06
0
500
1000
1500
8 Gr_Liv_Ar
−1e+06
1000
3000
5000

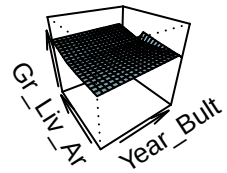
1 Bdrm_AbvG: Scnd_F_SF



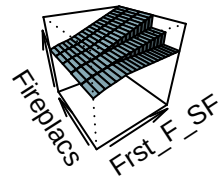
2 Year_Bult: Frst_F_SF



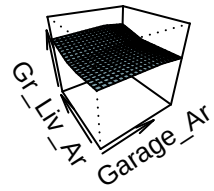
3 Year_Bult: Gr_Liv_Ar



4 Frst_F_SF: Fireplacs



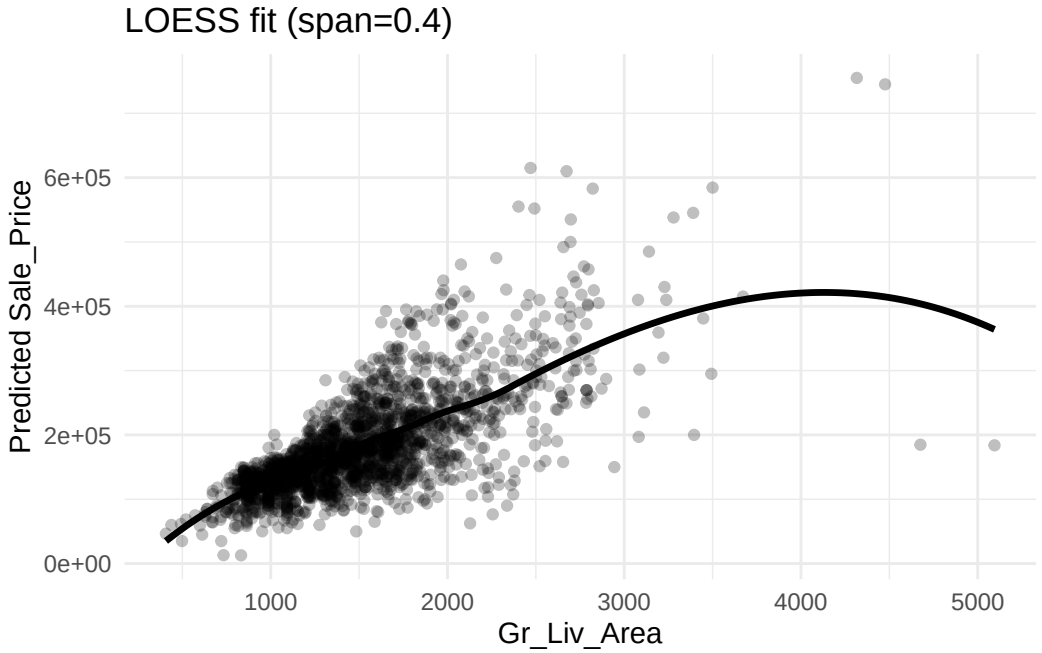
5 Garage_Ar: Gr_Liv_Ar



Interpreting MARS terms:
Look for `h()` pieces and products. Coefficients on `h(x-c)` indicate how slope changes after the knot `c`. Interaction products mean "the slope change depends on another variable".

4.5 5. LOESS (Visual local regression)

```
loess_fit <- loess(Sale_Price ~ Gr_Liv_Area, data=train, span=0.4)
grid_lo <- tibble(Gr_Liv_Area = seq(min(train$Gr_Liv_Area), max(train$Gr_Liv_Area), length.ou
grid_lo$pred <- predict(loess_fit, newdata=grid_lo)

ggplot() +
  geom_point(data=train, aes(Gr_Liv_Area, Sale_Price), alpha=.25) +
  geom_line(data=grid_lo, aes(Gr_Liv_Area, pred), linewidth=1.2) +
  labs(title="LOESS fit (span=0.4)", y="Predicted Sale_Price")
```



We keep LOESS as a visualization-oriented smoother (not used in the multivariate comparison below).

4.6 6. GAM Splines with `mgcv` – Simple → Complex

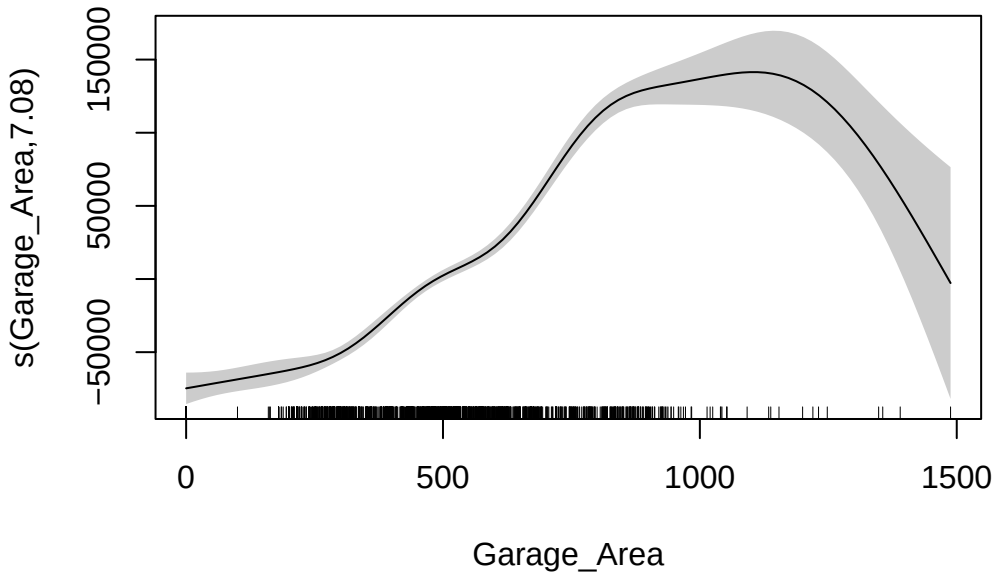
GAM smooths are written `s(x)` and estimated with penalized splines. The **edf** (effective degrees of freedom) in the summary tells you the smooth's complexity (larger edf wigglier function).

4.6.1 6A. Univariate GAM: `Sale_Price ~ s(Garage_Area)`

```
gam1 <- gam(Sale_Price ~ s(Garage_Area, k=9), data=train, method="REML")
summary(gam1)  # inspect edf and significance
```

```
Family: gaussian
Link function: identity

Formula:
Sale_Price ~ s(Garage_Area, k = 9)

Parametric coefficients:
            Estimate Std. Error t value Pr(>|t|)
(Intercept)   180897       1292     140   <2e-16 ***
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

Approximate significance of smooth terms:
                 edf Ref.df     F p-value
s(Garage_Area) 7.078  7.719 217.1  <2e-16 ***
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

R-sq.(adj) =   0.45   Deviance explained = 45.2%
-REML =  25418  Scale est. = 3.4232e+09  n = 2051
```

```
plot(gam1, pages=1, shade=TRUE, scheme=1, scale=0)
```



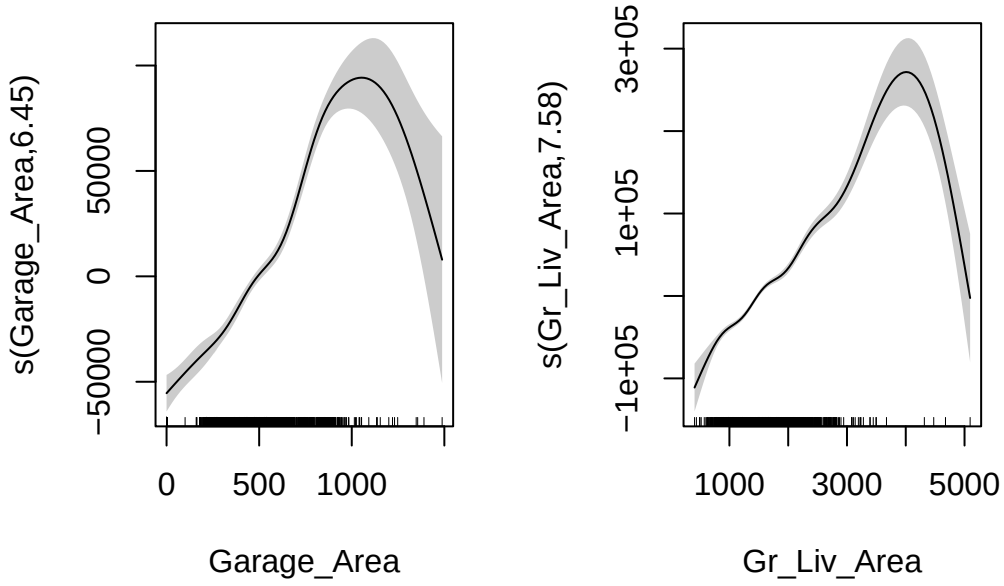
4.6.2 6B. Two-smooth Additive GAM: `s(Garage_Area) + s(Gr_Liv_Area)`

```
gam2 <- gam(Sale_Price ~ s(Garage_Area, k=9) + s(Gr_Liv_Area, k=9), data=train, method="REML"
summary(gam2)
```

```
Family: gaussian
Link function: identity

Formula:
Sale_Price ~ s(Garage_Area, k = 9) + s(Gr_Liv_Area, k = 9)

Parametric coefficients:
            Estimate Std. Error t value Pr(>|t|)
(Intercept)   180897       1026   176.2   <2e-16 ***
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

Approximate significance of smooth terms:
                 edf Ref.df     F p-value
s(Garage_Area) 6.446  7.293 103.3  <2e-16 ***
```

```
s(Gr_Liv_Area) 7.585  7.941 152.6  <2e-16 ***
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

R-sq.(adj) =  0.653   Deviance explained = 65.5%
-REML =  24949  Scale est. = 2.1608e+09  n = 2051
```

```r
plot(gam2, pages=1, shade=TRUE, scheme=1, scale=0)
```



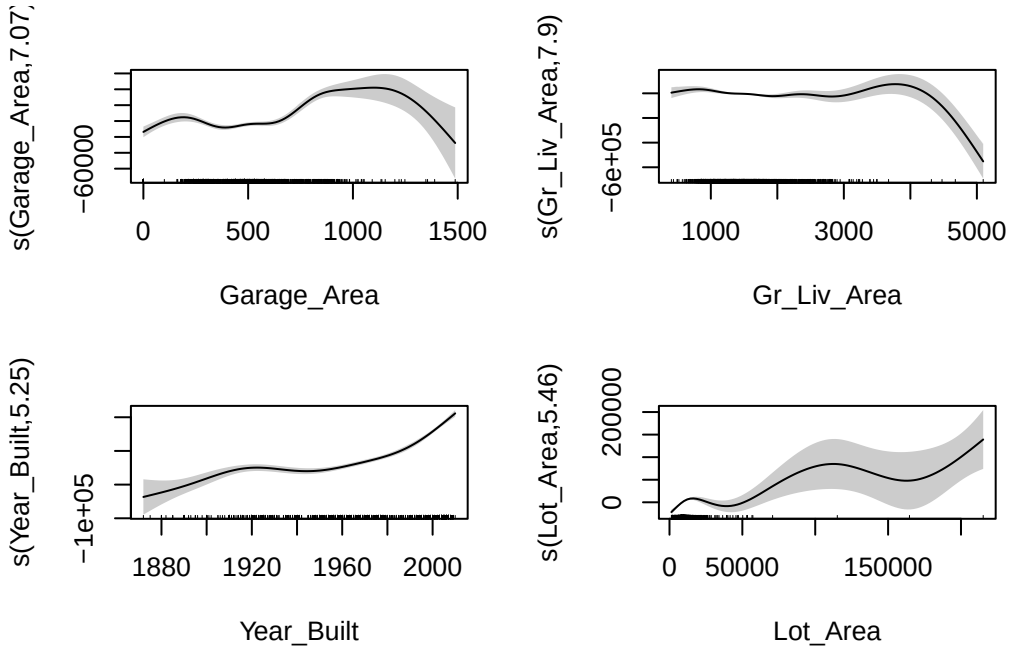
4.6.3 6C. Full Multivariate GAM (selected smooths + factors)

```r
gam3 <- gam(Sale_Price ~
              s(Garage_Area, k=9) +
              s(Gr_Liv_Area, k=9) +
              s(Year_Built, k=7) +
              s(Lot_Area, k=7) +
              Bedroom_AbvGr + Mo_Sold +
              Street + Central_Air +
              First_Flr_SF + Second_Flr_SF +
              Full_Bath + Half_Bath +
```

```
              Fireplaces + TotRms_AbvGrd,
            data=train, method="REML")
summary(gam3)
```

```
Family: gaussian
Link function: identity

Formula:
Sale_Price ~ s(Garage_Area, k = 9) + s(Gr_Liv_Area, k = 9) +
    s(Year_Built, k = 7) + s(Lot_Area, k = 7) + Bedroom_AbvGr +
    Mo_Sold + Street + Central_Air + First_Flr_SF + Second_Flr_SF +
    Full_Bath + Half_Bath + Fireplaces + TotRms_AbvGrd

Parametric coefficients:
               Estimate Std. Error t value Pr(>|t|)
(Intercept)   -21237.92   30106.84  -0.705   0.4806
Bedroom_AbvGr -10350.20    1382.38  -7.487 1.05e-13 ***
Mo_Sold         -409.12     270.24  -1.514   0.1302
StreetPave     30589.21   13486.61   2.268   0.0234 *
Central_AirY   17189.68    3366.03   5.107 3.59e-07 ***
First_Flr_SF     140.27      17.44   8.044 1.47e-15 ***
Second_Flr_SF    101.93      17.36   5.871 5.06e-09 ***
Full_Bath      -4548.94    2211.69  -2.057   0.0398 *
Half_Bath       2355.98    2175.40   1.083   0.2789
Fireplaces     13066.61    1371.59   9.527  < 2e-16 ***
TotRms_AbvGrd  -1640.87     948.67  -1.730   0.0838 .
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

Approximate significance of smooth terms:
                edf Ref.df      F p-value
s(Garage_Area) 7.067  7.712  24.96  <2e-16 ***
s(Gr_Liv_Area) 7.899  7.996  46.44  <2e-16 ***
s(Year_Built)  5.255  5.764 128.05  <2e-16 ***
s(Lot_Area)    5.465  5.883  16.44  <2e-16 ***
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

R-sq.(adj) =  0.827   Deviance explained =   83%
-REML =  24167  Scale est. = 1.0786e+09  n = 2051
```

```r
plot(gam3, pages=1, shade=TRUE, scheme=1, scale=0)
```



Interpreting GAM splines:
A smooth `s(x)` shows how the expected outcome varies with `x` *holding other terms constant.* The edf indicates complexity; wide confidence bands suggest greater uncertainty in those regions.

4.7 7. Model Comparison (R^2, RMSE, MSE)

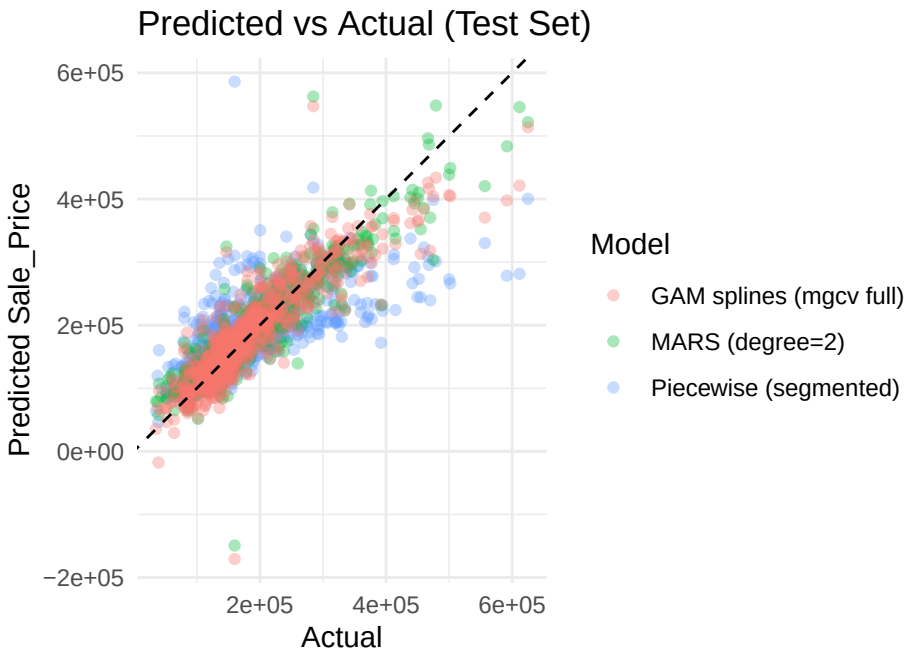
```r
# Helper for R^2 on test
rsq <- function(y, yhat) {
  1 - sum((y - yhat)^2) / sum((y - mean(y))^2)
}

# Predictions
p_piece <- predict(m_seg, newdata=test)
p_mars  <- predict(mars3, newdata=test)   # final MARS with interactions
p_gam   <- predict(gam3, newdata=test)    # full multivariate GAM
```

```r
# Metrics
tbl <- tibble(
  Model = c("Piecewise (segmented)", "MARS (degree=2)", "GAM splines (mgcv full)"),
  RMSE  = c(rmse(test$Sale_Price, p_piece),
            rmse(test$Sale_Price, p_mars),
            rmse(test$Sale_Price, p_gam)),
  MSE   = c(mse(test$Sale_Price, p_piece),
            mse(test$Sale_Price, p_mars),
            mse(test$Sale_Price, p_gam)),
  R2    = c(rsq(test$Sale_Price, p_piece),
            rsq(test$Sale_Price, p_mars),
            rsq(test$Sale_Price, p_gam))
) |> arrange(desc(R2))

tbl
```

```
# A tibble: 3 x 4
  Model                     RMSE         MSE    R2
  <chr>                    <dbl>       <dbl> <dbl>
1 MARS (degree=2)         33028. 1090840972. 0.839
2 GAM splines (mgcv full) 36796. 1353918715. 0.800
3 Piecewise (segmented)   59034. 3484963067. 0.484
```

4.8 8. Final Plot: Predicted vs Actual (Test Set)

```r
plot_df <- bind_rows(
  tibble(Model="Piecewise (segmented)", Actual=test$Sale_Price, Pred=p_piece),
  tibble(Model="MARS (degree=2)", Actual=test$Sale_Price, Pred=p_mars),
  tibble(Model="GAM splines (mgcv full)", Actual=test$Sale_Price, Pred=p_gam)
)

ggplot(plot_df, aes(Actual, Pred, color=Model)) +
  geom_point(alpha=.35) +
  geom_abline(intercept=0, slope=1, linetype="dashed") +
  labs(title="Predicted vs Actual (Test Set)", y="Predicted Sale_Price") +
  coord_equal()
```



4.9 9. How to Explain

- **Piecewise**: "There's a threshold where the effect changes." Simple story, limited flexibility.
- **MARS**: "The model finds breakpoints and sometimes *when* they matter (interactions)." Explain `h(x-c)` as *extra slope after c*.
- **GAM (splines)**: "We model smooth curves for key variables; edf shows wiggliness. More edf more flexibility."
- **Tradeoff**: More flexibility usually improves error (RMSE/MSE) and R^2, but reduce interpretability. Validate with cross-validation and keep the story aligned with business intuition.

5 GAM— Piecewise, LOESS, and GAM Splines - Python Version

Note: The original MARS section used the `sklearn-contrib-py-earth` package, which is **no longer actively maintained** and may fail to install on modern Python versions.
The MARS code is left **commented out** for reference — you can un-comment it if you successfully install `py-earth` (try `conda install -c conda-forge sklearn-contrib-py-earth`).
Equivalent flexibility can be achieved through **GAM** or **spline-based models**, which are shown below.

5.1 1. Setup and Data

```
import sys
print("Active Python interpreter:", sys.executable)

import numpy as np, pandas as pd
import matplotlib.pyplot as plt

from sklearn.model_selection import train_test_split
from sklearn.compose import ColumnTransformer
from sklearn.preprocessing import OneHotEncoder, StandardScaler
from sklearn.pipeline import Pipeline
from sklearn.metrics import mean_squared_error, r2_score
from sklearn.linear_model import LinearRegression, Ridge
import inspect

# --- Fix for SciPy >= 1.13 removing .A from sparse matrices ---
import scipy.sparse as sp

if not hasattr(sp.spmatrix, "A"):
    def _toarray(self):
```

```
        return self.toarray()
    sp.spmatrix.A = property(_toarray)
# -----------------------------------------------------------

np.random.seed(4321)
```

```
Active Python interpreter: /opt/anaconda3/bin/python
```

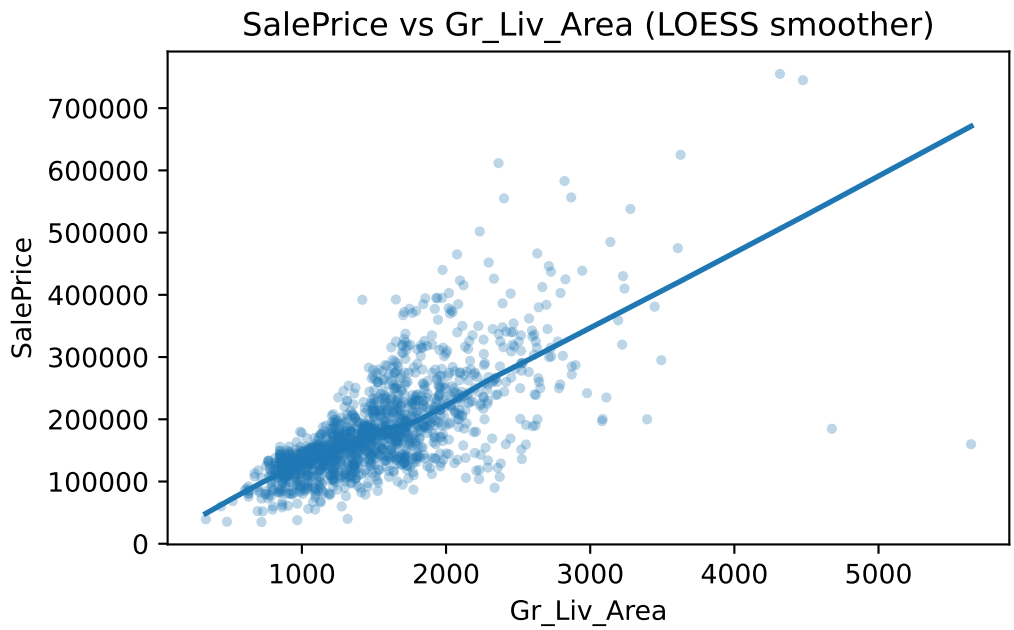
```
from sklearn.datasets import fetch_openml

ames = fetch_openml(name="house_prices", as_frame=True).frame

keep = {
    "SalePrice":"SalePrice",
    "BedroomAbvGr":"Bedroom_AbvGr",
    "YearBuilt":"Year_Built",
    "MoSold":"Mo_Sold",
    "LotArea":"Lot_Area",
    "Street":"Street",
    "CentralAir":"Central_Air",
    "1stFlrSF":"First_Flr_SF",
    "2ndFlrSF":"Second_Flr_SF",
    "FullBath":"Full_Bath",
    "HalfBath":"Half_Bath",
    "Fireplaces":"Fireplaces",
    "GarageArea":"Garage_Area",
    "GrLivArea":"Gr_Liv_Area",
    "TotRmsAbvGrd":"TotRms_AbvGrd"
}

df = ames[list(keep.keys())].rename(columns=keep).dropna().copy()

X = df.drop(columns=["SalePrice"])
y = df["SalePrice"].values

num_cols = X.select_dtypes(include=[np.number]).columns.tolist()
cat_cols = [c for c in X.columns if c not in num_cols]

# Handle OneHotEncoder version changes and ensure dense output
if "sparse_output" in inspect.signature(OneHotEncoder).parameters:
    encoder = OneHotEncoder(drop="first", handle_unknown="ignore", sparse_output=False)
```

```
else:
    encoder = OneHotEncoder(drop="first", handle_unknown="ignore", sparse=False)

pre = ColumnTransformer([
    ("num", StandardScaler(), num_cols),
    ("cat", encoder, cat_cols)
], sparse_threshold=0)

X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.3, random_state=4321)
df.head()
```

	SalePrice	Bedroom_AbvGr	Year_Built	Mo_Sold	Lot_Area	Street	Central_Air	First_Flr_SF
0	208500	3	2003	2	8450	Pave	Y	856
1	181500	3	1976	5	9600	Pave	Y	1262
2	223500	3	2001	9	11250	Pave	Y	920
3	140000	3	1915	2	9550	Pave	Y	961
4	250000	4	2000	12	14260	Pave	Y	1145

5.2 2. Visual Check for Nonlinearity

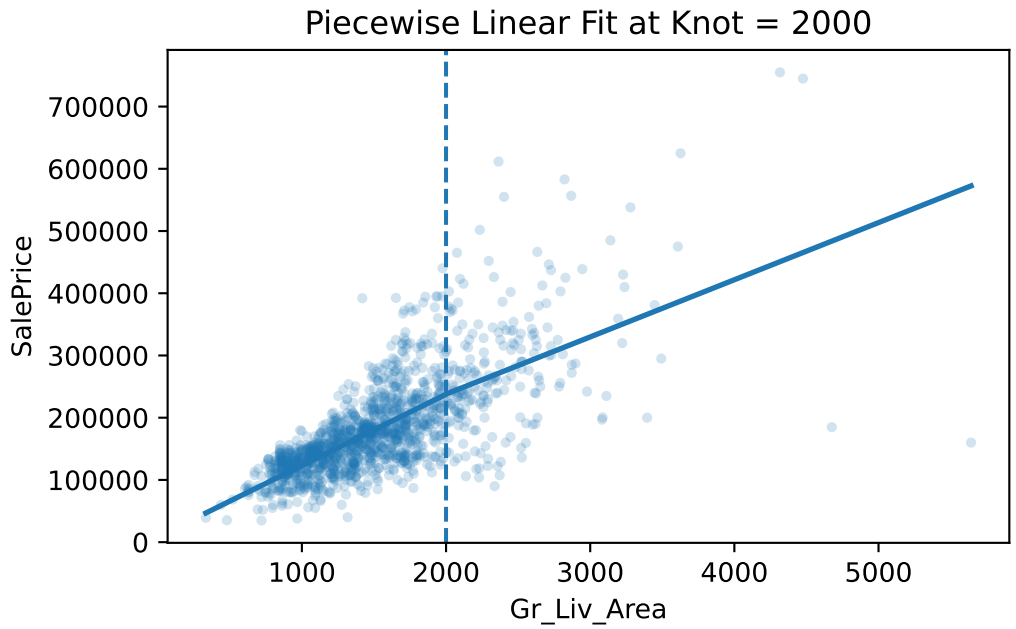
```
import statsmodels.api as sm
fig = plt.figure()
plt.scatter(df["Gr_Liv_Area"], df["SalePrice"], s=8, alpha=0.3)
low = sm.nonparametric.lowess(df["SalePrice"], df["Gr_Liv_Area"], frac=0.3, return_sorted=Tru
plt.plot(low[:,0], low[:,1], linewidth=2)
plt.title("SalePrice vs Gr_Liv_Area (LOESS smoother)")
plt.xlabel("Gr_Liv_Area"); plt.ylabel("SalePrice")
plt.tight_layout()
```



5.3 3. Piecewise Regression Example

```
class PiecewiseLinear:
    def __init__(self, knot=2000.0):
        self.knot = knot
        self.lin = LinearRegression()

    def _transform(self, x):
        x1 = np.asarray(x).reshape(-1,1)
        hx = np.maximum(0, x1 - self.knot)
        return np.hstack([x1, hx])

    def fit(self, x, y):
        Z = self._transform(x)
        self.lin.fit(Z, y)
        return self

    def predict(self, x):
        Z = self._transform(x)
```

```
        return self.lin.predict(Z)

pw = PiecewiseLinear(knot=2000).fit(df["Gr_Liv_Area"].values, df["SalePrice"].values)

grid = np.linspace(df["Gr_Liv_Area"].min(), df["Gr_Liv_Area"].max(), 200)
pred = pw.predict(grid)

plt.figure()
plt.scatter(df["Gr_Liv_Area"], df["SalePrice"], s=8, alpha=0.2)
plt.plot(grid, pred, linewidth=2)
plt.axvline(2000, linestyle="--")
plt.title("Piecewise Linear Fit at Knot = 2000")
plt.xlabel("Gr_Liv_Area"); plt.ylabel("SalePrice")
plt.tight_layout()
```

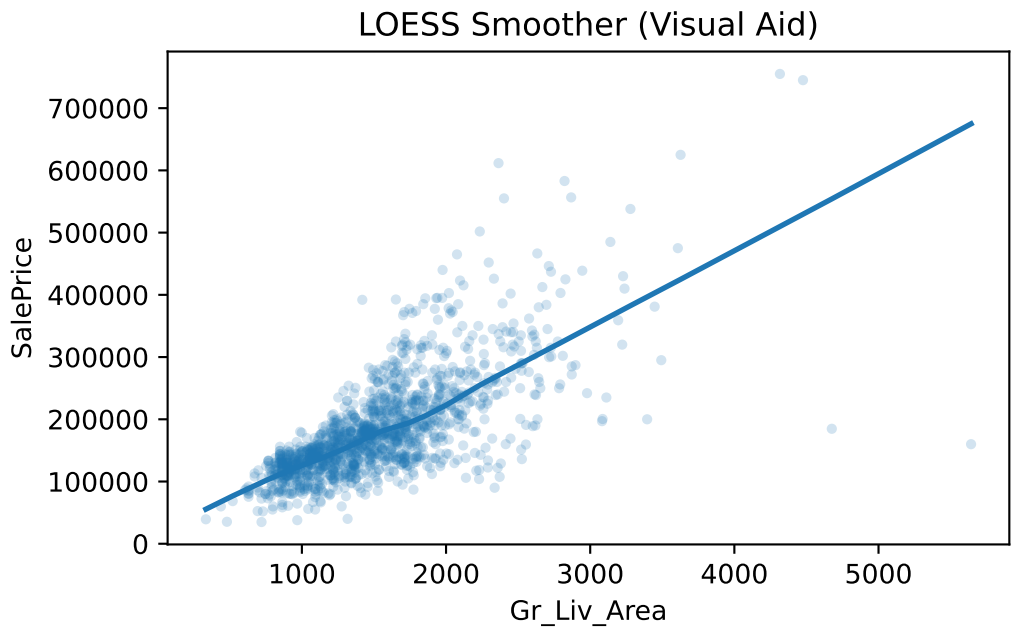


5.4 4. MARS (Commented Out — For Reference Only)

```
# The MARS implementation (py-earth) is not actively maintained and may fail to install on mo
# If you wish to try it, install from conda-forge:
#     conda install -c conda-forge sklearn-contrib-py-earth
#
# Example (commented out):
#
# from pyearth import Earth
#
# mars = Pipeline([("prep", pre), ("model", Earth(max_degree=2))]).fit(X_train, y_train)
# print(mars.named_steps["model"].summary())
#
# xs = np.linspace(X["Garage_Area"].min(), X["Garage_Area"].max(), 200)
# yhat = mars.predict(pd.DataFrame({"Garage_Area": xs}))
#
# plt.figure()
# plt.scatter(X_train["Garage_Area"], y_train, s=8, alpha=0.2)
# plt.plot(xs, yhat, linewidth=2)
# plt.title("MARS with Garage_Area (Commented Out Example)")
# plt.xlabel("Garage_Area"); plt.ylabel("SalePrice")
# plt.tight_layout()
```

5.5 5. LOESS Visualization

```
fig = plt.figure()
plt.scatter(df["Gr_Liv_Area"], df["SalePrice"], s=8, alpha=0.2)
low = sm.nonparametric.lowess(df["SalePrice"], df["Gr_Liv_Area"], frac=0.4, return_sorted=Tru
plt.plot(low[:,0], low[:,1], linewidth=2)
plt.title("LOESS Smoother (Visual Aid)")
plt.xlabel("Gr_Liv_Area"); plt.ylabel("SalePrice")
plt.tight_layout()
```



5.6 6. GAM and Spline Approaches (Modern Alternatives)

```
from pygam import LinearGAM, s
gam1 = LinearGAM(s(0)).fit(X_train[["Garage_Area"]].values, y_train)
print(gam1.summary())
gam2 = LinearGAM(s(0) + s(1)).fit(X_train[["Garage_Area","Gr_Liv_Area"]].values, y_train)
print(gam2.summary())
num_train = X_train[num_cols].values
gam3 = LinearGAM().fit(num_train, y_train)
print(gam3.summary())
```

```
LinearGAM
============================================= ==========================================
Distribution:                        NormalDist Effective DoF:
Link Function:                     IdentityLink Log Likelihood:
23263.1874
Number of Samples:                         1022 AIC:
                                                AICc:
```

```
                                              GCV:                                3
                                              Scale:                              3
                                              Pseudo R-Squared:
=============================================================================================
Feature Function                  Lambda              Rank         EDoF         P > x
================================= ================== ============ ============ ============
s(0)                              [0.6]               20           11.7         1.11e-
16     ***
intercept                                             1            0.0          1.11e-
16     ***
=============================================================================================
Significance codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

WARNING: Fitting splines and a linear function to a feature introduces a model identifiabilit
         which can cause p-values to appear significant when they are not.

WARNING: p-values calculated in this manner behave correctly for un-penalized models or model
         known smoothing parameters, but when smoothing parameters have been estimated, the p
values
         are typically lower than they should be, meaning that the tests reject the null too
None
LinearGAM
============================================= ==========================================
Distribution:                        NormalDist Effective DoF:
Link Function:                     IdentityLink Log Likelihood:
22770.8323
Number of Samples:                         1022 AIC:
                                                AICc:
                                                GCV:                               19
                                                Scale:                              1
                                                Pseudo R-Squared:
=============================================================================================
Feature Function                  Lambda              Rank         EDoF         P > x
================================= ================== ============ ============ ============
s(0)                              [0.6]               20           12.6         1.11e-
16     ***
s(1)                              [0.6]               20           8.3          1.11e-
16     ***
intercept                                             1            0.0          1.11e-
16     ***
=============================================================================================
Significance codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
```

```
WARNING: Fitting splines and a linear function to a feature introduces a model identifiabilit
         which can cause p-values to appear significant when they are not.

WARNING: p-values calculated in this manner behave correctly for un-penalized models or model
         known smoothing parameters, but when smoothing parameters have been estimated, the p
values
         are typically lower than they should be, meaning that the tests reject the null too
None
LinearGAM
============================================= ==========================================
Distribution:                        NormalDist Effective DoF:
Link Function:                     IdentityLink Log Likelihood:
21995.395
Number of Samples:                         1022 AIC:
                                                AICc:
                                                GCV:                                    10
                                                Scale:                                   8
                                                Pseudo R-Squared:
=================================================================================================
Feature Function                  Lambda               Rank         EDoF         P > x
================================= ==================== ============ ============ ============
s(0)                              [0.6]                20           8.7          6.58e-
07     ***
s(1)                              [0.6]                20           12.8         1.11e-
16     ***
s(2)                              [0.6]                20           11.0         5.15e-
01
s(3)                              [0.6]                20           8.4          1.11e-
16     ***
s(4)                              [0.6]                20           8.6          1.11e-
16     ***
s(5)                              [0.6]                20           10.1         2.17e-
03     **
s(6)                              [0.6]                20           3.2          9.76e-
02     .
s(7)                              [0.6]                20           2.2          4.19e-
01
s(8)                              [0.6]                20           2.9          3.16e-
10     ***
s(9)                              [0.6]                20           9.0          1.11e-
16     ***
s(10)                             [0.6]                20           5.7          1.11e-
16     ***
```

```
s(11)                              [0.6]                20           6.5          3.78e-
01
intercept                                               1            0.0          1.11e-
16     ***
================================================================================================
Significance codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

WARNING: Fitting splines and a linear function to a feature introduces a model identifiabilit
         which can cause p-values to appear significant when they are not.

WARNING: p-values calculated in this manner behave correctly for un-penalized models or model
         known smoothing parameters, but when smoothing parameters have been estimated, the p
values
         are typically lower than they should be, meaning that the tests reject the null too
None
```

5.7 7. Regression Spline Example (Fallback)

```
from patsy import dmatrix

design_tr = dmatrix("bs(Garage_Area, df=6) + bs(Gr_Liv_Area, df=6)", data=X_train, return_typ
design_te = dmatrix("bs(Garage_Area, df=6) + bs(Gr_Liv_Area, df=6)", data=X_test, return_type

ridge = Ridge(alpha=1.0).fit(design_tr, y_train)
pred_spline = ridge.predict(design_te)
```

5.8 8. Model Comparison (Piecewise vs GAM vs Spline)

```
def rmse(y, yhat):
    try:
        return mean_squared_error(y, yhat, squared=False)
    except TypeError:
        return np.sqrt(mean_squared_error(y, yhat))
```

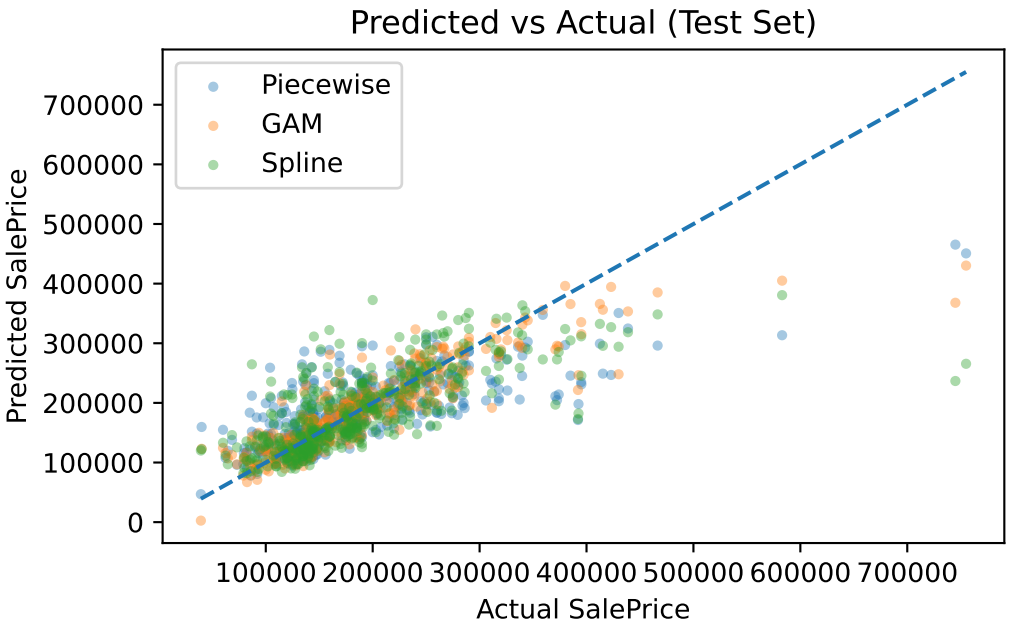
```
def mse(y, yhat): return mean_squared_error(y, yhat)
def rsq(y, yhat): return r2_score(y, yhat)

p_piece = pw.predict(X_test["Gr_Liv_Area"].values)

try:
    p_gam = gam3.predict(X_test[num_cols].values)
except Exception:
    p_gam = np.full_like(y_test, np.nan, dtype=float)

p_spline = pred_spline

cmp = pd.DataFrame({
    "Model": ["Piecewise", "GAM (pyGAM)" if not np.isnan(p_gam).all() else "GAM (not availabl
    "RMSE": [rmse(y_test, p_piece), rmse(y_test, p_gam) if not np.isnan(p_gam).all() else np
    "MSE": [mse(y_test, p_piece), mse(y_test, p_gam) if not np.isnan(p_gam).all() else np.nan
    "R2": [rsq(y_test, p_piece), rsq(y_test, p_gam) if not np.isnan(p_gam).all() else np.nan
}).sort_values("R2", ascending=False)

cmp
```

	Model	RMSE	MSE	R2
1	GAM (pyGAM)	40805.867162	1.665119e+09	0.766556
0	Piecewise	58108.537553	3.376602e+09	0.526612
2	Regression Spline	60289.000524	3.634764e+09	0.490419

5.9 9. Predicted vs Actual Comparison

```
plt.figure()
plt.scatter(y_test, p_piece, s=8, alpha=0.4, label="Piecewise")
if not np.isnan(p_gam).all():
    plt.scatter(y_test, p_gam, s=8, alpha=0.4, label="GAM")
plt.scatter(y_test, p_spline, s=8, alpha=0.4, label="Spline")
plt.plot([y_test.min(), y_test.max()], [y_test.min(), y_test.max()], linestyle="--")
plt.legend()
```

```
plt.title("Predicted vs Actual (Test Set)")
plt.xlabel("Actual SalePrice"); plt.ylabel("Predicted SalePrice")
plt.tight_layout()
```



5.10 10. Interpretation Summary

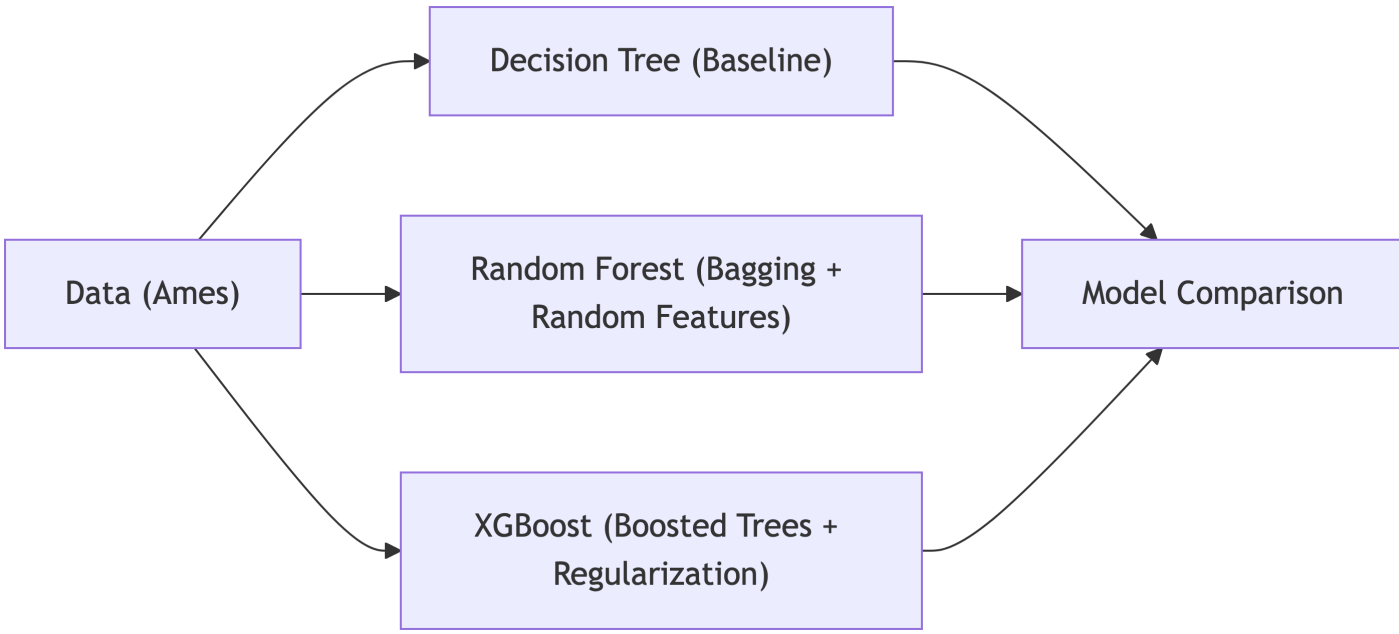
- **Piecewise:** adds simple thresholds, easy to explain.
- **GAMs:** provide smooth, interpretable nonlinearities.
- **Splines:** flexible approximations that behave like local polynomials.
- **MARS (optional):** similar conceptually but less maintained in Python; consider using R's `earth` instead.

6 Tree Based Models - R Version

Comparison: **Decision Tree**, **Random Forest**, and **XGBoost** on the Ames dataset. Includes random-variable baselines, variable importance, partial dependence plots.

6.1 Visual Roadmap

```
flowchart LR
    A["Data (Ames)"]
    DT["Decision Tree (Baseline)"]
    RF["Random Forest (Bagging + Random Features)"]
    XGB["XGBoost (Boosted Trees + Regularization)"]
    Compare["Model Comparison"]

    A --> DT
    A --> RF
    A --> XGB
    DT --> Compare
    RF --> Compare
    XGB --> Compare
```



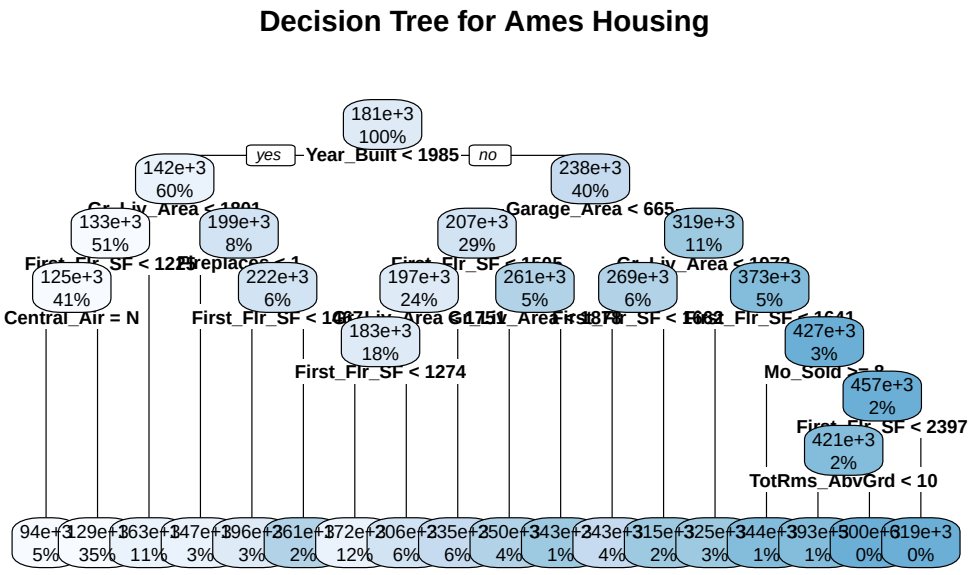
6.2 Libraries, Data, Split

```
library(tidyverse)
library(caret)
library(rpart)
library(rpart.plot)
library(randomForest)
library(xgboost)
library(Ckmeans.1d.dp)
library(pdp)
library(AmesHousing)

ames <- make_ordinal_ames()
ames <- ames %>% mutate(id = row_number())

set.seed(4321)
training <- ames %>% sample_frac(0.7)
testing <- anti_join(ames, training, by = 'id')

training <- training %>%
  select(Sale_Price,
```

```
        Bedroom_AbvGr, Year_Built, Mo_Sold, Lot_Area, Street, Central_Air,
        First_Flr_SF, Second_Flr_SF, Full_Bath, Half_Bath, Fireplaces,
        Garage_Area, Gr_Liv_Area, TotRms_AbvGrd)

training.df <- as.data.frame(training)
```

6.3 Decision Tree (Baseline)

```
set.seed(12345)
tree.ames <- rpart(Sale_Price ~ ., data = training.df, method = "anova",
                   control = rpart.control(cp = 0.005))

rpart.plot(tree.ames, type = 2, fallen.leaves = TRUE, cex = 0.6,
           main = "Decision Tree for Ames Housing")
```



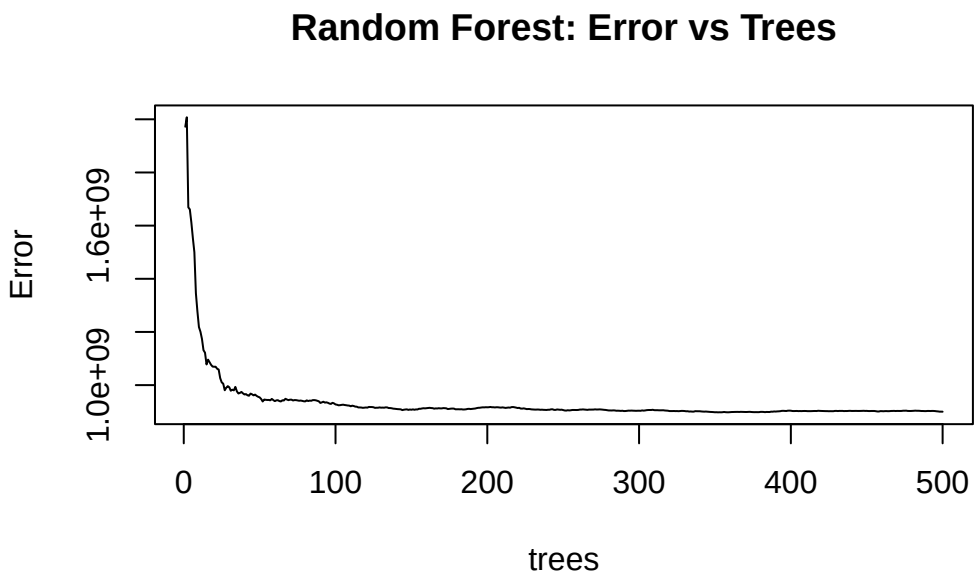

```
testing_sel <- testing %>% select(names(training.df))
pred_tree <- predict(tree.ames, newdata = testing_sel)

tree_perf <- testing_sel %>%
```

```
  mutate(pred = pred_tree,
         AE = abs(Sale_Price - pred),
         APE = 100*abs((Sale_Price - pred)/Sale_Price)) %>%
  summarise(Model = "Decision Tree",
            RMSE = sqrt(mean((Sale_Price - pred)^2)),
            MAE = mean(AE),
            MAPE = mean(APE),
            R2 = 1 - sum((Sale_Price - pred)^2)/sum((Sale_Price - mean(Sale_Price))^2))
tree_perf
```

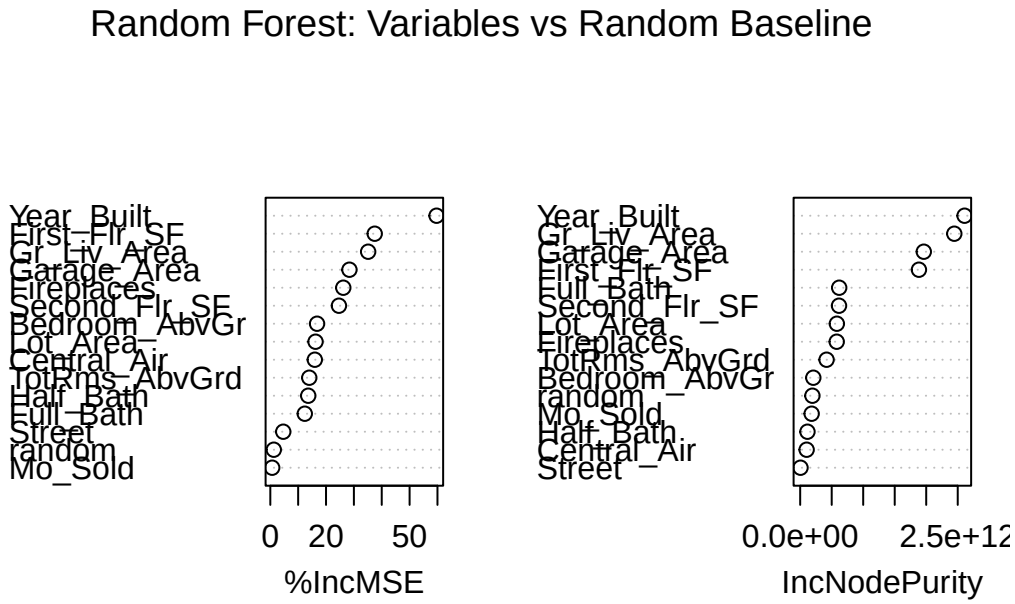
```
# A tibble: 1 x 5
  Model           RMSE    MAE  MAPE    R2
  <chr>          <dbl>  <dbl> <dbl> <dbl>
1 Decision Tree 42938. 28322.  17.5 0.727
```

6.4 Random Forest: Random Variable, PDPs, and Importance

```
# Add random variable for feature selection sanity check
training.df$random <- rnorm(nrow(training.df))

set.seed(12345)
rf.ames <- randomForest(Sale_Price ~ ., data = training.df, ntree = 500, mtry = 4, importanc
plot(rf.ames, main = "Random Forest: Error vs Trees")
```



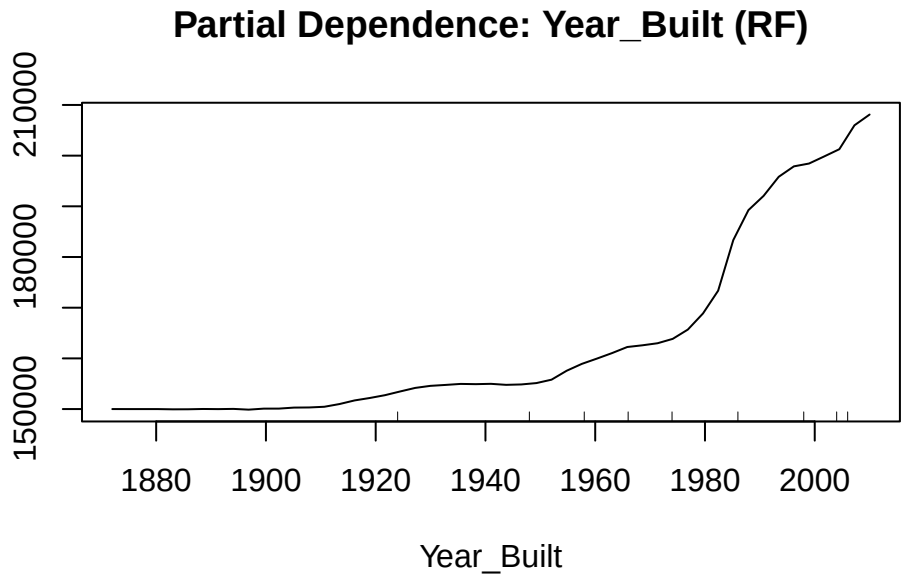

```
varImpPlot(rf.ames, sort = TRUE, n.var = 15, main = "Random Forest: Variables vs Random Basel
```



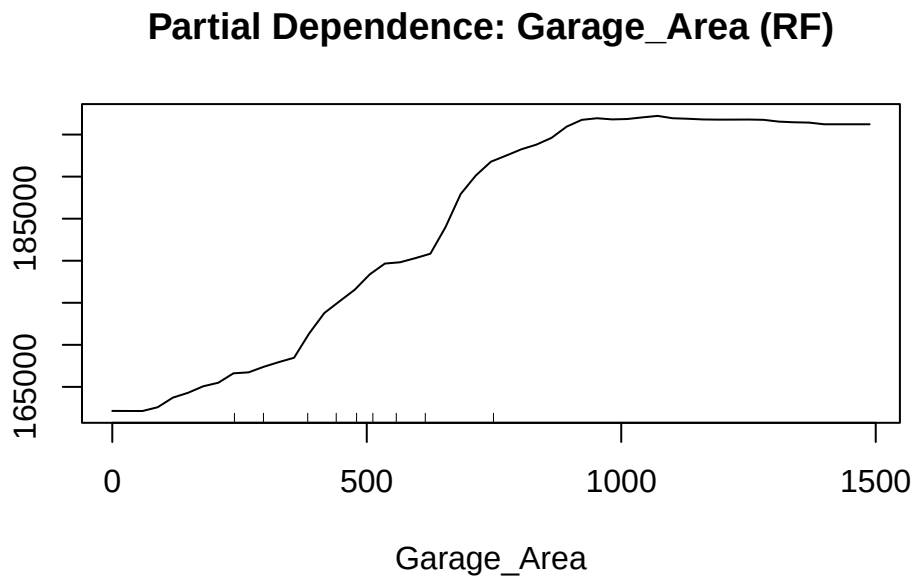
```r
importance(rf.ames)
```

```
                %IncMSE IncNodePurity
Bedroom_AbvGr 16.7830622  2.081292e+11
Year_Built    59.6827740  2.608576e+12
Mo_Sold        0.6717216  1.820404e+11
Lot_Area      16.2410098  5.818823e+11
Street         4.6451725  5.141533e+09
Central_Air   15.9709319  1.023533e+11
First_Flr_SF  37.4760827  1.884177e+12
Second_Flr_SF 24.6324055  6.159270e+11
Full_Bath     12.3716046  6.191162e+11
Half_Bath     13.6183708  1.153802e+11
Fireplaces    26.1893181  5.796209e+11
Garage_Area   28.4006898  1.961373e+12
Gr_Liv_Area   35.1279551  2.445543e+12
TotRms_AbvGrd 13.9865453  4.189078e+11
random         1.2436860  1.937137e+11
```

```r
# Partial dependence for key variables
partialPlot(rf.ames, training.df, Year_Built, main = "Partial Dependence: Year_Built (RF)")
```



```r
partialPlot(rf.ames, training.df, Garage_Area, main = "Partial Dependence: Garage_Area (RF)")
```



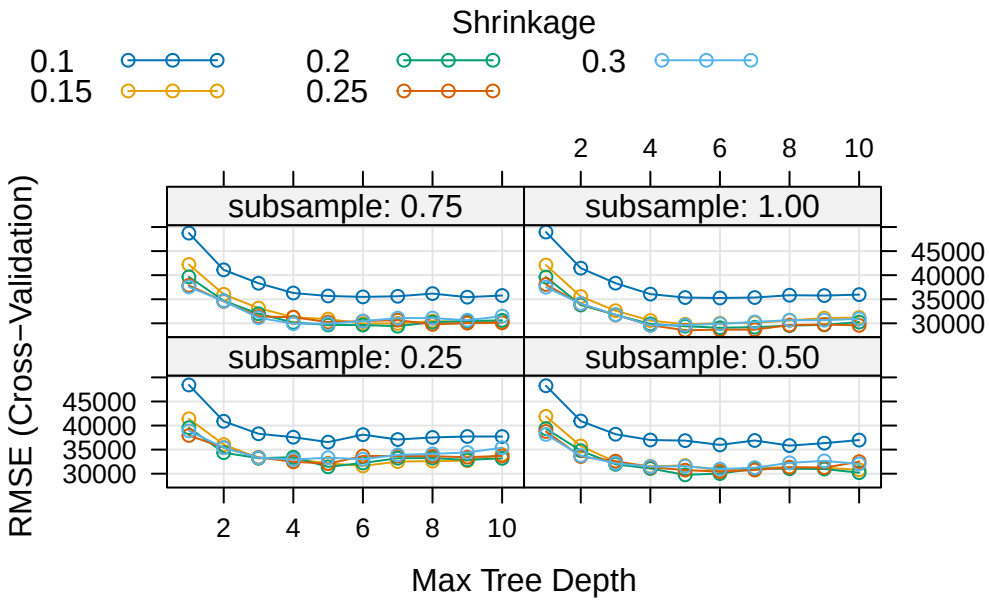
6.4.1 Random Forest Results

```r
testing_sel$random <- rnorm(nrow(testing_sel))
testing_sel$pred_rf <- predict(rf.ames, testing_sel)

rf_perf <- testing_sel %>%
  mutate(AE = abs(Sale_Price - pred_rf),
         APE = 100*abs((Sale_Price - pred_rf)/Sale_Price)) %>%
  summarise(Model = "Random Forest",
            RMSE = sqrt(mean((Sale_Price - pred_rf)^2)),
            MAE = mean(AE),
            MAPE = mean(APE),
            R2 = 1 - sum((Sale_Price - pred_rf)^2)/sum((Sale_Price - mean(Sale_Price))^2))
rf_perf
```

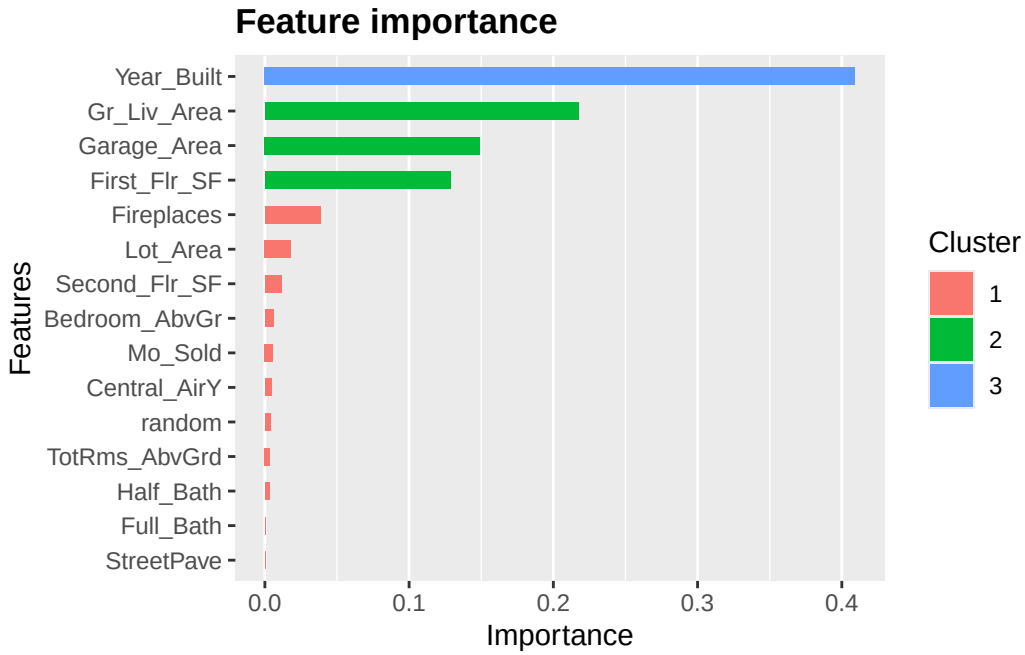
```
# A tibble: 1 x 5
  Model           RMSE    MAE  MAPE    R2
  <chr>          <dbl>  <dbl> <dbl> <dbl>
1 Random Forest 30147. 19223.  11.6 0.866
```

6.5 XGBoost: Random Variable, CV, Tuning, PDPs

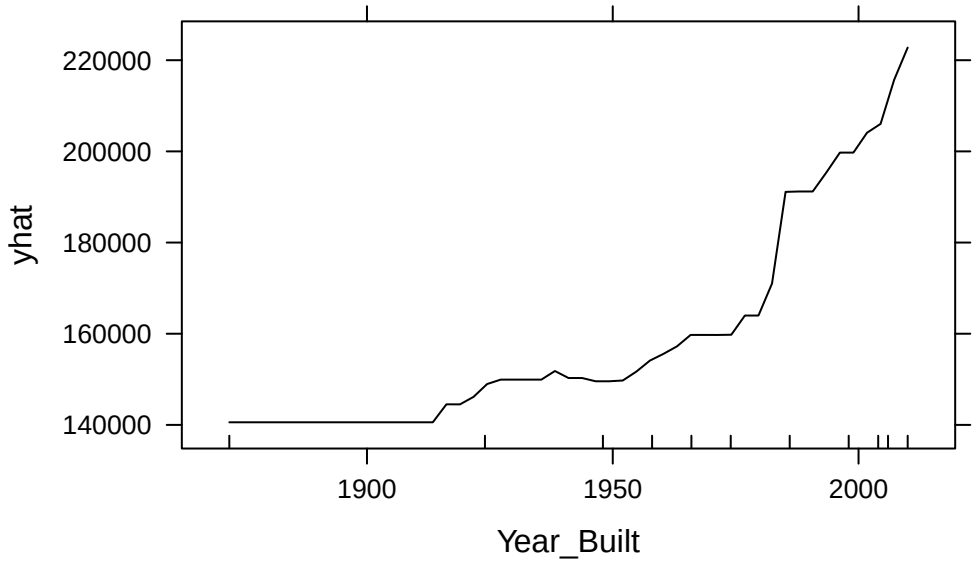
```
training$random <- rnorm(nrow(training))
train_x <- model.matrix(Sale_Price ~ ., data = training)[, -1]
train_y <- training$Sale_Price

set.seed(12345)
xgb.ames <- xgboost(
  data = train_x, label = train_y, subsample = 0.5, nrounds = 100,
  objective = "reg:squarederror", verbose = 0
)

xgbcv.ames <- xgb.cv(
  data = train_x, label = train_y, subsample = 0.5, nrounds = 100,
  nfold = 10, objective = "reg:squarederror", verbose = 0
)

tune_grid <- expand.grid(
  nrounds = 24,
  eta = c(0.1, 0.15, 0.2, 0.25, 0.3),
  max_depth = c(1:10),
  gamma = c(0),
  colsample_bytree = 1,
  min_child_weight = 1,
  subsample = c(0.25, 0.5, 0.75, 1)
)

set.seed(12345)
xgb.ames.caret <- train(
  x = train_x, y = train_y,
  method = "xgbTree",
  tuneGrid = tune_grid,
  trControl = trainControl(method = 'cv', number = 10)
)
plot(xgb.ames.caret)
```



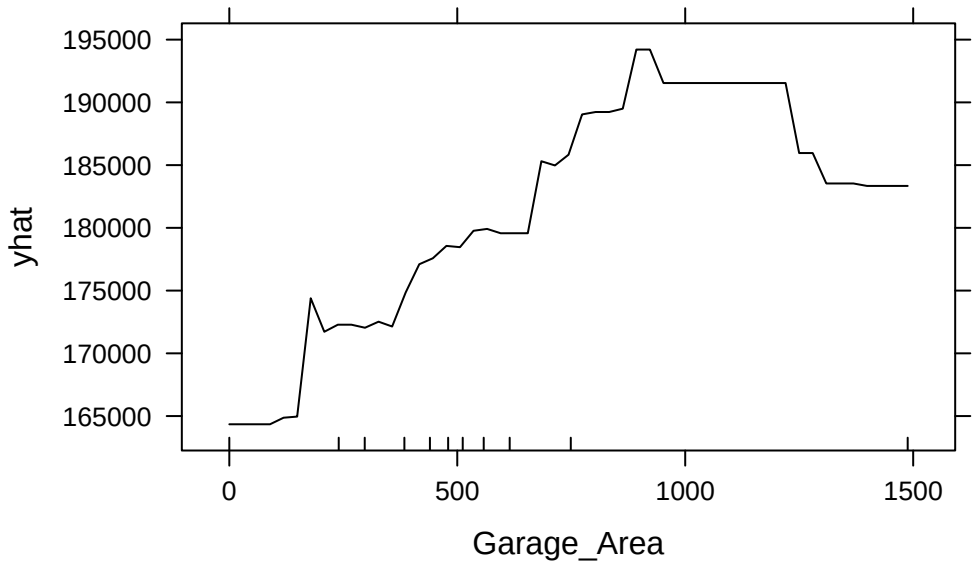

```r
# Fit tuned model
xgb.ames <- xgboost(
  data = train_x, label = train_y, subsample = 1,
  nrounds = 24, eta = 0.25, max_depth = 5,
  objective = "reg:squarederror", verbose = 0
)

# Variable importance including random variable
xgb.ggplot.importance(xgb.importance(feature_names = colnames(train_x), model = xgb.ames))
```




```
# Partial dependence plots
train_df <- as.data.frame(train_x)

partial(xgb.ames, pred.var = "Year_Built",
        plot = TRUE, rug = TRUE, alpha = 0.1, plot.engine = "lattice",
        train = train_df, pdp.color = "red")
```




```
partial(xgb.ames, pred.var = "Garage_Area",
        plot = TRUE, rug = TRUE, alpha = 0.1, plot.engine = "lattice",
        train = train_df)
```



6.5.1 XGBoost Results (Aligned Columns)

```
testing$random <- rnorm(nrow(testing))
testing_x <- model.matrix(Sale_Price ~ ., data = testing)[, -1]

# Align test columns to training columns
missing_cols <- setdiff(colnames(train_x), colnames(testing_x))
for (col in missing_cols) {
  testing_x <- cbind(testing_x, 0)
  colnames(testing_x)[ncol(testing_x)] <- col
}
testing_x <- testing_x[, colnames(train_x)]

testing_sel$pred_xgb <- predict(xgb.ames, testing_x)

xgb_perf <- testing_sel %>%
  mutate(AE = abs(Sale_Price - pred_xgb),
         APE = 100*abs((Sale_Price - pred_xgb)/Sale_Price)) %>%
  summarise(Model = "XGBoost",
            RMSE = sqrt(mean((Sale_Price - pred_xgb)^2)),
            MAE = mean(AE),
            MAPE = mean(APE),
            R2 = 1 - sum((Sale_Price - pred_xgb)^2)/sum((Sale_Price - mean(Sale_Price))^2))
xgb_perf
```

```
# A tibble: 1 x 5
  Model     RMSE    MAE  MAPE    R2
  <chr>    <dbl>  <dbl> <dbl> <dbl>
1 XGBoost 30355. 19820.  11.8 0.864
```

6.6 Final Model Comparison

```
bind_rows(tree_perf, rf_perf, xgb_perf) %>%
  arrange(RMSE)
```

```
# A tibble: 3 x 5
  Model           RMSE    MAE  MAPE    R2
  <chr>          <dbl>  <dbl> <dbl> <dbl>
```

```
1 Random Forest 30147. 19223.  11.6 0.866
2 XGBoost       30355. 19820.  11.8 0.864
3 Decision Tree 42938. 28322.  17.5 0.727
```

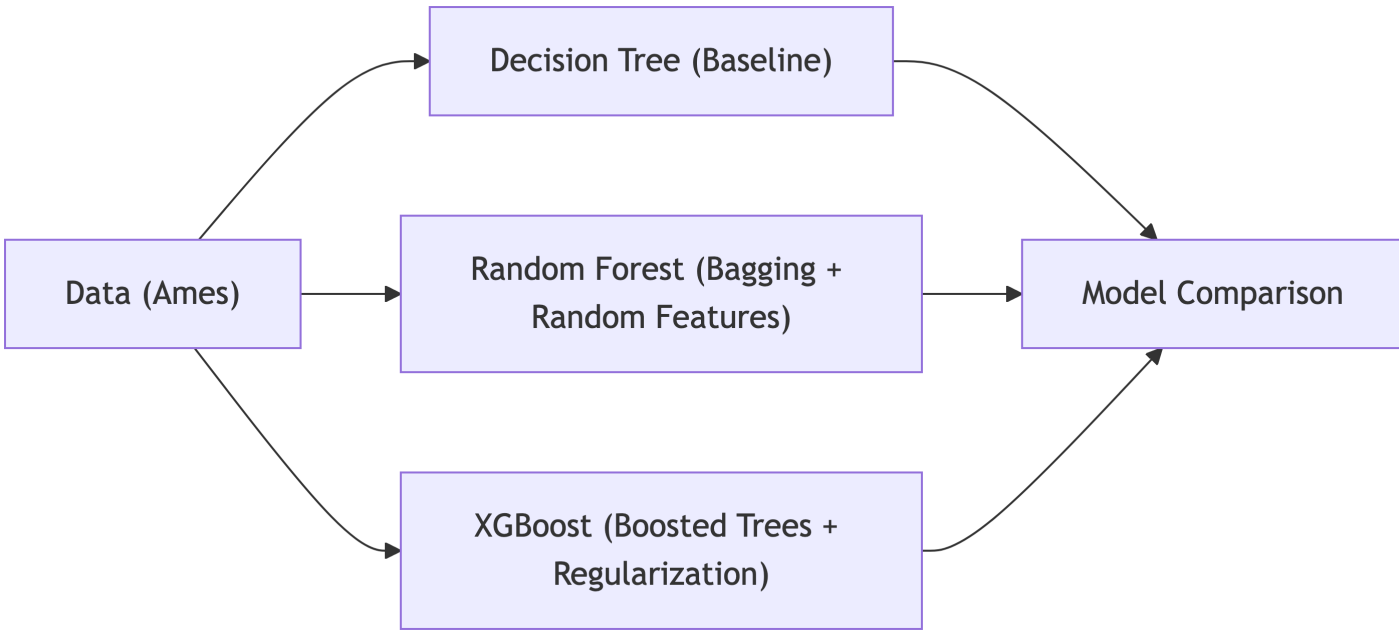
Summary: The Decision Tree provides interpretability, Random Forest and XGBoost attempt to further reduce bias and manage variance error through randomization and ensembles of weak learners. The issue is that they become less and less interpretable, so we have to introduce other tools to understand what variables are important and how they are relating to the outcome.

7 Tree Based Models - Python Version

Tree Models: **Decision Tree**, **Random Forest**, and **XGBoost** on the Ames Housing dataset. Includes random-variable baselines, variable importance, partial dependence plots, and hyperparameter tuning (random search).

7.1 Visual Roadmap

```
flowchart LR
    A["Data (Ames)"]
    DT["Decision Tree (Baseline)"]
    RF["Random Forest (Bagging + Random Features)"]
    XGB["XGBoost (Boosted Trees + Regularization)"]
    Compare["Model Comparison"]

    A --> DT
    A --> RF
    A --> XGB
    DT --> Compare
    RF --> Compare
    XGB --> Compare
```



7.2 Setup and Data Preparation

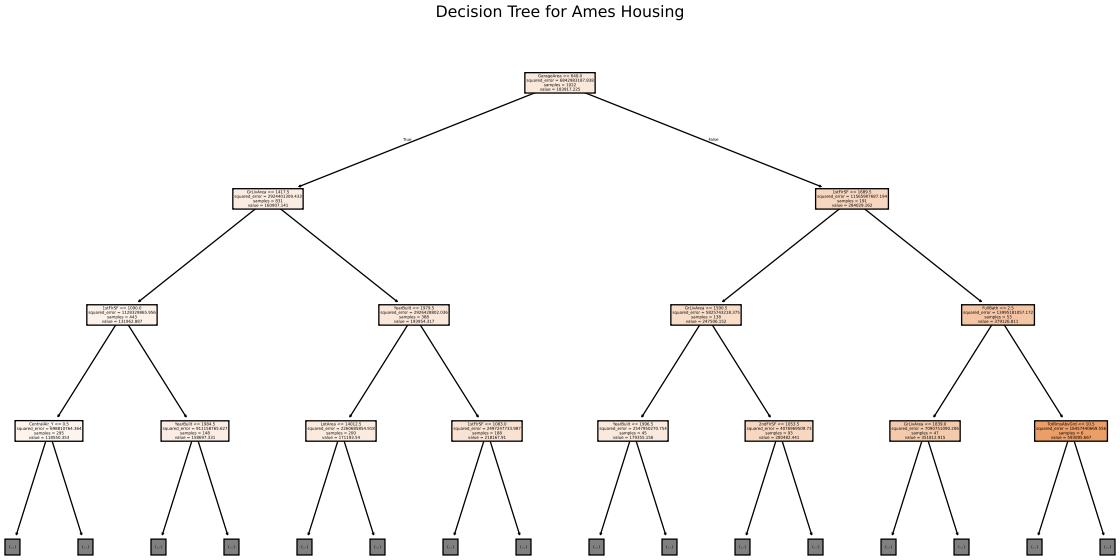
```
import pandas as pd
import numpy as np
from sklearn.datasets import fetch_openml
from sklearn.model_selection import train_test_split, RandomizedSearchCV, KFold
from sklearn.tree import DecisionTreeRegressor, plot_tree
from sklearn.ensemble import RandomForestRegressor
from sklearn.inspection import PartialDependenceDisplay
from xgboost import XGBRegressor, plot_importance
from sklearn.metrics import mean_squared_error, mean_absolute_error, r2_score
import matplotlib.pyplot as plt
import warnings
warnings.filterwarnings("ignore")

# Set random seed for reproducibility
seed = 12345
np.random.seed(seed)

# Load Ames dataset
ames = fetch_openml(name="house_prices", as_frame=True)
```

```
df = ames.frame

# Feature selection matching R version
features = [
    'BedroomAbvGr', 'YearBuilt', 'MoSold', 'LotArea', 'Street', 'CentralAir',
    '1stFlrSF', '2ndFlrSF', 'FullBath', 'HalfBath', 'Fireplaces',
    'GarageArea', 'GrLivArea', 'TotRmsAbvGrd'
]
df = df[features + ['SalePrice']].dropna()

# Encode categoricals
df = pd.get_dummies(df, drop_first=True)

# Train/test split
train, test = train_test_split(df, test_size=0.3, random_state=seed)

# Add random variable for variable-importance baseline
train['random'] = np.random.randn(len(train))
test['random'] = np.random.randn(len(test))

X_train = train.drop(columns=['SalePrice'])
y_train = train['SalePrice']
X_test = test.drop(columns=['SalePrice'])
y_test = test['SalePrice']
```

7.3 Decision Tree (Baseline)

```
tree = DecisionTreeRegressor(random_state=seed, max_depth=5)
tree.fit(X_train, y_train)

plt.figure(figsize=(16,8))
plot_tree(tree, feature_names=X_train.columns, filled=True, max_depth=3)
plt.title("Decision Tree for Ames Housing")
plt.show()

# Metrics
pred_tree = tree.predict(X_test)
rmse_tree = mean_squared_error(y_test, pred_tree)
mae_tree = mean_absolute_error(y_test, pred_tree)
```

```
mape_tree = np.mean(np.abs((y_test - pred_tree) / y_test)) * 100
r2_tree = r2_score(y_test, pred_tree)

pd.DataFrame([["Decision Tree", rmse_tree, mae_tree, mape_tree, r2_tree]],
             columns=["Model", "RMSE", "MAE", "MAPE", "R2"])
```



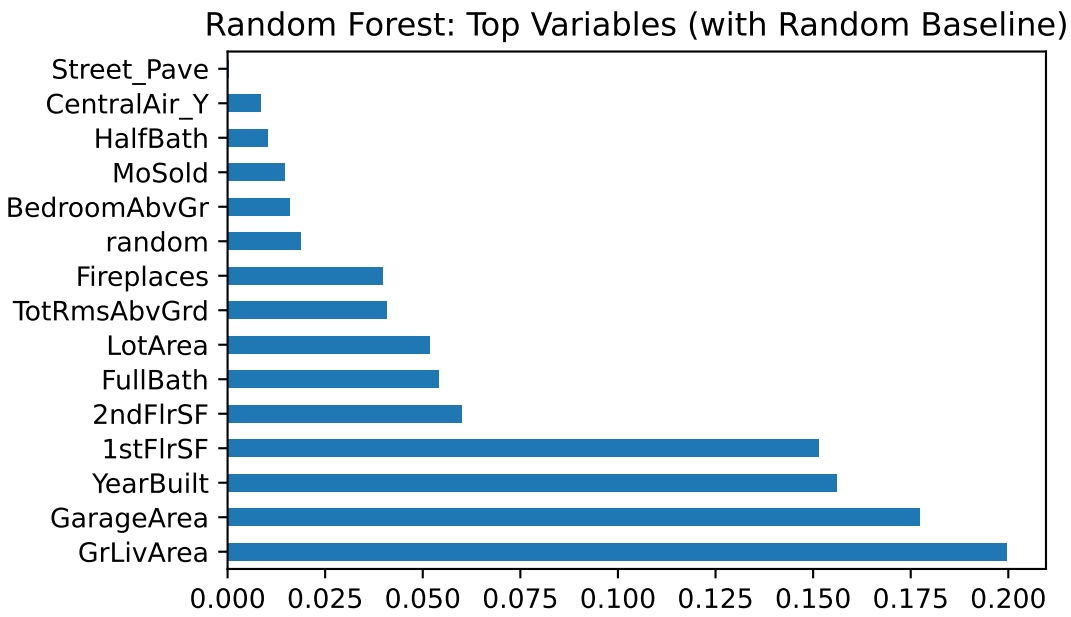
	Model	RMSE	MAE	MAPE	R2
0	Decision Tree	2.378377e+09	28465.348099	17.701235	0.522975

7.4 Random Forest: Random Variable, Importance, PDPs, and Tuning

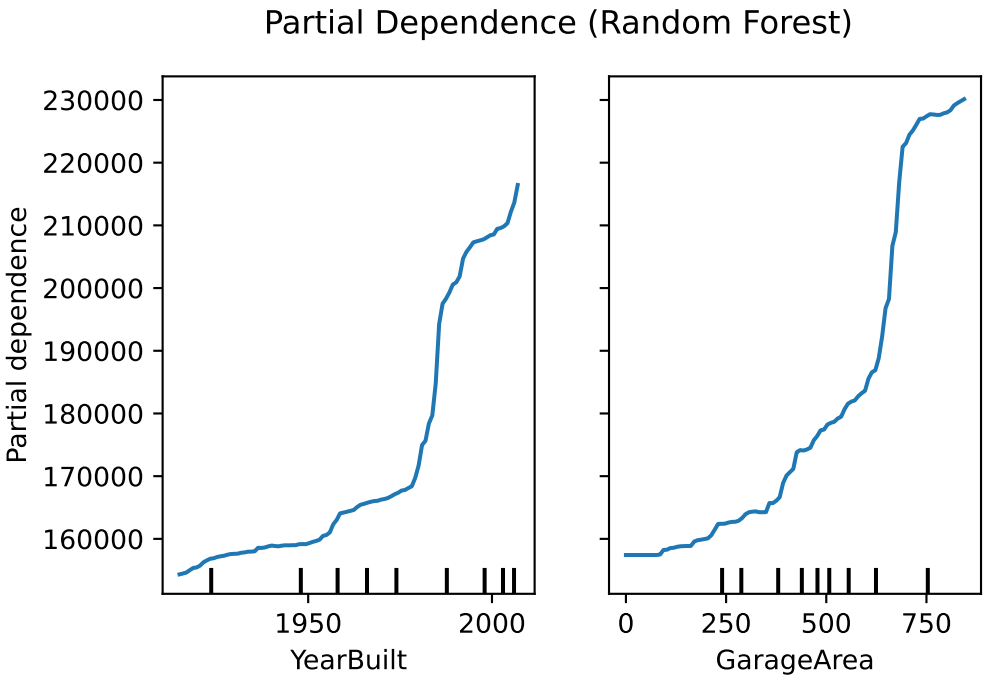
```
rf = RandomForestRegressor(random_state=seed)

# Define parameter grid (similar to R tuning)
param_dist_rf = {
    'n_estimators': [100, 200, 300, 400, 500],
    'max_features': ['sqrt', 'log2', None, 4, 6, 8],
    'max_depth': [None, 5, 10, 15, 20],
    'min_samples_split': [2, 5, 10]
```

```
}

cv = KFold(n_splits=10, shuffle=True, random_state=seed)

rf_random = RandomizedSearchCV(
    estimator=rf,
    param_distributions=param_dist_rf,
    n_iter=20,
    cv=cv,
    random_state=seed,
    n_jobs=-1,
    scoring='neg_root_mean_squared_error'
)

rf_random.fit(X_train, y_train)
rf_best = rf_random.best_estimator_
print("Best Random Forest Parameters:", rf_random.best_params_)

# Importance with random variable
importances = pd.Series(rf_best.feature_importances_, index=X_train.columns).sort_values(asc
importances[:15].plot(kind='barh', title='Random Forest: Top Variables (with Random Baseline
plt.show()

# Partial dependence plots
plt.figure()
PartialDependenceDisplay.from_estimator(rf_best, X_train, ['YearBuilt', 'GarageArea'])
plt.suptitle("Partial Dependence (Random Forest)")
plt.show()

# Performance metrics
pred_rf = rf_best.predict(X_test)
rmse_rf = mean_squared_error(y_test, pred_rf)
mae_rf = mean_absolute_error(y_test, pred_rf)
mape_rf = np.mean(np.abs((y_test - pred_rf) / y_test)) * 100
r2_rf = r2_score(y_test, pred_rf)

pd.DataFrame([["Random Forest (Tuned)", rmse_rf, mae_rf, mape_rf, r2_rf]],
             columns=["Model", "RMSE", "MAE", "MAPE", "R2"])
```

```
Best Random Forest Parameters: {'n_estimators': 500, 'min_samples_split': 2, 'max_features':
```




```
<Figure size 1650x1050 with 0 Axes>
```



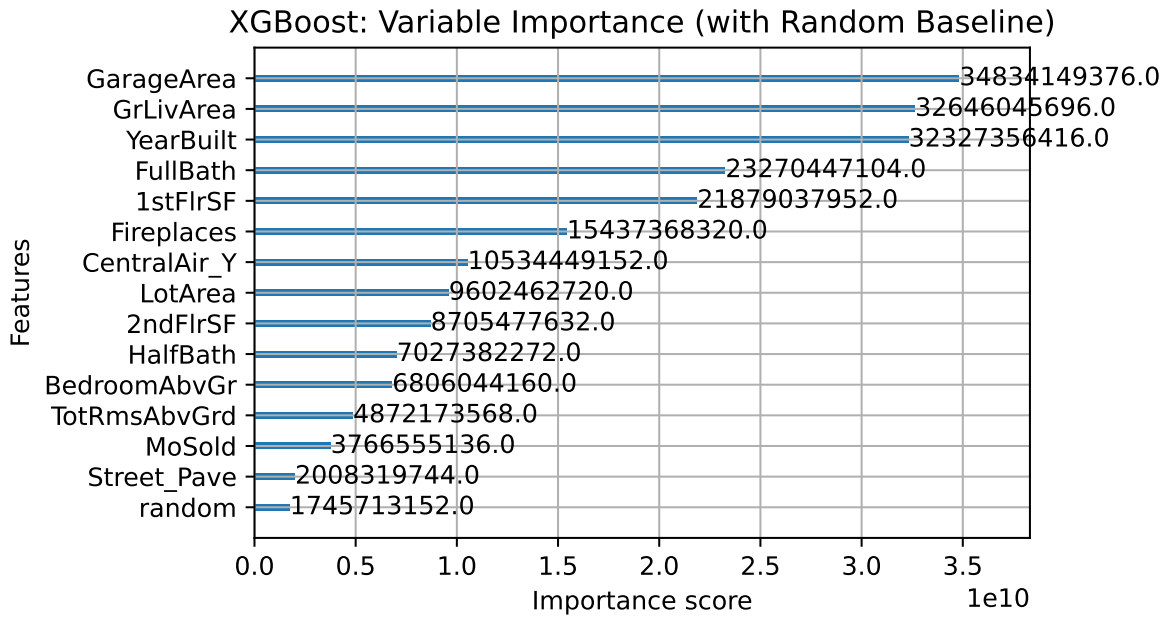
	Model	RMSE	MAE	MAPE	R2
0	Random Forest (Tuned)	1.093313e+09	19889.772151	12.832756	0.780717

7.5 XGBoost: Random Variable, Importance, PDPs, and Tuning

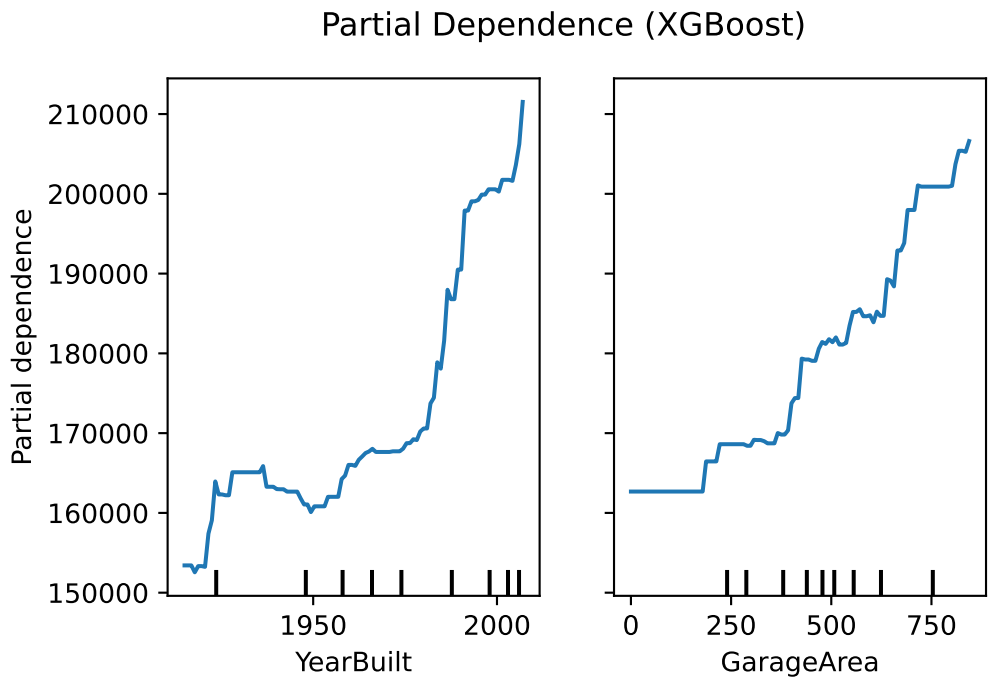
```
xgb = XGBRegressor(
    objective='reg:squarederror',
    random_state=seed,
    n_jobs=-1
)

# Define tuning grid similar to R (eta = learning_rate, subsample, max_depth)
param_dist_xgb = {
    'n_estimators': [24, 50, 100, 200],
    'learning_rate': [0.1, 0.15, 0.2, 0.25, 0.3],
    'max_depth': [1, 3, 5, 7, 10],
    'subsample': [0.25, 0.5, 0.75, 1],
    'colsample_bytree': [0.5, 0.75, 1]
}

xgb_random = RandomizedSearchCV(
    estimator=xgb,
    param_distributions=param_dist_xgb,
    n_iter=20,
    cv=cv,
    random_state=seed,
    n_jobs=-1,
    scoring='neg_root_mean_squared_error'
)

xgb_random.fit(X_train, y_train)
xgb_best = xgb_random.best_estimator_
print("Best XGBoost Parameters:", xgb_random.best_params_)

# Importance plot with random variable
plot_importance(xgb_best, max_num_features=15, importance_type='gain')
plt.title("XGBoost: Variable Importance (with Random Baseline)")
plt.show()
```

```
# Partial dependence plots
plt.figure()
PartialDependenceDisplay.from_estimator(xgb_best, X_train, ['YearBuilt', 'GarageArea'])
plt.suptitle("Partial Dependence (XGBoost)")
plt.show()

# Metrics
pred_xgb = xgb_best.predict(X_test)
rmse_xgb = mean_squared_error(y_test, pred_xgb)
mae_xgb = mean_absolute_error(y_test, pred_xgb)
mape_xgb = np.mean(np.abs((y_test - pred_xgb) / y_test)) * 100
r2_xgb = r2_score(y_test, pred_xgb)

pd.DataFrame([["XGBoost (Tuned)", rmse_xgb, mae_xgb, mape_xgb, r2_xgb]],
             columns=["Model", "RMSE", "MAE", "MAPE", "R2"])
```

```
Best XGBoost Parameters: {'subsample': 0.75, 'n_estimators': 50, 'max_depth': 5, 'learning_ra
```




```
<Figure size 1650x1050 with 0 Axes>
```



	Model	RMSE	MAE	MAPE	R2
0	XGBoost (Tuned)	1.133839e+09	20819.921875	13.330445	0.772589

7.6 Final Model Comparison

```
results = pd.DataFrame([
    ["Decision Tree", rmse_tree, mae_tree, mape_tree, r2_tree],
    ["Random Forest (Tuned)", rmse_rf, mae_rf, mape_rf, r2_rf],
    ["XGBoost (Tuned)", rmse_xgb, mae_xgb, mape_xgb, r2_xgb]
], columns=["Model", "RMSE", "MAE", "MAPE", "R2"]).sort_values("RMSE")
results
```

	Model	RMSE	MAE	MAPE	R2
1	Random Forest (Tuned)	1.093313e+09	19889.772151	12.832756	0.780717
2	XGBoost (Tuned)	1.133839e+09	20819.921875	13.330445	0.772589
0	Decision Tree	2.378377e+09	28465.348099	17.701235	0.522975

Summary: The Decision Tree provides interpretability, Random Forest and XGBoost attempt to further reduce bias and manage variance error through randomization and ensembles of weak learners. The issue is that they become less and less interpretable, so we have to introduce other tools to understand what variables are important and how they are relating to the outcome.

8 Understanding Random Forest and XGBoost Hyperparameters

8.1 Random Forest Hyperparameters

8.1.1 Switching Between Regression and Classification

Language	How to Specify Regression	How to Specify Classification
R	`randomForest(y ~ ., data = df)` where `y` is **numeric**	`randomForest(y ~ ., data = df)` where `y` is a **factor**
Python	`RandomForestRegressor()`	`RandomForestClassifier()`

In R, the outcome type (`numeric` vs `factor`) determines whether the model performs **regression** or **classification** automatically.
In Python, you explicitly choose the class (`RandomForestRegressor` or `RandomForestClassifier`).

Concept	**R** (`randomForest`)	**Python** (`RandomForestRegressor` / `RandomForestClassifier`)	Meaning / Purpose	Typical Values	Impact / Interpretation
Model type	Determined by outcome type (`numeric` vs `factor`)	Different estimator class	Defines whether the forest predicts continuous or categorical outcomes	–	Choose based on target type

Concept	**R** (`randomForest`)	**Python** (`RandomForestRegressor` / `RandomForestClassifier`)	Meaning / Role	Typical Values	Impact / Interpretation
Number of trees	`ntree`	`n_estimators`	Number of trees to grow in the forest	100–1000 (500 common)	More trees reduce variance but increase runtime
Features per split	`mtry`	`max_features`	# of variables randomly selected at each split	Regression: $\sqrt{p}$; Classification: p/3	Smaller → higher bias, more diversity; larger → lower bias, more correlation
Max tree depth	*(no direct arg, via `nodesize`)*	`max_depth`	Maximum depth of each tree	None, or 5–30	Deeper trees capture complexity but risk overfitting
Min samples per split	*(via `nodesize` indirectly)*	`min_samples_split`	Minimum # of samples to split a node	2–10	Larger → smoother, higher bias; smaller → lower bias, higher variance
Min samples per leaf	*(via `nodesize`)*	`min_samples_leaf`	Minimum samples per terminal node	1–5	Prevents tiny leaves, reduces overfitting
Bootstrap sampling	`replace`	`bootstrap`	Whether to sample with replacement	TRUE/FALSE	Adds randomness; TRUE recommended
Sample fraction per tree	*(rarely tuned)*	`max_samples`	Fraction of training data used per tree	0.5–1.0	Lower → more randomness, higher bias

Concept	R (`randomForest`)	Python (`RandomForestRegressor` / `RandomForestClassifier`)	Meaning / Tasks	Typical Values	Impact / Interpretation
Learning objective	Implicit: minimize MSE (regression) or Gini/entropy (classification)	Implicit: minimize MSE or Gini/entropy	Objective changes automatically with model type	–	Regression averages numeric values; classification votes on class labels
Random seed	`set.seed()`	`random_state`	Ensures reproducibility	any integer	Use same seed for consistency

---

8.1.2 Thinking About Random Forest Tuning

- Start with **many trees** (500).
- Tune `mtry` / `max_features` to balance bias vs variance.
- Use `max_depth`, `min_samples_split`, or `min_samples_leaf` to prevent overfitting.
- RF tuning is relatively forgiving — focus on **efficiency and stability** rather than micro-optimization.
- For **classification**, check **OOB error** or **ROC-AUC** rather than RMSE.

8.2 XGBoost Hyperparameters

Concept	R (`xgboost`)	Python (`XGBRegressor` / `XGBClassifier`)	Meaning / Role	Typical Values	Impact / Interpretation
Boosting rounds	`nrounds`	`n_estimators`	Total # of trees (boosting steps)	100–1000	Too high → overfit unless learning rate small
Learning rate	`eta`	`learning_rate`	Shrinks each tree's contribution	0.01–0.3	Lower = slower, safer; higher = faster, riskier
Tree depth	`max_depth`	`max_depth`	Max depth per tree	3–10	Shallow = less overfit; deep = more complex
Min child weight	`min_child_weight`	`min_child_weight`	Min sum of instance weights per leaf	1–10	Larger = more conservative splits
Subsample (rows)	`subsample`	`subsample`	Fraction of training rows used per round	0.5–1.0	Lower adds randomness, improves generalization
Col sample (features)	`colsample_bytree`	`colsample_bytree`	Fraction of features per tree	0.5–1.0	Reduces feature correlation
Gamma	`gamma`	`gamma`	Min loss reduction to split a node	0–5	Higher = more conservative, prevents overfitting
L1 regularization	`alpha`	`reg_alpha`	Penalty on abs(weights)	0–10	Encourages sparsity (feature selection)
L2 regularization	`lambda`	`reg_lambda`	Penalty on squared(weights)	0–10	Controls weight magnitudes (stability)

Concept	R (`xgboost`)	Python (`XGBRegressor` / `XGBClassifier`)	Meaning / Role	Typical Values	Impact / Interpretation
Learning objective	`objective`	`objective`	Specifies prediction type	`"reg:squarederror"`, `"binary:logistic"`, etc.	Must match your target type
Early stopping	via `xgb.cv()`	`early_stopping_rounds`	Stop when no improvement	10–50 rounds	Prevents unnecessary rounds
Seed	`set.seed()`	`random_state`	Ensures reproducibility	any integer	Consistent results across runs

---

8.2.1 Thinking About XGBoost Tuning

1. **Start simple** — tune `n_estimators`, `max_depth`, and `learning_rate` first.
2. Add regularization (`gamma`, `alpha`, `lambda`) to control overfitting.
3. Use `subsample` and `colsample_bytree` to add randomness.

8.3 Practical Tips and Heuristics

Situation	Random Forest Strategy	XGBoost Strategy
Overfitting	Increase `min_samples_split`, reduce `max_depth`	Lower `learning_rate`, increase `gamma` or `min_child_weight`
Underfitting	Increase `n_estimators`, allow deeper trees	Lower regularization, increase `max_depth` or `n_estimators`
Slow training	Reduce trees or sample fraction	Increase `learning_rate`, reduce trees

Situation	Random Forest Strategy	XGBoost Strategy
Unstable results	Fix `random_state`, use more trees	Fix `random_state`, enable early stopping
High variance	Lower `max_features`, increase randomness	Increase `subsample` diversity, lower `max_depth`

8.4 Rule-of-Thumb Summary

Parameter	Low Value →	High Value →
`n_estimators`	Underfitting	Stable but slower
`max_depth`	Simpler model (underfit)	Complex model (overfit)
`learning_rate`	Slow, steady, robust	Fast, risk of overfit
`subsample` / `colsample_bytree`	More bias, less variance	Lower bias, higher variance
`mtry` / `max_features`	More randomization	More correlation between trees
`min_samples_split` / `min_child_weight`	Flexible, low bias	Conservative, higher bias

8.4.1 Key Takeaways

- **Random Forest**
  - Regression/classification determined by outcome type (R) or model class (Python).
  - Reduces variance through bagging.
  - Tuning focuses on **stability** and **efficiency**.
- **XGBoost**
  - Requires explicit objective selection.
  - Reduces error via boosting and regularization.
- Use interpretability tools (importance, PDPs, SHAP) to understand **why** tuning changes model behavior.

9 Neural Networks in R

9.1 Introduction

This chapter introduces **neural networks in R**, progressing from feed-forward to recurrent (RNN) and convolutional (CNN) architectures.
It uses the Ames housing dataset, a sine wave sequence, and the MNIST digit dataset to demonstrate increasingly complex network structures.

Each example illustrates a different ANN architecture.

9.2 1. Feed-Forward Neural Networks (Ames Data)

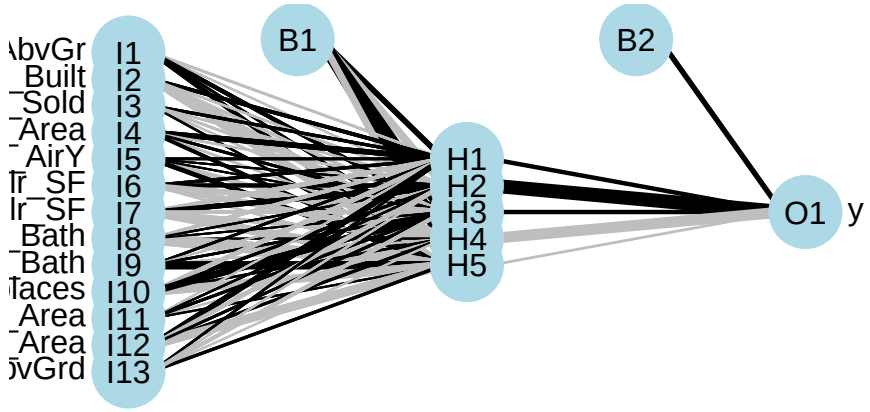
Feed-forward networks are the simplest type of neural networks. They map input features (like home size, rooms, or year built) to an output (home price) through **layers of neurons**. Each neuron applies a weighted sum and activation function, allowing the network to learn complex relationships between variables.

```
#Note: This is specific to my environment
library(reticulate)
use_python("/Users/donaldhale/r-arm/bin/python", required = TRUE)

library(keras3)

tf <- reticulate::import("tensorflow", delay_load = FALSE)
cat("TensorFlow version:", tf$`__version__`, "\n")
```

```
TensorFlow version: 2.16.1
```

```r
library(tidyverse)
library(caret)
library(nnet)
library(NeuralNetTools)
library(AmesHousing)

set.seed(4321)

# Load and sample data
ames <- make_ordinal_ames() %>% mutate(id = row_number())
train_idx <- sample(1:nrow(ames), 0.7 * nrow(ames))
training <- ames[train_idx, ]
testing  <- ames[-train_idx, ]

# Select a manageable subset of variables
vars <- c("Sale_Price", "Bedroom_AbvGr", "Year_Built", "Mo_Sold", "Lot_Area",
          "Street", "Central_Air", "First_Flr_SF", "Second_Flr_SF",
          "Full_Bath", "Half_Bath", "Fireplaces", "Garage_Area",
          "Gr_Liv_Area", "TotRms_AbvGrd")
training <- training[, vars]
testing  <- testing[, vars]

# Convert categorical predictors into dummy numeric columns
x_train <- model.matrix(Sale_Price ~ ., data = training)[, -1]
y_train <- training$Sale_Price
x_test  <- model.matrix(Sale_Price ~ ., data = testing)[, -1]
y_test  <- testing$Sale_Price

# Drop low-variance predictors and scale for stable training
nzv <- nearZeroVar(x_train)
if (length(nzv) > 0) {
  x_train <- x_train[, -nzv, drop = FALSE]
  x_test  <- x_test[, -nzv, drop = FALSE]
}

x_train <- scale(x_train)
x_test  <- scale(x_test, center = attr(x_train, "scaled:center"),
                 scale = attr(x_train, "scaled:scale"))

y_train_s <- scale(y_train)

# Combine into a single training dataframe
```

```r
train_df <- data.frame(y = as.numeric(y_train_s), x_train)

set.seed(123)
nn.ames <- nnet(y ~ ., data = train_df, size = 5, linout = TRUE, trace = FALSE)
plotnet(nn.ames)
```



9.2.1 1.2 Cross-Validation Tuning

We tune two important parameters: - **`size`** — number of neurons in the hidden layer - **`decay`** — regularization strength (prevents overfitting)

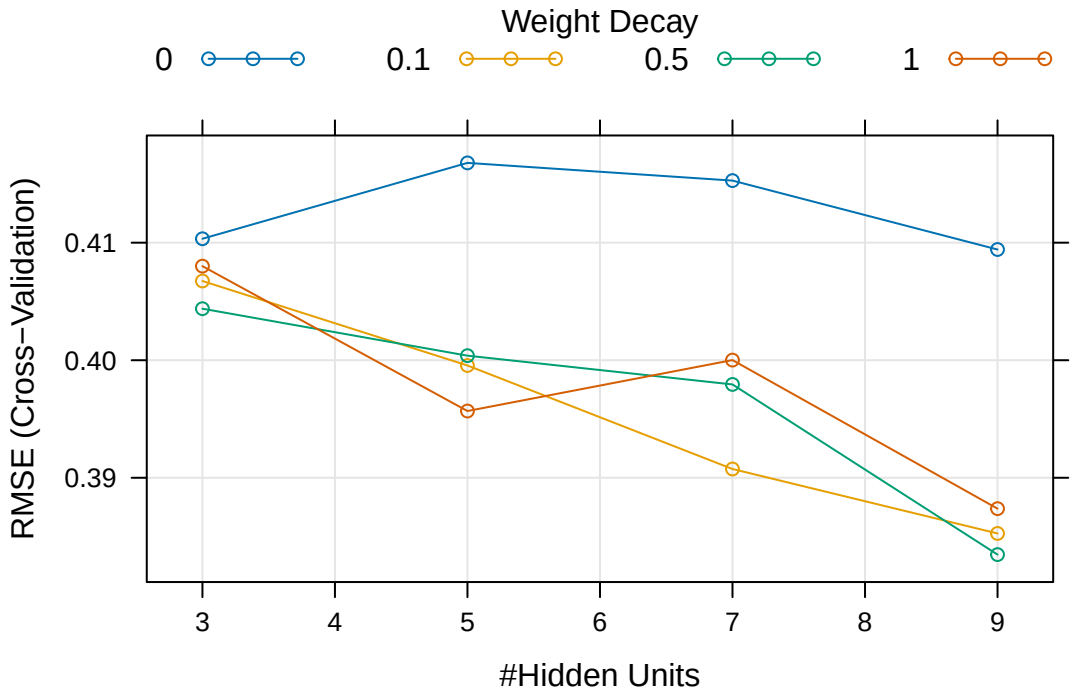
`caret::train()` automates this search using k-fold cross-validation.

```r
tune_grid <- expand.grid(size = c(3,5,7,9), decay = c(0,0.1,0.5,1))
ctrl <- trainControl(method = "cv", number = 5)

set.seed(123)
```

```r
nn.tuned <- train(
  x = x_train, y = as.numeric(y_train_s),
  method = "nnet",
  tuneGrid = tune_grid,
  trControl = ctrl,
  linout = TRUE,
  trace = FALSE
)

nn.tuned$bestTune
```

```
   size decay
15    9   0.5
```

```r
plot(nn.tuned)
```



9.2.2 1.3 Evaluate Performance

After identifying the best parameters, we retrain the model and evaluate its predictive accuracy.

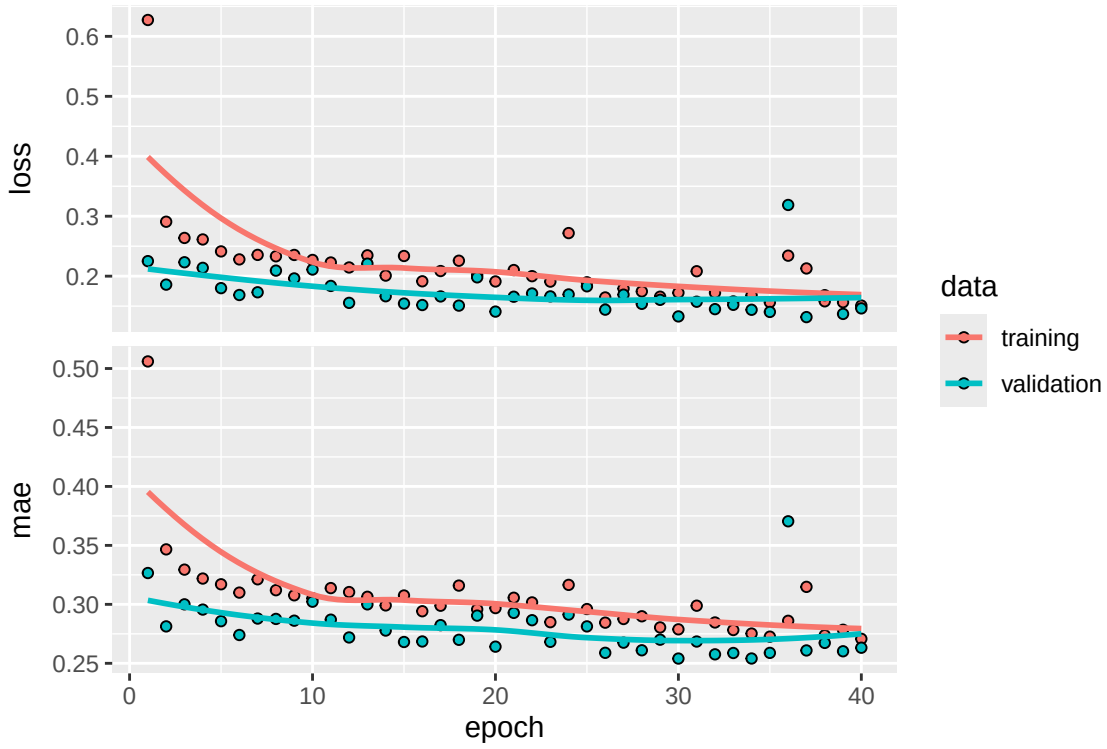
```
best_size  <- nn.tuned$bestTune$size
best_decay <- nn.tuned$bestTune$decay

set.seed(123)
nn.final <- nnet(
  y ~ .,
  data = train_df,
  size = best_size,
  decay = best_decay,
  linout = TRUE,
  trace = FALSE
)

pred_std <- predict(nn.final, newdata = as.data.frame(x_test))
pred_nn  <- pred_std * attr(y_train_s, "scaled:scale") + attr(y_train_s, "scaled:center")

perf <- data.frame(
  MAE  = mean(abs(y_test - pred_nn)),
  MAPE = mean(abs((y_test - pred_nn) / y_test)) * 100
)
perf
```

```
       MAE     MAPE
1 20392.57 12.63864
```

9.3 2. Feed-Forward Neural Networks (Keras 3)

The **Keras 3** API is a modern deep learning interface. It supports more complex models and GPUs while maintaining a simple syntax.
This example shows the same concept as above using a multi-layer perceptron (MLP).

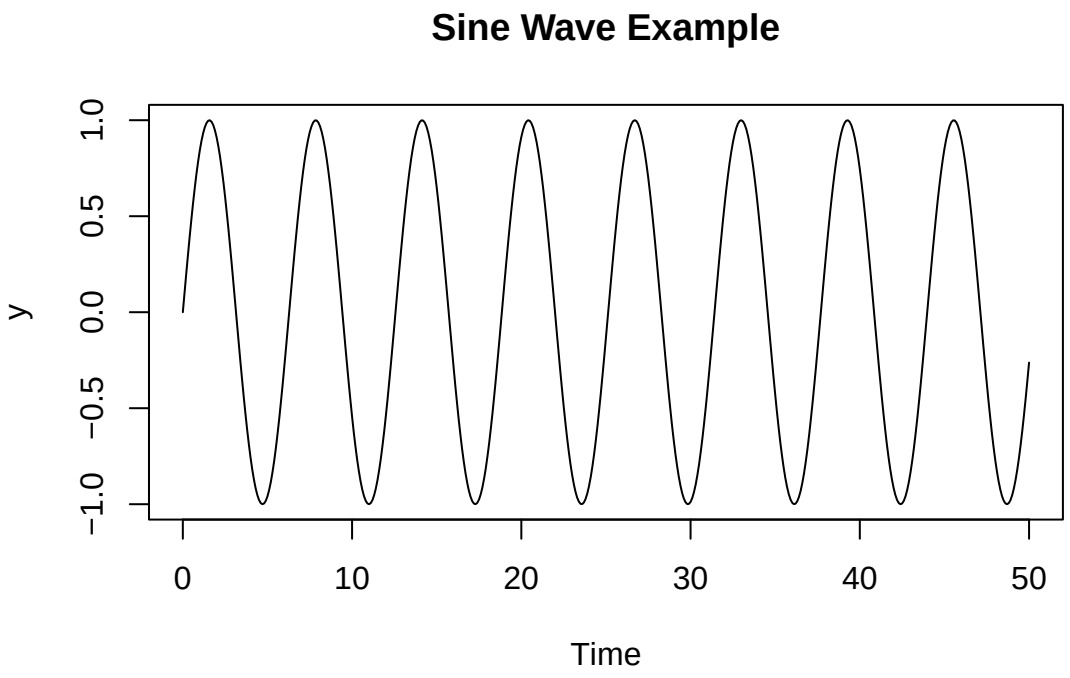
```r
#library(keras3)
set.seed(123)

model_ff <- keras_model_sequential() |>
  layer_dense(units = 16, activation = "relu", input_shape = ncol(x_train)) |>
  layer_dropout(rate = 0.1) |>
  layer_dense(units = 1)

compile(model_ff, optimizer = optimizer_adam(learning_rate = 0.01),
        loss = "mse", metrics = "mae")

history <- fit(model_ff, x_train, as.numeric(y_train_s),
               epochs = 40, batch_size = 32, validation_split = 0.2, verbose = 0)

plot(history)
```



9.4 3. Recurrent Neural Network (RNN)

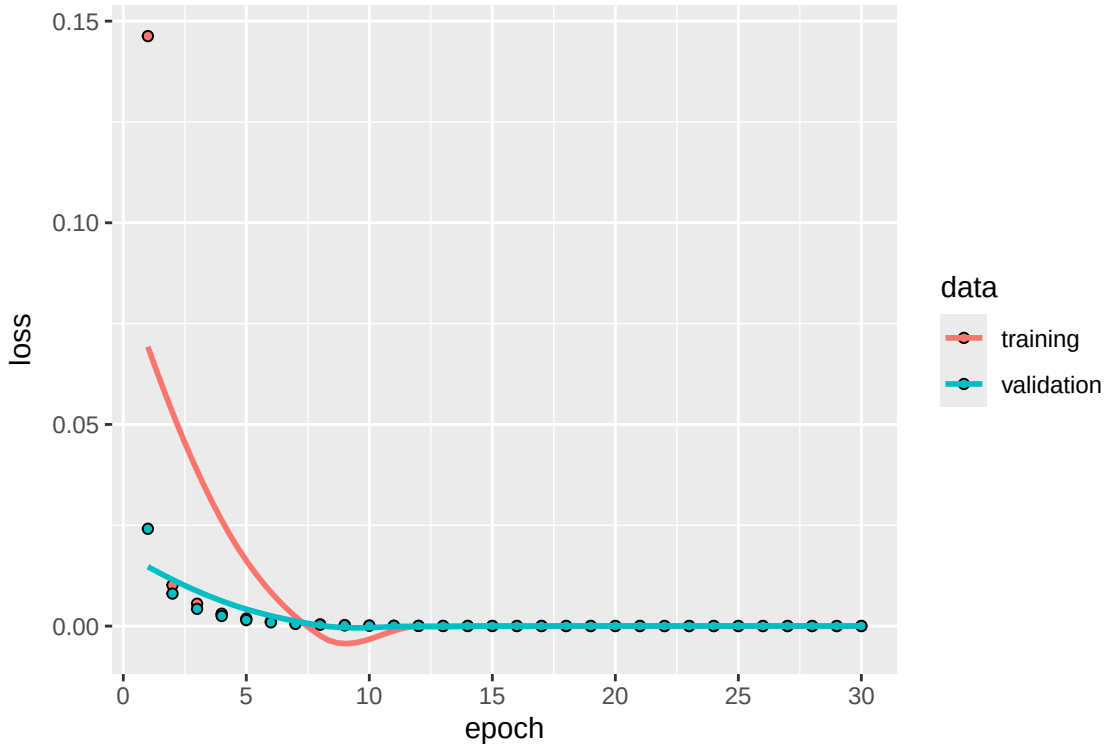
RNNs are ideal for **sequential data** such as time series, text, or speech.
We use a sine wave as a toy example because its repeating pattern requires the model to "remember" recent values — something feed-forward networks cannot do.

```
set.seed(42)
t <- seq(0, 50, by = 0.1)
y <- sin(t)
plot(t, y, type='l', main="Sine Wave Example", ylab="y", xlab="Time")
```




```
window_size <- 20
X <- sapply(1:(length(y) - window_size), function(i) y[i:(i + window_size - 1)])
X <- t(X)
y_out <- y[(window_size + 1):length(y)]
X <- array(X, dim = c(nrow(X), window_size, 1))

split <- round(0.8 * nrow(X))
x_train <- X[1:split,,]; y_train <- y_out[1:split]
```

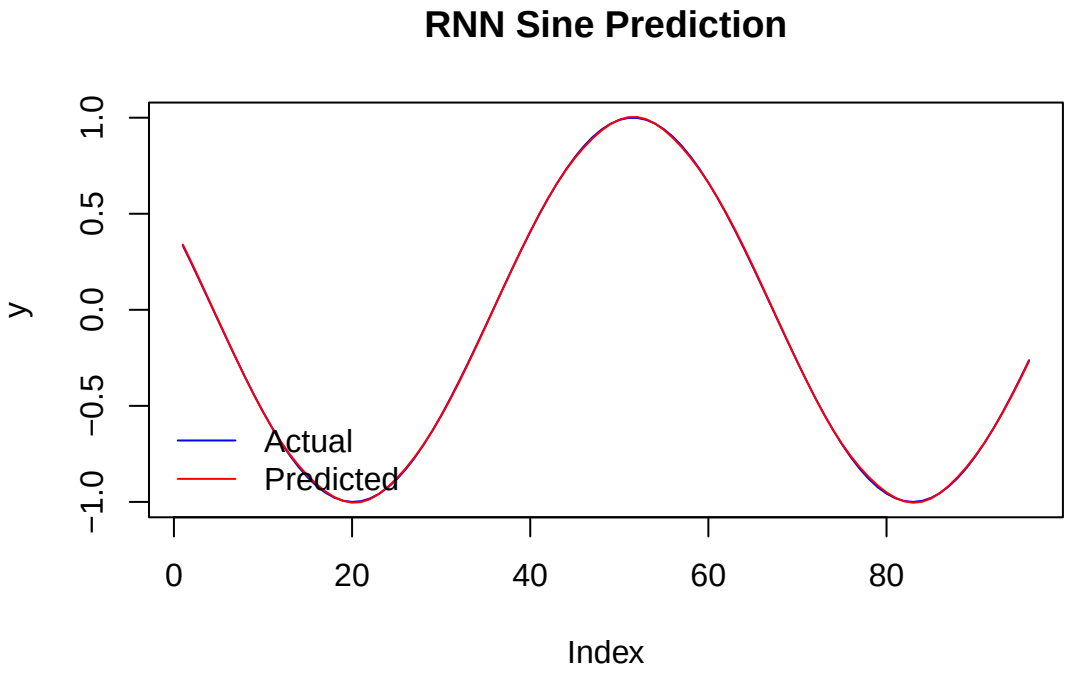
```
x_test  <- X[(split + 1):nrow(X),,]; y_test  <- y_out[(split + 1):length(y_out)]

model_rnn <- keras_model_sequential() |>
  layer_simple_rnn(units = 16, activation = "tanh", input_shape = c(window_size, 1)) |>
  layer_dense(units = 1)

compile(model_rnn, loss = "mse", optimizer = "adam")

history_rnn <- fit(model_rnn, x_train, y_train, epochs = 30, batch_size = 16,
                   validation_split = 0.2, verbose = 0)

plot(history_rnn)
```




```
preds <- predict(model_rnn, x_test)
```

```
3/3 - 0s - 11ms/step
```

```r
plot(y_test, type='l', main="RNN Sine Prediction", col='blue', ylab="y")
lines(preds, col='red')
legend("bottomleft", legend=c("Actual","Predicted"), col=c("blue","red"), lty=1, bty="n")
```



9.5 4. Convolutional Neural Network (CNN)

CNNs are built for **spatial pattern recognition** — ideal for images.
The **MNIST** dataset contains 70,000 grayscale images of handwritten digits (0–9), each 28×28 pixels.
These lines prepare the data for use in a CNN.

```r
#library(keras3)

# Load and preview the MNIST data
mnist <- dataset_mnist()
```

```r
# We only use a subset (5k train, 1k test) for quick demonstration
x_train <- mnist$train$x[1:5000,,]
y_train <- mnist$train$y[1:5000]
x_test  <- mnist$test$x[1:1000,,]
y_test  <- mnist$test$y[1:1000]

# Normalize pixel values: 0-255 -> 0-1
x_train <- x_train / 255
x_test  <- x_test / 255

# Reshape to [samples, height, width, channels]
# MNIST images are grayscale, so channels = 1
x_train <- array_reshape(x_train, c(nrow(x_train), 28, 28, 1))
x_test  <- array_reshape(x_test,  c(nrow(x_test),  28, 28, 1))

# Convert labels to categorical one-hot vectors (10 classes for digits 0-9)
y_train_cat <- to_categorical(y_train, 10)
y_test_cat  <- to_categorical(y_test, 10)
```

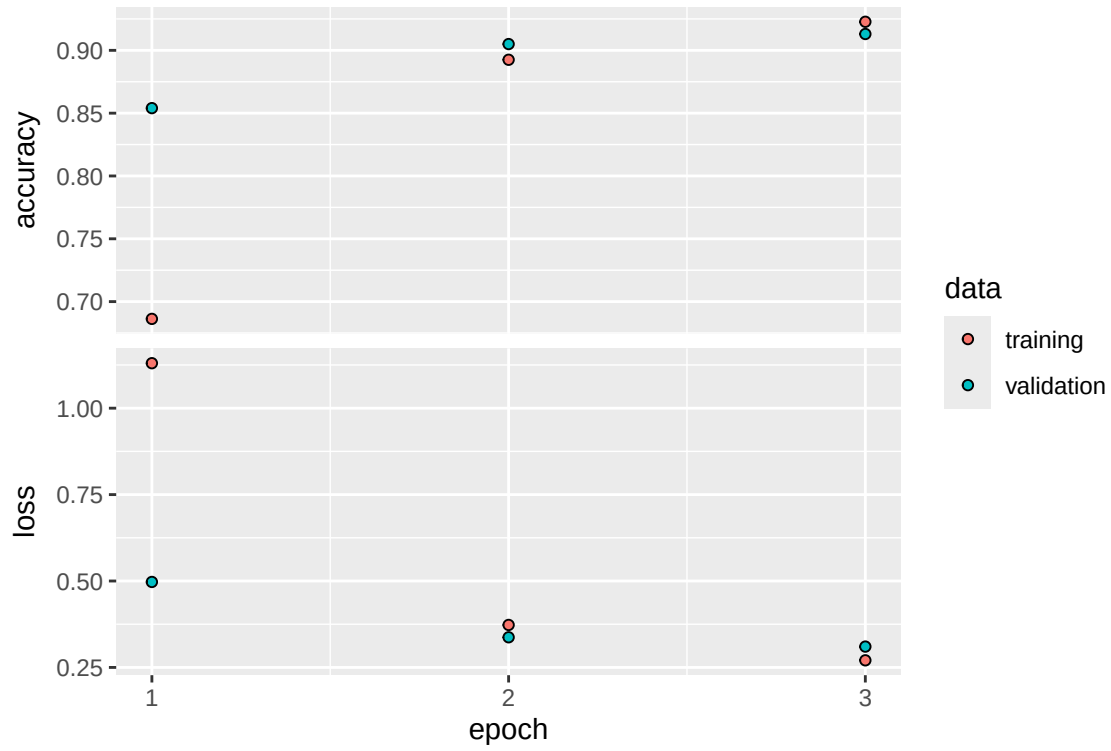
9.5.1 Explanation of the MNIST Preprocessing

1. **Load** the built-in dataset from Keras.
2. **Subsample** for faster training (5,000 train / 1,000 test).
3. **Normalize** pixel intensities to the 0–1 range.
4. **Reshape** to the 4D structure required by CNNs: `[samples, height, width, channels]`.
5. **One-hot encode** labels, converting a digit like `3` into a vector `[0,0,0,1,0,0,0,0,0,0]`.

These steps turn raw pixel data into numeric tensors suitable for learning.

9.5.2 4.1 Train and Evaluate the CNN

```
model_cnn <- keras_model_sequential() |>
  layer_conv_2d(filters = 16, kernel_size = c(3,3), activation = "relu", input_shape = c(28,
  layer_max_pooling_2d(pool_size = c(2,2)) |>
  layer_flatten() |>
  layer_dense(units = 32, activation = "relu") |>
  layer_dense(units = 10, activation = "softmax")

compile(model_cnn, optimizer = "adam",
        loss = "categorical_crossentropy",
        metrics = "accuracy")

history_cnn <- fit(model_cnn, x_train, y_train_cat,
                   epochs = 3, batch_size = 64, validation_split = 0.2, verbose = 0)

plot(history_cnn)
```




```
score <- evaluate(model_cnn, x_test, y_test_cat)
```

```
32/32 - 0s - 745us/step - accuracy: 0.9050 - loss: 0.3117
```

```
score
```

```
$accuracy
[1] 0.905

$loss
[1] 0.3116771
```

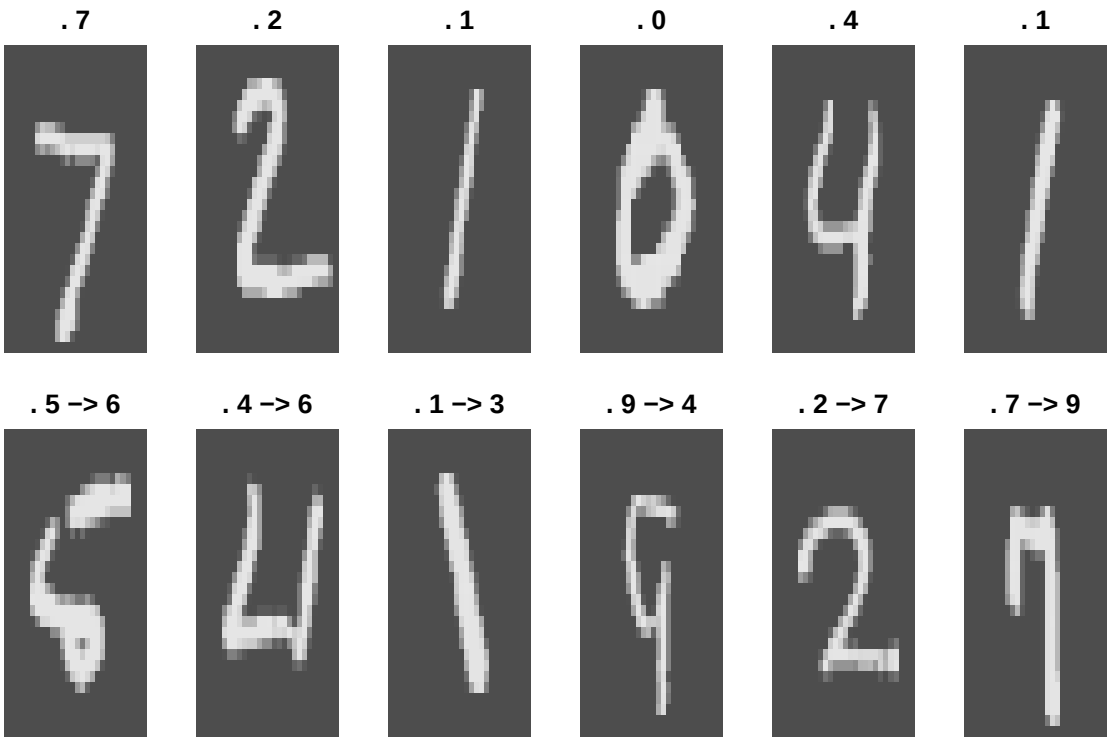
9.5.3 4.2 Visualizing Correct and Incorrect Predictions

We visualize examples of digits that the CNN classified correctly and incorrectly.

```
pred_probs <- predict(model_cnn, x_test)
```

```
32/32 - 0s - 1ms/step
```

```
pred_classes <- apply(pred_probs, 1, which.max) - 1

correct_idx <- which(pred_classes == y_test)[1:6]
incorrect_idx <- which(pred_classes != y_test)[1:6]

par(mfrow=c(2,6), mar=c(1,1,2,1))
for (i in correct_idx) {
  img <- x_test[i,,,1]
  image(1:28, 1:28, t(apply(img, 2, rev)), col=gray.colors(255), axes=FALSE,
        main=paste(" ", y_test[i]))
}
for (i in incorrect_idx) {
  img <- x_test[i,,,1]
  image(1:28, 1:28, t(apply(img, 2, rev)), col=gray.colors(255), axes=FALSE,
        main=paste(" ", y_test[i], "→", pred_classes[i]))
}
```



9.6 Summary

- **Feed-forward networks**: handle static, tabular data (e.g., housing prices).
- **RNNs**: handle sequential data with temporal dependencies (e.g., sine wave, stock prices, text, timer series).
- **CNNs**: handle image and spatial data by learning patterns like edges and textures.

10 Feedforward Neural Network - Python

10.1 Feedforward Neural Network (MLP) Using Ames Housing (Python)

This chapter fits a feed-forward neural network to the Ame Housing subset used in the R version.
RMSE is computed manually for compatibility with older scikit-learn versions.

10.1.1 1. Load and Prepare Data

```
import numpy as np
import pandas as pd

from sklearn.datasets import fetch_openml
from sklearn.model_selection import train_test_split, GridSearchCV
from sklearn.preprocessing import StandardScaler
from sklearn.neural_network import MLPRegressor
from sklearn.metrics import mean_squared_error

np.random.seed(12345)

ames = fetch_openml(name="house_prices", as_frame=True)
df = ames.frame

selected_cols = [
    "SalePrice","BedroomAbvGr","YearBuilt","MoSold","LotArea",
    "Street","CentralAir","1stFlrSF","2ndFlrSF","FullBath",
    "HalfBath","Fireplaces","GarageArea","GrLivArea","TotRmsAbvGrd"
]

df_sub = df[selected_cols].dropna()
```

```
df_sub = pd.get_dummies(df_sub, columns=["Street","CentralAir"], drop_first=True)

y = df_sub["SalePrice"]
X = df_sub.drop("SalePrice", axis=1)
```

10.1.2 2. Train/Test Split & Scaling

```
X_train, X_test, y_train, y_test = train_test_split(
    X, y, test_size=0.2, random_state=12345
)

scaler = StandardScaler()
scaler.fit(X_train)

X_train_s = scaler.transform(X_train)
X_test_s = scaler.transform(X_test)
```

10.1.3 3. Simple MLP Model

```
nn_simple = MLPRegressor(
    solver="lbfgs",
    alpha=1e-5,
    hidden_layer_sizes=(5,),
    random_state=12345,
    max_iter=5000
)

nn_simple.fit(X_train_s, y_train)
pred_simple = nn_simple.predict(X_test_s)

rmse_simple = mean_squared_error(y_test, pred_simple) ** 0.5
rmse_simple
```

```
38907.904754996634
```

10.1.4 4. Grid Search Tuning

```
param_grid = {
    "hidden_layer_sizes": [(h,) for h in [3,4,5,6,7]],
    "alpha": [0.00005,0.0005],
    "solver": ["lbfgs"]
}

nn_base = MLPRegressor(max_iter=5000, random_state=12345)

grid = GridSearchCV(
    estimator=nn_base,
    param_grid=param_grid,
    cv=10,
    scoring="neg_mean_squared_error",
    n_jobs=-1
)

grid.fit(X_train_s, y_train)

grid.best_params_
```

```
{'alpha': 5e-05, 'hidden_layer_sizes': (5,), 'solver': 'lbfgs'}
```

```
best_model = grid.best_estimator_
pred_gs = best_model.predict(X_test_s)

rmse_gs = mean_squared_error(y_test, pred_gs) ** 0.5
rmse_gs
```

```
39176.951700141195
```

11 Naive Bayes Classification R version

11.1 1. Overview and Learning Objectives

In this chapter, we will:

1. Use the **Ames Housing** data to create a **binary classification** problem:
    - Predict whether a house is **high price** vs **low price** based on its features..
2. Understand how Naive Bayes handles:
    - **Categorical vs numeric** predictors
    - **Laplace correction** (additive smoothing)
    - **Kernel density estimation** for numeric variables
3. Use **caret** (`train(method = "nb")`) to:
    - **Tune** key hyperparameters (`usekernel`, `fL`)
    - Evaluate performance with **cross-validation**
4. Interpret:
    - Class priors
    - Conditional feature distributions
    - Predictions & confusion matrix
    - ROC curve and AUC

11.2 2. Data Setup and Binary Target Creation

```
library(tidyverse)
library(caret)
library(AmesHousing)
library(pROC)

set.seed(4321)

ames <- make_ordinal_ames() %>% mutate(id = row_number())

training <- ames %>% sample_frac(0.7)
testing  <- anti_join(ames, training, by = "id")

predictors <- c(
  "Sale_Price","Bedroom_AbvGr","Year_Built","Mo_Sold","Lot_Area",
  "Street","Central_Air","First_Flr_SF","Second_Flr_SF","Full_Bath",
  "Half_Bath","Fireplaces","Garage_Area","Gr_Liv_Area","TotRms_AbvGrd"
)

training <- training %>% select(all_of(predictors))
testing  <- testing %>% select(all_of(predictors))
```

We turn `Sale_Price` into a **classification target**:

- `"High"` if above the training-set median
- `"Low"` otherwise

This keeps the task aligned with Naive Bayes as a **classification** model.

```
price_median <- median(training$Sale_Price)

training <- training %>% mutate(Price_High = factor(ifelse(Sale_Price > price_median,"High","
testing  <- testing  %>% mutate(Price_High = factor(ifelse(Sale_Price > price_median,"High","

training_nb <- training %>% select(-Sale_Price)
testing_nb  <- testing  %>% select(-Sale_Price)
```

11.3 3. Naive Bayes Intuition

Naive Bayes is a **generative classifier** based on Bayes' theorem:

[P(Y = k X) P(Y=k) _{j=1}^p P(x_j Y=k)]

Key assumptions:

11.3.0.1 1. Conditional Independence

Given the class, features are treated as independent:

[P(x_1, x_2, ..., x_p Y=k) _j P(x_j Y=k)]

This is rarely exactly true, yet often works surprisingly well.

11.3.0.2 2. Modeling Numeric Features

Naive Bayes supports two approaches:

- **Gaussian (Normal):** assumes each numeric feature is normally distributed within each class.
- **Kernel Density Estimation:** more flexible, handles skewness and multimodal distributions.

11.3.0.3 3. Laplace Correction

Prevents zero probabilities when a category never appears in a given class:

''text P(x=v | Y=k) = (count + 1) / (N + K)'''

This keeps the model from collapsing.

11.4 4. Fitting Naive Bayes (Baseline Model)

```r
set.seed(12345)

nb.ames <- naiveBayes(
  Price_High ~ .,
  data = training_nb,
  laplace = 0,
  usekernel = TRUE
)

nb.ames
```

```
Naive Bayes Classifier for Discrete Predictors

Call:
naiveBayes.default(x = X, y = Y, laplace = laplace, usekernel = TRUE)

A-priori probabilities:
Y
     High       Low
0.4987811 0.5012189

Conditional probabilities:
      Bedroom_AbvGr
Y          [,1]      [,2]
  High 2.948192 0.8218245
  Low  2.770428 0.8207514

      Year_Built
Y          [,1]     [,2]
  High 1989.604 23.12763
  Low  1953.780 25.88135

      Mo_Sold
Y          [,1]     [,2]
  High 6.353861 2.732694
  Low  6.157588 2.691989

      Lot_Area
Y           [,1]      [,2]
  High 11726.913 10345.651
  Low   8673.011  4673.732
```

```
      Street
Y              Grvl         Pave
  High 0.0009775171 0.9990224829
  Low  0.0058365759 0.9941634241

      Central_Air
Y                N           Y
  High 0.008797654 0.991202346
  Low  0.122568093 0.877431907

      First_Flr_SF
Y           [,1]     [,2]
  High 1333.7087 408.4059
  Low   986.6527 264.8090

      Second_Flr_SF
Y          [,1]     [,2]
  High 447.7625 493.0434
  Low  222.7490 317.7090

      Full_Bath
Y          [,1]      [,2]
  High 1.913001 0.4415516
  Low  1.233463 0.4412614

      Half_Bath
Y           [,1]      [,2]
  High 0.5259042 0.5168999
  Low  0.2324903 0.4406718

      Fireplaces
Y           [,1]      [,2]
  High 0.8670577 0.6271108
  Low  0.3180934 0.5359368

      Garage_Area
Y          [,1]     [,2]
  High 581.0430 173.7891
  Low  362.3016 190.0580

      Gr_Liv_Area
Y          [,1]     [,2]
```

```
  High 1783.837 473.5654
  Low  1215.255 342.4179

     TotRms_AbvGrd
Y          [,1]     [,2]
  High 7.043988 1.516711
  Low  5.825875 1.373640
```

11.4.0.1 Explanation of key settings

- **laplace = 0**
  No smoothing; probability estimates reflect raw counts.
  Zero probabilities may occur for rare categorical levels.
- **usekernel = TRUE**
  Numeric features are modeled using **kernel density estimates** instead of assuming normality.

11.4.0.2 What the output means

- **A-priori probabilities:**
  Estimated from class frequencies (High vs Low).
- **Conditional tables:**
  - For categorical features: probability of each level given class.
  - For numerical features: kernel-based density info (not printed in summary).

11.5 5. Predicting on Test Data

```r
test_pred_class <- predict(nb.ames, testing_nb)
test_pred_prob  <- predict(nb.ames, testing_nb, type="raw")
head(test_pred_prob)
```

```
          High          Low
[1,] 0.9999965 3.462116e-06
[2,] 0.9850961 1.490391e-02
[3,] 0.9830462 1.695380e-02
[4,] 0.5936144 4.063856e-01
[5,] 0.9437367 5.626333e-02
[6,] 0.9998253 1.746816e-04
```

- `test_pred_class`: predicted High/Low label
- `test_pred_prob`: estimated probability for each class

11.6 6. Confusion Matrix Interpretation

```r
conf_matrix <- confusionMatrix(
  test_pred_class,
  testing_nb$Price_High,
  positive="High"
)

conf_matrix
```

```
Confusion Matrix and Statistics

          Reference
Prediction High Low
      High  373  49
      Low    55 402

               Accuracy : 0.8817
                 95% CI : (0.8585, 0.9023)
    No Information Rate : 0.5131
    P-Value [Acc > NIR] : <2e-16

                  Kappa : 0.7631

 Mcnemar's Test P-Value : 0.6239
```

```
            Sensitivity : 0.8715
            Specificity : 0.8914
         Pos Pred Value : 0.8839
         Neg Pred Value : 0.8796
             Prevalence : 0.4869
         Detection Rate : 0.4243
   Detection Prevalence : 0.4801
      Balanced Accuracy : 0.8814

       'Positive' Class : High
```

11.6.0.1 What the confusion matrix tells us

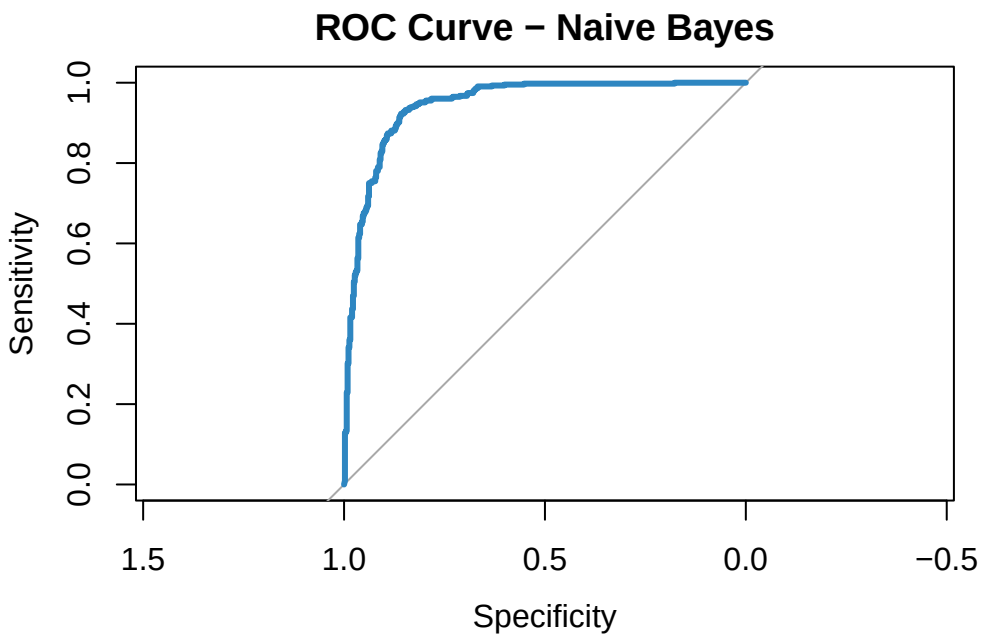
- **Accuracy**: overall correctness
- **Sensitivity (Recall for High)**: proportion of expensive homes correctly identified
- **Specificity (Recall for Low)**: proportion of cheaper homes correctly identified
- **Balanced Accuracy**: average of sensitivity & specificity
- **Kappa**: agreement beyond chance

This is the primary evaluation for classification problems.

11.7 7. ROC Curve & AUC

ROC curves evaluate model ranking ability — how well it separates High from Low across thresholds.

```
roc_obj <- roc(
  response = testing_nb$Price_High,
  predictor = test_pred_prob[,"High"],
  levels = c("Low","High")
)

plot(roc_obj, col="#2E86C1", lwd=3, main="ROC Curve - Naive Bayes")
```




```
auc_value <- auc(roc_obj)
auc_value
```

```
Area under the curve: 0.9457
```

11.7.0.1 Interpretation

- **ROC Curve:** visual representation of sensitivity vs (1 - specificity)
- **AUC:** single summary metric
  - 0.5 = no better than random
  - 0.7 = decent
  - 0.8 = strong
  - 0.9 = outstanding

11.8 8. Tuning Naive Bayes Using `caret::train()`

We now tune:

- `usekernel` {TRUE, FALSE}
- `fL` {0, 0.5, 1} *(fractional Laplace)*

```
set.seed(12345)

tune_grid <- expand.grid(
  usekernel = c(TRUE, FALSE),
  fL        = c(0, 0.5, 1),
  adjust    = 1
)

ctrl <- trainControl(
  method = "cv",
  number = 10,
  classProbs = TRUE,
  summaryFunction = twoClassSummary
)

training_nb$Price_High <- relevel(training_nb$Price_High, ref="High")
testing_nb$Price_High  <- relevel(testing_nb$Price_High, ref="High")

nb.ames.caret <- train(
  Price_High ~ .,
  data = training_nb,
  method = "nb",
  tuneGrid = tune_grid,
  trControl = ctrl,
  metric ="ROC"
)

nb.ames.caret
```

```
Naive Bayes

2051 samples
  14 predictor
   2 classes: 'High', 'Low'
```

```
No pre-processing
Resampling: Cross-Validated (10 fold)
Summary of sample sizes: 1845, 1846, 1846, 1846, 1846, 1845, ...
Resampling results across tuning parameters:

  usekernel  fL   ROC        Sens       Spec
  FALSE      0.0  0.9325587  0.9217484  0.8453794
  FALSE      0.5  0.9325587  0.9217484  0.8453794
  FALSE      1.0  0.9325587  0.9217484  0.8453794
   TRUE      0.0  0.9549057  0.9197316  0.8589473
   TRUE      0.5  0.9549057  0.9197316  0.8589473
   TRUE      1.0  0.9549057  0.9197316  0.8589473

Tuning parameter 'adjust' was held constant at a value of 1
ROC was used to select the optimal model using the largest value.
The final values used for the model were fL = 0, usekernel = TRUE and adjust
 = 1.
```

###$ How tuning works

- **Cross-validation** evaluates each combination of hyperparameters.
- Best model chosen by the metric (`ROC`).
- caret automatically handles class probabilities and ROC scoring.

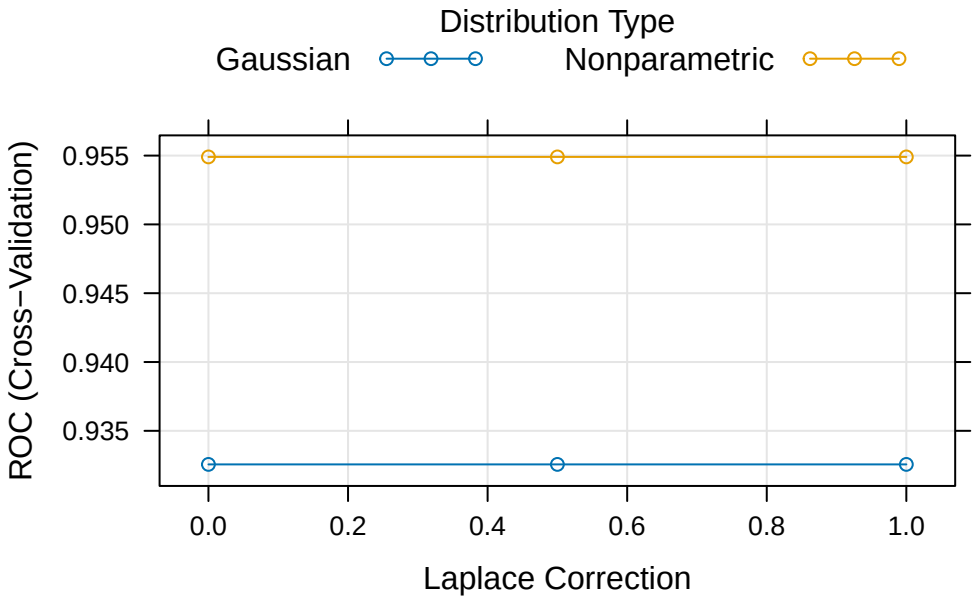
```
nb.ames.caret$results
```

```
  usekernel  fL adjust       ROC      Sens      Spec      ROCSD     SensSD
1     FALSE 0.0      1 0.9325587 0.9217484 0.8453794 0.02290800 0.03508486
4      TRUE 0.0      1 0.9549057 0.9197316 0.8589473 0.01199907 0.04510391
2     FALSE 0.5      1 0.9325587 0.9217484 0.8453794 0.02290800 0.03508486
5      TRUE 0.5      1 0.9549057 0.9197316 0.8589473 0.01199907 0.04510391
3     FALSE 1.0      1 0.9325587 0.9217484 0.8453794 0.02290800 0.03508486
6      TRUE 1.0      1 0.9549057 0.9197316 0.8589473 0.01199907 0.04510391
      SpecSD
1 0.02659457
4 0.03971772
2 0.02659457
5 0.03971772
3 0.02659457
6 0.03971772
```

```
nb.ames.caret$bestTune
```

```
  fL usekernel adjust
4  0      TRUE      1
```

```
plot(nb.ames.caret)
```



11.9 9. Evaluating the Tuned Model on the Test Set

```
test_pred_class_tuned <- predict(nb.ames.caret, testing_nb)
test_pred_prob_tuned <- predict(nb.ames.caret, testing_nb, type="prob")

confusionMatrix(
  test_pred_class_tuned,
  testing_nb$Price_High,
  positive="High"
)
```

```
Confusion Matrix and Statistics

          Reference
Prediction High Low
      High  398  77
      Low    30 374

               Accuracy : 0.8783
                 95% CI : (0.8548, 0.8992)
    No Information Rate : 0.5131
    P-Value [Acc > NIR] : < 2.2e-16

                  Kappa : 0.7571

 Mcnemar's Test P-Value : 8.708e-06

            Sensitivity : 0.9299
            Specificity : 0.8293
         Pos Pred Value : 0.8379
         Neg Pred Value : 0.9257
             Prevalence : 0.4869
         Detection Rate : 0.4528
   Detection Prevalence : 0.5404
      Balanced Accuracy : 0.8796

       'Positive' Class : High
```

11.10 10. Summary

- Naive Bayes is a **probabilistic, generative classifier**.
- It models **class priors** + **conditional likelihoods** for each predictor.
- Numeric predictors can use:
  - **Gaussian distributions**, or
  - **Kernel density estimates** (`usekernel = TRUE`).

- **Laplace smoothing** protects against zero probabilities.
- **ROC** and **AUC** evaluate ranking performance.
- **Confusion matrix** evaluates classification performance.
- **caret** enables systematic tuning over smoothing and density options.

12 Naive Bayes Classification - Python Version

12.1 1. Overview and Learning Objectives

In this Python chapter, we will:

1. Load the **Ames Housing** dataset from OpenML.
2. Create a **High vs Low** price classification target.
3. Fit **Gaussian Naive Bayes**.
4. Produce predictions and class probabilities.
5. Compute a **correct ROC curve and AUC**.
6. Mirror the structure of the R version.

12.2 2. Setup and Data Preparation

```
import pandas as pd
import numpy as np
from sklearn.model_selection import train_test_split
from sklearn.naive_bayes import GaussianNB
from sklearn.metrics import confusion_matrix, classification_report, roc_curve, auc
import matplotlib.pyplot as plt
import seaborn as sns
from sklearn.datasets import fetch_openml

# Load Ames Housing dataset
ames = fetch_openml(name="house_prices", as_frame=True)
df = ames.frame.copy()

df.rename(columns={"SalePrice": "Sale_Price"}, inplace=True)

df.head()
```

	Id	MSSubClass	MSZoning	LotFrontage	LotArea	Street	Alley	LotShape	LandContour	Utilit
0	1	60	RL	65.0	8450	Pave	NaN	Reg	Lvl	AllPu
1	2	20	RL	80.0	9600	Pave	NaN	Reg	Lvl	AllPu
2	3	60	RL	68.0	11250	Pave	NaN	IR1	Lvl	AllPu
3	4	70	RL	60.0	9550	Pave	NaN	IR1	Lvl	AllPu
4	5	60	RL	84.0	14260	Pave	NaN	IR1	Lvl	AllPu

12.3 2.1 Select Variables of Interest

```
predictors = [
    "Sale_Price",
    "BedroomAbvGr",
    "YearBuilt",
    "MoSold",
    "LotArea",
    "Street",
    "CentralAir",
    "1stFlrSF",
    "2ndFlrSF",
    "FullBath",
    "HalfBath",
    "Fireplaces",
    "GarageArea",
    "GrLivArea",
    "TotRmsAbvGrd"
]

df = df[predictors].dropna()
```

12.4 2.2 Create a Binary Target Variable

```
median_price = df["Sale_Price"].median()

df["Price_High"] = np.where(df["Sale_Price"] > median_price, "High", "Low")
df["Price_High"] = df["Price_High"].astype("category")

df["Price_High"].value_counts(normalize=True)
```

```
Price_High
Low     0.50137
High    0.49863
Name: proportion, dtype: float64
```

12.5 2.3 Train/Test Split

```
X = df.drop(columns=["Sale_Price", "Price_High"])
y = df["Price_High"]

# Encode categorical predictors
X = pd.get_dummies(X, drop_first=True)

X_train, X_test, y_train, y_test = train_test_split(
    X, y, test_size=0.30, random_state=4321, stratify=y
)

X_train.shape, X_test.shape
```

```
((1022, 14), (438, 14))
```

12.6 3. Gaussian Naive Bayes Model

```
gnb = GaussianNB()
gnb.fit(X_train, y_train)
```

```
GaussianNB()
```

12.7 4. Predictions

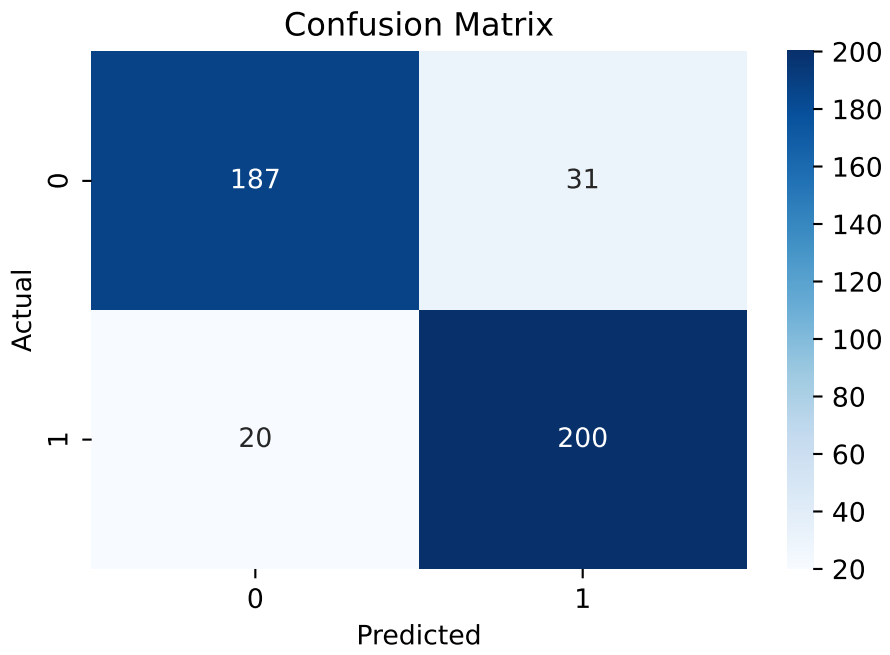
```
y_pred = gnb.predict(X_test)

# Extract the probability of the TRUE positive class ("High")
pos_index = np.where(gnb.classes_ == "High")[0][0]
y_prob_high = gnb.predict_proba(X_test)[:, pos_index]
```

12.8 5. Confusion Matrix & Classification Report

```
cm = confusion_matrix(y_test, y_pred)
cm
```

```
array([[187,  31],
       [ 20, 200]])
```

```
sns.heatmap(cm, annot=True, fmt="d", cmap="Blues")
plt.title("Confusion Matrix")
plt.xlabel("Predicted")
plt.ylabel("Actual")
plt.show()
```




```
print(classification_report(y_test, y_pred))
```

```
              precision    recall  f1-score   support

        High       0.90      0.86      0.88       218
         Low       0.87      0.91      0.89       220

    accuracy                           0.88       438
   macro avg       0.88      0.88      0.88       438
weighted avg       0.88      0.88      0.88       438
```

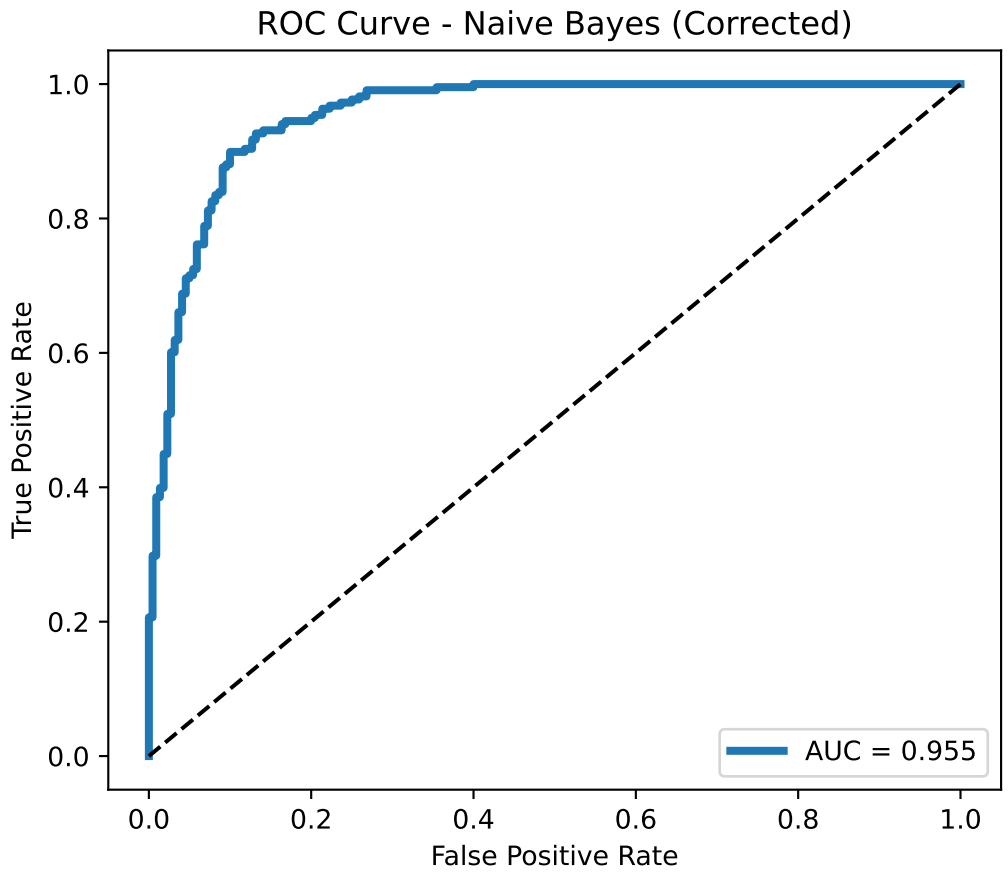
12.9 6. ROC Curve (Corrected) and AUC

```
# Convert y_test to binary (1 = High, 0 = Low)
y_test_binary = (y_test == "High").astype(int)

# Compute ROC using correct positive class probability
```

```
fpr, tpr, thresholds = roc_curve(y_test_binary, y_prob_high)
roc_auc = auc(fpr, tpr)

roc_auc
```

```
np.float64(0.9547122602168474)
```

```
plt.figure(figsize=(6,5))
plt.plot(fpr, tpr, label=f"AUC = {roc_auc:.3f}", linewidth=3)
plt.plot([0,1], [0,1], "k--")
plt.xlabel("False Positive Rate")
plt.ylabel("True Positive Rate")
plt.title("ROC Curve - Naive Bayes (Corrected)")
plt.legend()
plt.show()
```



12.10 7. Interpretation Notes

12.10.1 Naive Bayes Assumptions

- Predictors are independent given the class.
- Numeric features follow Gaussian distributions.
- One-hot encoded categorical variables behave like Bernoulli variables.

12.10.2 Strengths

- Fast and simple.
- Works well as a baseline.
- Highly interpretable.

12.10.3 Weaknesses

- Independence assumption rarely holds.
- Gaussian assumption is an approximation.

12.11 8. Summary

This chapter:

- Loaded the Ames Housing dataset.
- Created a binary classification target.
- Fit Gaussian Naive Bayes.
- Correctly computed ROC & AUC.
- Provided fully aligned interpretation with the R version.

13 model-agnostic-interpretability

14 support-vector-machines