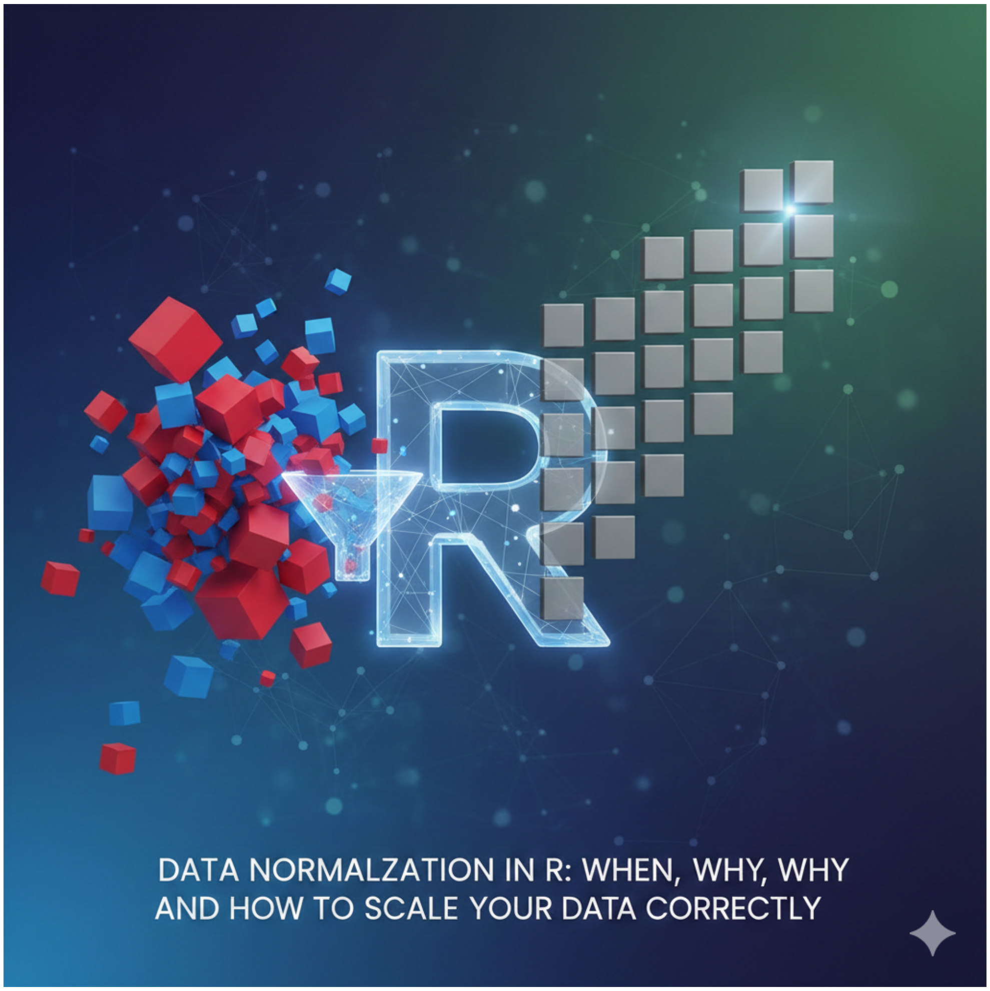
Data Normalization in R: When, Why, and How to Scale Your Data Correctly

Table of contents


DATA NORMALZATION IN R: WHEN, WHY, WHY
AND HOW TO SCALE YOUR DATA CORRECTLY

Introduction

This article is part of a broader series on **data preprocessing in R**. In earlier posts, we focused on two problems that quietly ruin analyses long before modeling begins: **missing data** and **outliers**. Both topics shared a common theme: preprocessing choices are not cosmetic; they change what the model is allowed to learn. In this installment, we move to the next decision point in the same pipeline: **normalization (scaling)**—often treated as "just a quick step," but in practice a decisive modeling choice.

Related posts in this preprocessing series

- *Handling Missing Data in R: A Comprehensive Guide*
  https://medium.com/r-evolution/handling-missing-data-in-r-a-comprehensive-guide-eca195eaead3

- *Outliers in Data Analysis: Detecting Extreme Values Before Modeling in R* https://medium.com/r-evolution/outliers-in-data-analysis-detecting-extreme-values-before-modeling-in-r-with-i%CC%87stanbul-airbnb-data-3b37e9ee989e

Normalization (or more broadly, scaling) is frequently presented as a minor technical adjustment—something to apply quickly and forget. In practice, scaling is not a technical detail but a modeling decision. When the same dataset is processed using different scaling strategies, the behavior of many models changes substantially. Distances, similarity measures, penalty terms, and optimization paths are all affected. As a result, the nearest neighbors selected by KNN, the clusters formed by K-means, the principal components identified by PCA, and even the coefficients chosen by Ridge or Lasso regression can differ. Scaling does not merely "prepare" the data; it actively shapes how a model interprets importance and structure.

More importantly, scaling is not universally beneficial. Applied in the wrong context, it can degrade model performance or—worse—introduce subtle forms of **data leakage** that contaminate evaluation. A common example is learning scaling parameters (such as means and standard deviations) from the entire dataset before splitting into training and test sets. This procedure allows information from the test distribution to leak into the training process, producing performance estimates that cannot be trusted. In such cases, the issue is not the scaling method itself, but **when and how** it is applied. Knowing how to call `scale()` in R is trivial; understanding what to scale, when to scale it, and why is not.

In this article, normalization is treated as an integral part of the modeling strategy rather than a routine preprocessing step. We will address, step by step, the following questions:

- Why is normalization necessary?
- Should it always be applied?
- At what stage should it be performed—before or after the train–test split?
- Which scaling methods are commonly used, and in which contexts do they make sense?
- Should different data types be treated differently?
- Is scaling appropriate for all variables, including the target variable?

By combining conceptual discussion with practical R implementations, this guide aims to provide clear and principled answers to each of these questions.

Normalization vs. Standardization: Clearing Up the Terminology

In both academic writing and everyday practice, the terms *normalization* and *standardization* are frequently used interchangeably. This loose usage is one of the main sources of confusion in data preprocessing. In reality, these terms refer to **different scaling strategies**, each with distinct assumptions, effects, and use cases. Before discussing when and how scaling should be applied, it is therefore essential to clarify what is actually meant by each approach.

Standardization, often referred to as *z-score scaling*, rescales a variable so that it has a mean of zero and a standard deviation of one. Formally, each observation is transformed by subtracting the sample mean and dividing by the sample standard deviation. In the R ecosystem, this logic is implemented in preprocessing tools such as `step_normalize()` from the **recipes** package. Standardization preserves the shape of the original distribution while putting variables on a comparable scale. It is particularly useful for models that are sensitive to the relative magnitude of predictors, such as linear models with regularization, support vector machines, and neural networks.

Normalization, in a stricter sense, often refers to *min–max scaling*. This approach rescales variables to lie within a fixed interval, most commonly [0,1]. Each value is transformed based on the minimum and maximum observed in the training data. Min–max scaling is easy to interpret and is frequently used in algorithms where bounded inputs are desirable. However, it is also more sensitive to extreme values, since a single outlier can heavily influence the scaling range.

A third commonly used approach is **robust scaling**, which relies on the median and the interquartile range (IQR) instead of the mean and standard deviation. By construction, this method is less affected by outliers and heavy-tailed distributions. Robust scaling is especially useful in real-world datasets where extreme values are not errors but genuine observations. At the same time, it is not a universal solution; in some data structures, robust measures may become unstable or uninformative.

The reason terminology becomes blurred in practice is simple: many practitioners use the word *normalization* as a generic label for "any kind of scaling." As a result, two people may both say they normalized their data while having applied entirely different transformations. Throughout this article, we will avoid this ambiguity by explicitly stating which scaling method is used and why. This distinction is not pedantic—it is essential for understanding how scaling choices influence model behavior.

Why Is Normalization Necessary?

The necessity of normalization becomes clear once we recognize that many modeling techniques do not operate on raw variable values directly, but on **relationships derived from them**—such as distances, similarities, penalties, or variance directions. When predictors are measured on different scales, these derived quantities can be dominated by variables with larger numerical ranges, regardless of their substantive importance. In such cases, the model does not learn from the data structure itself, but from arbitrary measurement units.

This issue is most apparent in **distance-based methods** such as k-nearest neighbors (KNN) and K-means clustering. These algorithms rely explicitly on distance calculations, typically Euclidean distance. If one variable ranges between 0 and 1 while another ranges between 0 and 10,000, the latter will dominate the distance computation almost entirely. As a result, proximity is determined not by overall similarity but by the scale of a single variable. Normalization ensures that each predictor contributes to the distance metric in a balanced and interpretable way, allowing the algorithm to reflect genuine similarity rather than numerical magnitude.

Normalization is equally critical in models that incorporate **regularization**, such as Ridge and Lasso regression. In these models, coefficients are penalized to control model complexity. However, the penalty term is directly tied to the scale of the predictors. If variables are not on comparable scales, the regularization mechanism will shrink coefficients unevenly, effectively penalizing some predictors more than others for reasons unrelated to their predictive relevance. Scaling aligns the predictors so that regularization operates as intended: as a constraint on model complexity rather than an artifact of measurement units.

Other widely used techniques—including **support vector machines (SVMs)**, **neural networks**, and **principal component analysis (PCA)**—are also highly sensitive to scaling. In SVMs and neural networks, optimization procedures depend on gradients that are influenced by feature magnitudes, affecting both convergence speed and stability. In PCA, the directions of maximum variance are determined by the scale of the variables; without normalization, components may simply reflect variables with the largest variances rather than the most informative underlying structure. In all these cases, scaling is not an optional refinement but a prerequisite for meaningful model behavior.

By contrast, **tree-based models** such as decision trees, random forests, and gradient boosting machines are generally invariant to monotonic transformations of individual predictors. Since splits are based on ordering rather than distance or magnitude, scaling is often unnecessary for these methods. Nevertheless, this does not imply that normalization is universally irrelevant in tree-based pipelines. Hybrid workflows—where tree-based models are combined with distance-based components, rule-based similarity measures, or downstream models sensitive to scale—may still require careful consideration of scaling choices. The key point is not that normalization should always be applied, but that it should be applied **with respect to the assumptions of the modeling approach**.

From a broader perspective, normalization plays a central role in modern predictive modeling workflows. As emphasized in the predictive modeling literature, preprocessing steps are not independent of the model; they are part of the modeling strategy itself. Scaling decisions shape how information is represented and, ultimately, how learning takes place. Understanding *why* normalization is necessary is therefore a prerequisite for deciding *when* and *how* it should be applied—a topic we address next.

Should Normalization Always Be Applied?

A natural question at this point is whether normalization should be applied by default in every modeling task. The short answer is **no**. Normalization is not a universally beneficial preprocessing step; its usefulness depends on the assumptions and internal mechanics of the chosen model. Applying scaling blindly can be as problematic as ignoring it altogether. What is needed is a **decision framework** that links model characteristics to preprocessing choices.

For a large class of models, normalization is **strongly recommended**. This group includes distance-based methods such as k-nearest neighbors (KNN) and K-means clustering, as well as techniques like principal component analysis (PCA), support vector machines (SVMs), neural networks, and penalized regression models (Ridge, Lasso, Elastic Net). In all these cases, either distances, inner products, variance directions, or penalty terms play a central role. Without scaling, these mechanisms are dominated by variables with larger numerical ranges, leading to distorted learning behavior. For such models, normalization is not a refinement but a prerequisite for meaningful results.

By contrast, normalization is **generally unnecessary** for tree-based models such as decision trees, random forests, and gradient boosting machines (e.g., XGBoost, GBM). These models rely on recursive binary splits based on variable ordering rather than on distances or magnitudes. Since monotonic transformations do not affect the relative ordering of values, scaling typically has no impact on model performance. As a result, normalization is often omitted in purely tree-based pipelines without any loss of effectiveness.

Between these two extremes lies a set of models for which normalization is **context-dependent**. Ordinary linear regression, for example, does not require scaling for estimation itself, but normalization may still be useful for numerical stability, interpretability of coefficients, or comparability across predictors. Similarly, Naive Bayes models may or may not benefit from scaling depending on the assumed feature distributions and the types of variables involved. In these cases, the decision to normalize should be guided by the modeling objective rather than by a fixed rule.

The key takeaway is that normalization should be applied **with respect to the model's assumptions**, not as a default preprocessing habit. To make this decision explicit, Table 1 summarizes common modeling approaches and whether normalization is typically required.

When Is Normalization Needed? A Model-Based Decision Table

Model / Method	Is Normalization Recommended?	Rationale
KNN	Yes	Distance calculations are scale-sensitive
K-means	Yes	Cluster assignment depends on distances
PCA	Yes	Variance directions dominated by scale
SVM	Yes	Optimization and margins depend on feature magnitude
Neural Networks	Yes	Gradient-based optimization is scale-sensitive
Ridge / Lasso / Elastic Net	Yes	Penalty terms depend on predictor scale
Linear Regression (OLS)	Depends	Not required for estimation, but useful for stability and interpretation

Model / Method	Is Normalization Recommended?	Rationale
Naive Bayes	Depends	Depends on feature types and distributional assumptions
Decision Trees	No	Split rules depend on ordering, not scale
Random Forest / GBM / XGBoost	No	Tree-based structure is scale-invariant

When Should Normalization Be Applied? Before or After the Train–Test Split?

This is the most critical question in the entire preprocessing workflow—and the point at which many otherwise sound analyses quietly go wrong. The issue is not *whether* normalization should be applied, but **when** it should be applied. At the center of this question lies a fundamental concept in predictive modeling: **data leakage**.

Data leakage occurs when information from outside the training set is used, directly or indirectly, during model training. In the context of normalization, leakage typically arises when scaling parameters—such as means and standard deviations (for standardization) or minimum and maximum values (for min–max scaling)—are estimated using the full dataset before splitting into training and test sets. Although this may appear harmless, it allows information from the test set to influence the preprocessing step, leading to overly optimistic performance estimates.

The correct principle is straightforward but non-negotiable:
scaling parameters must be learned exclusively from the training data.
Once learned, the *same transformation*—with fixed parameters—must be applied to the test set and to any future, unseen data. This ensures that the test set truly represents new information and that model evaluation reflects genuine generalization rather than procedural artifacts.

This principle is central to modern modeling frameworks. In the **tidymodels/recipes** philosophy, preprocessing steps are *trained* on the training data and then *applied* consistently to all other datasets. Similarly, in the **caret** framework, preprocessing transformations are estimated from the training set and reused when predicting on new data. In both cases, preprocessing is treated as part of the model training process—not as an independent, preliminary operation.

To see why this distinction matters, consider the following conceptual comparison.

An Illustrative Example: Scaling Before vs. After the Split

Suppose we have a dataset that we intend to split into training and test sets. We want to standardize a numeric predictor using z-score scaling.

Incorrect approach (scaling before the split):

1. Compute the mean and standard deviation using the *entire dataset*.
2. Standardize all observations using these global parameters.
3. Split the scaled data into training and test sets.
4. Train and evaluate the model.

At first glance, this workflow seems efficient. However, the scaling parameters already incorporate information from the test set. The test data are no longer independent of the training process, even though they were not explicitly used to fit the model.

Correct approach (scaling after the split):

1. Split the raw data into training and test sets.
2. Compute scaling parameters (mean, standard deviation, etc.) *using only the training set*.
3. Apply the learned transformation to the training set.
4. Apply the *same* transformation to the test set.
5. Train the model on the scaled training data and evaluate it on the scaled test data.

In practice, these two approaches can lead to noticeably different evaluation results. Models trained using the incorrect workflow often appear to perform better on the test set—not because they generalize better, but because the preprocessing step has already "seen" the test data. This difference is especially pronounced in smaller datasets, in datasets with strong distributional differences between training and test splits, or when extreme values are present.

The takeaway is unambiguous:

> **Split the data first.**
> **Fit preprocessing steps on the training data.**
> **Apply the same transformations to the training and test sets.**

Any deviation from this sequence undermines the validity of model evaluation, regardless of how sophisticated the modeling technique may be.

Common Normalization Methods and When to Use Them

Normalization is not a single technique but a family of transformations, each designed to address a specific modeling concern. Choosing an appropriate method requires understanding **what problem the transformation is solving** and **which assumptions it implicitly makes**. In this section, we review the most commonly used scaling approaches, discuss their strengths and limitations, and clarify when each method is appropriate.

Z-score Standardization

Z-score standardization rescales a variable so that it has a mean of zero and a standard deviation of one. Each observation x_i is transformed as:

$$z_i = \frac{x_i - \mu}{\sigma},$$

where μ denotes the sample mean and σ the sample standard deviation, both estimated **from the training data only**.

Advantages.
Z-score standardization places variables on a comparable scale while preserving the shape of their original distributions. It is particularly suitable for models that rely on inner products, gradient-based optimization, or regularization (e.g., penalized linear models, SVMs, neural networks).

Limitations.
A widespread misconception is that standardization assumes normally distributed data. This is incorrect. Z-score scaling does **not** require normality; it only uses the first two moments of the distribution. However, it is sensitive to extreme values: large outliers can inflate σ, thereby reducing the relative influence of most observations.

When to use.
A strong default choice when predictors differ substantially in scale and when outliers are either absent or have already been treated.

Min–Max (Range) Scaling

Min–max scaling rescales variables to a fixed interval, most commonly $[0, 1]$. The transformation is:

$$x_i^* = \frac{x_i - \min(x)}{\max(x) - \min(x)}.$$

Advantages.
Intuitive and ensures all transformed values lie within a predefined range. Often used when bounded inputs are desirable (e.g., some neural network settings).

Limitations.
Highly sensitive to extreme values: a single outlier can stretch the range and compress most observations. Also, when applied to test or future data, transformed values may fall outside

$[0,1]$ if they exceed the training-set min/max. This is expected and must be handled in deployment.

When to use.
When input bounds are meaningful and the training data represent the likely range of future observations.

Robust Scaling (Median and IQR)

Robust scaling replaces mean and standard deviation with the median and the interquartile range (IQR). The transformation is:

$$x_i^* = \frac{x_i - \mathrm{median}(x)}{\mathrm{IQR}(x)},$$

where:

$$\mathrm{IQR}(x) = Q_{0.75} - Q_{0.25}.$$

Advantages.
Less affected by extreme values and heavy-tailed distributions; useful when outliers are meaningful rather than errors.

Limitations.
Not universally stable. In highly concentrated variables, $\mathrm{IQR}(x)$ (or related robust measures such as MAD) may be zero or extremely small, making the transformation unstable or undefined. This must be checked explicitly.

When to use.
When outliers are present and structurally inherent, and you want scaling that is less sensitive to extremes.

Power Transformations Combined with Scaling (Box–Cox and Yeo–Johnson)

Power transformations aim to stabilize variance and reduce skewness before scaling.

The **Box–Cox transformation** (for strictly positive data) is:

$$x_i^{(\lambda)} = \begin{cases} \frac{x_i^\lambda - 1}{\lambda}, & \lambda \neq 0, \\ \\ \log(x_i), & \lambda = 0. \end{cases}$$

The **Yeo–Johnson transformation** (allows zero and negative values) is:

$$x_i^{(\lambda)} = \begin{cases} \frac{(x_i+1)^\lambda - 1}{\lambda}, & x_i \geq 0,\ \lambda \neq 0, \\ \\ \log(x_i + 1), & x_i \geq 0,\ \lambda = 0, \\ \\ -\frac{(-x_i+1)^{2-\lambda} - 1}{2-\lambda}, & x_i < 0,\ \lambda \neq 2, \\ \\ -\log(-x_i + 1), & x_i < 0,\ \lambda = 2. \end{cases}$$

Why combine with scaling?
Power transformations modify distributional shape but do not put variables on a common scale. After applying Box–Cox or Yeo–Johnson, variables are typically centered and scaled.

Order matters.
A practical default sequence is: **power transformation → centering → scaling**. Scaling before addressing skewness can weaken the effect of the transformation and complicate interpretation.

When to use.
When strong skewness or heteroscedasticity is present and when model assumptions or optimization benefit from more symmetric distributions.

Choosing a Method: No Single Best Answer

There is no universally optimal normalization method. Each approach reflects a trade-off between robustness, interpretability, and sensitivity to data characteristics. The appropriate choice depends on the model, the data structure, and the modeling objective.

> The relevant question is not *“Which normalization method is best?”*
> but *“Which transformation aligns with my data and my model’s assumptions?”*

Do Different Data Types Require Different Scaling Strategies?

Normalization decisions should never be made independently of data types. Different variable types carry different semantic meanings, and applying the same scaling strategy indiscriminately can lead to misleading representations or unnecessary transformations. A principled preprocessing workflow therefore begins by distinguishing between variable types and understanding how each interacts with scaling.

Continuous Numeric Variables

Continuous numeric variables are the primary candidates for normalization. When such variables are measured on different scales—such as income in thousands and proportions between 0 and 1—scaling is often essential for models that rely on distances, gradients, or regularization. Z-score standardization, min–max scaling, or robust scaling are all reasonable options, depending on the presence of outliers and the modeling objective.

In practice, most normalization methods are designed with continuous variables in mind, and applying them here rarely raises conceptual concerns. The main decision revolves around *which* scaling method is most appropriate, not *whether* scaling should be applied at all.

Count and Ordinal Numeric Variables

Some numeric variables are technically continuous in storage but conceptually represent counts or ordered categories. Examples include the number of visits, rankings, Likert-scale responses, or discrete event counts. Treating such variables as purely continuous can be problematic, especially when their distributions are highly skewed or bounded at zero.

In these cases, applying a logarithmic or power transformation before scaling is often more appropriate than direct normalization. Power transformations can reduce skewness and stabilize variance, after which standardization or robust scaling may be applied. The key point is that

the meaning of the variable matters: a difference of one unit in a count variable does not necessarily carry the same interpretation across its range.

Categorical Variables (Factors or Characters)

Categorical variables should **never** be scaled directly. Their values represent qualitative categories rather than numerical magnitudes, and applying normalization to raw category codes is meaningless.

When categorical variables are included in models that require numeric inputs, they must first be transformed using an encoding scheme such as one-hot (dummy) encoding. After encoding, the question of scaling arises again. In many cases, scaling encoded variables is unnecessary. However, in penalized regression models or distance-based methods, normalization of one-hot encoded variables may be beneficial to ensure that categorical and continuous predictors are treated on comparable scales.

The important distinction is that scaling applies **after encoding**, not before, and only when the model's assumptions justify it.

Binary Variables (0/1 Indicators)

Binary variables occupy a special position. Since they already lie on a fixed and interpretable scale, normalization is usually unnecessary and may even obscure interpretation. For many models, leaving binary indicators unchanged is the most transparent choice.

That said, binary variables often enter preprocessing pipelines automatically when a rule such as "scale all numeric predictors" is applied. In such cases, standardization will transform a 0/1 variable into values centered around zero with unit variance. While this does not usually harm model performance, it changes the interpretation of coefficients and can complicate downstream analysis.

This highlights an important practical lesson: automated preprocessing pipelines should be used with care. Even when a transformation is mathematically valid, it may not be conceptually desirable for all variable types.

Summary: Scaling Depends on Variable Meaning

The decision to normalize should always be guided by the *semantic role* of a variable, not merely by its storage type. Continuous measurements, counts, ordered responses, categorical indicators, and binary flags interact with scaling in fundamentally different ways. Effective preprocessing therefore requires more than applying a generic rule—it requires aligning transformations with the structure and meaning of the data.

Should All Variables Be Scaled?

A common mistake in preprocessing workflows is to treat normalization as a blanket operation applied to every variable in the dataset. In reality, **not all variables should be scaled**, and doing so indiscriminately can reduce interpretability or even introduce unintended distortions. Scaling decisions must therefore be made at the variable level, guided by both statistical and semantic considerations.

The Target Variable (y)

In most predictive modeling tasks, the target variable should **not** be normalized. Scaling the response does not improve model estimation and often complicates interpretation, particularly in regression settings where coefficients and predictions are expected to be expressed in the original units.

There are, however, notable exceptions. In neural network regression or other optimization-heavy models, scaling the target variable can improve numerical stability and convergence behavior. In such cases, predictions must be transformed back to the original scale before evaluation and interpretation. Outside these specific contexts, leaving the target variable unchanged remains the standard and preferred practice.

Predictor Variables

For predictor variables, scaling should be applied selectively rather than universally.

Numeric Predictors Only

Normalization is meaningful only for numeric predictors. Applying scaling to non-numeric variables—either directly or implicitly through arbitrary numeric coding—has no conceptual justification. As discussed earlier, categorical variables must first be encoded, and even then, scaling is optional and model-dependent.

Excluding Non-informative Numeric Variables

Not all numeric variables carry meaningful quantitative information. Identifier variables such as IDs, account numbers, or arbitrary codes may be stored as numeric values but do not represent magnitudes or distances. Scaling such variables is meaningless and potentially harmful, as it introduces artificial structure where none exists. These variables should be excluded from the modeling process altogether, not merely from scaling.

Handling Low-Variance Predictors

Variables with extremely low or zero variance provide little to no information for modeling. Scaling such predictors does not solve the underlying problem; it merely rescales noise. In practice, low-variance and zero-variance predictors should be identified and removed **before** normalization.

Many preprocessing frameworks formalize this step. For example, approaches based on the logic of zero-variance or near-zero-variance filtering (often referred to as `zv` or `nzv` steps) ensure that only informative predictors enter the scaling stage. This not only improves computational efficiency but also reduces the risk of numerical instability in downstream models.

A Practical Rule of Thumb

A disciplined preprocessing workflow follows a clear sequence:

1. Identify and remove non-informative variables (IDs, constants, near-constants).
2. Select numeric predictors that represent meaningful quantities.
3. Apply appropriate scaling only to this subset.
4. Leave the target variable unscaled, unless there is a compelling model-specific reason to do otherwise.

Scaling is most effective when it is **deliberate and selective**, not automatic. Treating normalization as a universal operation may simplify code, but it rarely leads to better models.

Application Plan in R: Data and Modeling Scenario

To demonstrate the practical implications of normalization decisions, we use the **Ames Housing** dataset, a well-known benchmark dataset designed for predictive modeling. The dataset contains **2,930 observations** and a rich set of predictors describing residential properties in Ames, Iowa. These predictors span multiple data types, including continuous numeric variables, discrete counts, ordinal ratings, and categorical features. This diversity makes the dataset particularly suitable for illustrating how scaling interacts with different variable types.

The Ames Housing dataset is distributed within the **modeldata** package in the tidymodels ecosystem. It was explicitly curated for teaching and methodological demonstrations, ensuring a realistic but well-documented structure. The presence of variables measured on vastly different scales—such as living area, lot size, and quality scores—provides a natural setting for exploring the effects of normalization.

Modeling Objective

The primary goal of this application is **not** to optimize predictive performance, but to isolate and examine the impact of different normalization strategies. For this reason, the modeling task is intentionally kept simple. We focus on predicting the **sale price of a house** as a regression problem, using a fixed model specification across all experiments.

The model itself serves merely as a vehicle for comparison. By holding the model constant and varying only the preprocessing strategy, we can attribute differences in performance and behavior directly to scaling decisions rather than to model complexity or tuning choices.

Scope and Focus

Throughout the application section, the emphasis remains firmly on preprocessing:

- the same training–test split is used across all scenarios,
- the same set of predictors is retained,
- the same model structure is applied.

Only the normalization strategy changes. This design allows us to answer a focused question:

> *How much do scaling choices matter when everything else is kept equal?*

By structuring the analysis in this way, the results highlight normalization as an integral component of the modeling pipeline rather than a secondary technical detail.

Transition to Implementation

In the next section, we move from design to execution. We begin by defining a train–test split and establishing a baseline preprocessing workflow. From there, we introduce alternative normalization strategies and compare their effects using consistent evaluation criteria.

Data Access and Availability

The Ames Housing dataset used in this application is available through the **modeldata** package, which is part of the tidymodels ecosystem. No external download is required. Once the package is installed, the dataset can be accessed directly within R.

The dataset is provided for educational and methodological purposes and is accompanied by detailed documentation. For reference, the official description is available at:

https://modeldata.tidymodels.org/reference/ames.html

In the next section, we load the dataset directly from the package and proceed with the train–test split and preprocessing workflow.

Implementation in R: Split, Baseline, and the Cost of Doing It Wrong

In this section, we operationalize the key principle introduced earlier:

> **Split → fit preprocessing on train → apply to train/test**

We use the Ames Housing dataset from the `modeldata` package (no external download required) and compare three pipelines using the **same model**:

1. **Baseline (no scaling)**
2. **Incorrect scaling (data leakage)**: scaling parameters learned from the full dataset
3. **Correct scaling**: scaling parameters learned from the training set only

The goal is not to build the best possible model but to **isolate the effect of scaling decisions**.

Setup and Variable Selection

Before defining any model, we clarify what we are modeling and why these variables are used.

Modeling goal.
We treat `Sale_Price` as the target variable and build a regression model that predicts house sale prices based on a small set of numeric predictors. The purpose is not to maximize predictive accuracy, but to create a controlled environment where the effect of scaling choices is easy to observe.

Why a small subset of predictors?
The Ames dataset contains many variables, including categorical and ordinal predictors. For the normalization demonstrations, we intentionally select a compact set of **numeric** features with clearly different measurement scales. This makes the consequences of scaling (and data leakage) more visible and easier to interpret.

Selected variables (interpretation).

- `Sale_Price`: sale price of the house (response variable).
- `Gr_Liv_Area`: above-ground living area (a size-related continuous measure).
- `Lot_Area`: lot size (typically much larger numeric range than living area).
- `Year_Built`: construction year (a temporal numeric variable).
- `Overall_Cond`: overall condition rating (an ordinal-like numeric score).
- `Latitude`, `Longitude`: geographic coordinates capturing location effects.

Load Data and Create a Working Dataset

```r
library(tidymodels)
library(modeldata)

data(ames, package = "modeldata")

set.seed(2026)

ames_small <- ames %>%
  dplyr::select(
    Sale_Price,
    Gr_Liv_Area,
    Lot_Area,
    Year_Built,
    Overall_Cond,
    Latitude,
    Longitude
  )

# Missing-value check within the selected columns
ames_small %>%
  summarise(across(everything(), ~ sum(is.na(.)))) %>%
  tidyr::pivot_longer(everything(), names_to = "variable", values_to = "n_missing")
```

```
# A tibble: 7 x 2
  variable     n_missing
  <chr>            <int>
1 Sale_Price           0
2 Gr_Liv_Area          0
3 Lot_Area             0
4 Year_Built           0
5 Overall_Cond         0
6 Latitude             0
7 Longitude            0
```

This step constructs a clean working dataset (`ames_small`) and confirms whether missing values exist in the selected columns. For the comparisons in the next sections, it is important that the pipelines differ only by preprocessing choices (e.g., scaling), not by inconsistent handling of missing data.

Train–Test Split and Evaluation Setup

Before discussing scaling, we must establish a clean evaluation setup. The key idea is simple:

> **Split first. Then learn any preprocessing parameters from the training set only.**

Without a proper train–test split, we cannot meaningfully talk about generalization, and any comparison involving normalization risks becoming misleading.

Create a Stratified Train–Test Split

```r
set.seed(2026)

split_obj <- initial_split(ames_small, prop = 0.80, strata = Sale_Price)

train_data <- training(split_obj)
test_data  <- testing(split_obj)

nrow(train_data)
```

```
[1] 2342
```

```r
nrow(test_data)
```

```
[1] 588
```

What this does.

- `prop = 0.80` assigns roughly 80% of the data to training and 20% to testing.
- `strata = Sale_Price` performs a *stratified* split based on the target variable.
  This reduces the risk that the test set ends up with an atypical concentration of very low or very high prices—something that can easily happen with skewed targets like house prices.

How to interpret the output.

- If the full dataset contains 2,930 observations, you should see approximately:

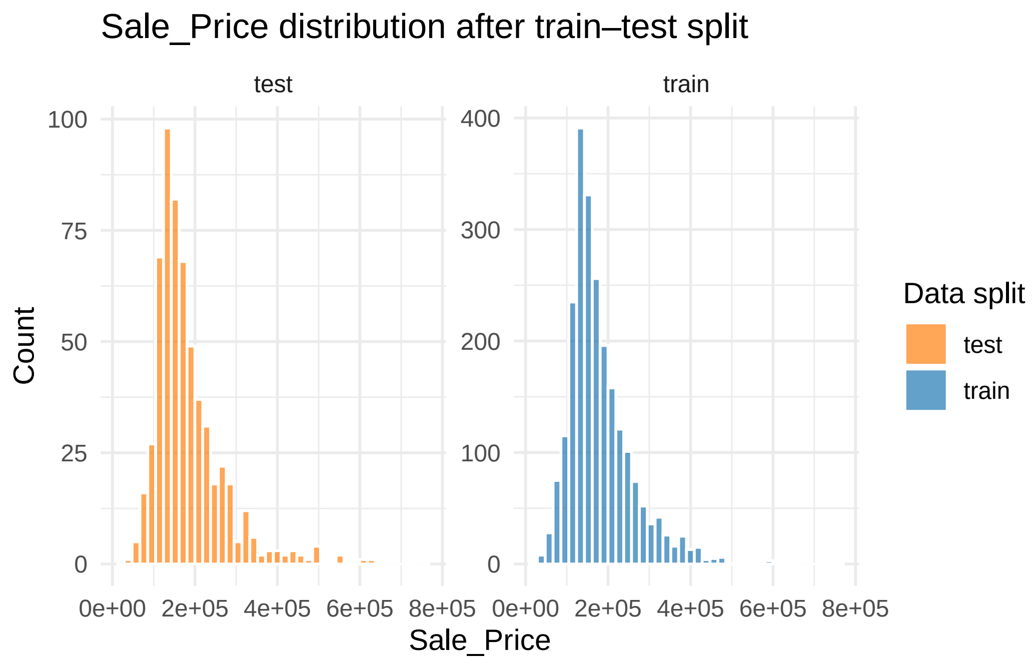
```
-    training: 2,342 rows
```

```
-    test: 588 rows
```

This corresponds closely to the intended 80/20 split and indicates that no unintended row loss occurred during preprocessing.

Sanity Check: Is the Target Distribution Similar Across Splits?

```r
bind_rows(
  train_data %>% mutate(split = "train"),
  test_data  %>% mutate(split = "test")
) %>%
  ggplot(aes(x = Sale_Price, fill = split)) +
  geom_histogram(bins = 40, alpha = 0.7, color = "white") +
  facet_wrap(~ split, scales = "free_y") +
  scale_fill_manual(
    values = c(train = "#1f77b4", test = "#ff7f0e")
  ) +
  labs(
    title = "Sale_Price distribution after train-test split",
    x = "Sale_Price",
    y = "Count",
    fill = "Data split"
  ) +
  theme_minimal()
```



What to look for.

- Both distributions should be right-skewed with a similar central mass.
- There should be no strong imbalance where most expensive (or cheapest) homes appear in only one split.

In the plot, the overall shapes are highly similar and the mid-range is well represented in both sets, indicating that stratification preserved the structure of the target variable across splits.

Optional Check: Quick Summary Statistics

This is a compact numerical confirmation of what the plot shows.

```r
train_summary <- train_data %>%
summarise(
split = "train",
n = n(),
mean = mean(Sale_Price),
median = median(Sale_Price),
sd = sd(Sale_Price),
min = min(Sale_Price),
max = max(Sale_Price)
)
```

```
test_summary <- test_data %>%
summarise(
split = "test",
n = n(),
mean = mean(Sale_Price),
median = median(Sale_Price),
sd = sd(Sale_Price),
min = min(Sale_Price),
max = max(Sale_Price)
)

bind_rows(train_summary, test_summary)
```

```
# A tibble: 2 x 7
  split     n    mean median     sd   min    max
  <chr> <int>   <dbl>  <dbl>  <dbl> <int>  <int>
1 train  2342 180447. 160000 79157. 12789 755000
2 test    588 182185. 160500 82784. 35311 625000
```

How to interpret this.

- Small differences between train and test are expected.
- Large gaps—especially in the median—may indicate an unbalanced split.

Your summaries show nearly identical means and medians (train: 180,447 / 160,000; test: 182,185 / 160,500) and similar standard deviations, supporting the conclusion that the split is well balanced. Differences in the maximum values are expected due to rare high-priced homes and do not indicate a problematic split.

The train–test split is well balanced and suitable for downstream modeling. The test set can be treated as a genuine proxy for unseen data, allowing us to evaluate normalization strategies without confounding effects from an unbalanced split.

Model Specification: A Scale-Sensitive Baseline

Before comparing different normalization strategies, we must fix the modeling component of the pipeline. This ensures that any performance differences observed later can be attributed to preprocessing choices rather than to changes in the model itself.

Why KNN Regression?

We deliberately choose **k-nearest neighbors (KNN) regression** for this demonstration. The reason is methodological, not practical.

KNN is a **distance-based algorithm**: predictions are determined by the distances between observations in the feature space. As a result, KNN is highly sensitive to the scale of the predictors. Variables with larger numeric ranges can dominate distance calculations, even if they are not substantively more important.

This property makes KNN an ideal diagnostic tool for studying the effects of scaling.

Model Specification

We define a single KNN model that will be used in all subsequent scenarios.

```r
knn_spec <- nearest_neighbor(
  neighbors = 15,
  weight_func = "rectangular"
) %>%
  set_engine("kknn") %>%
  set_mode("regression")
```

Commentary.

- The number of neighbors is fixed at 15 to reduce variance while maintaining locality.
- No hyperparameter tuning is performed, as optimization is not the goal here.
- This model specification will remain unchanged across all preprocessing pipelines.

Scenario A — Baseline: No Scaling

We begin with a baseline workflow in which **no scaling is applied**. This provides a reference point against which all normalized pipelines will be compared.

```r
rec_none <- recipe(Sale_Price ~ ., data = train_data)

wf_none <- workflow() %>%
add_recipe(rec_none) %>%
add_model(knn_spec)

fit_none <- fit(wf_none, data = train_data)
```

Note

Note on model engines.
In the tidymodels ecosystem, model specifications are defined independently of the underlying computational engines. Although we specify the KNN model via `nearest_neighbor()`, the actual implementation is provided by the `kknn` package.

If the package is not installed, fitting the model will fail. To proceed, install and load the required engine:

```
install.packages("kknn")
library(kknn)
```

This separation between model specification and engine implementation is intentional and allows tidymodels to remain modular and extensible.

Evaluate on the Test Set

```
pred_none <- predict(fit_none, test_data) %>%
bind_cols(test_data %>% dplyr::select(Sale_Price))

metrics_none <- yardstick::metrics(
pred_none,
truth = Sale_Price,
estimate = .pred
)

metrics_none
```

```
# A tibble: 3 x 3
  .metric .estimator .estimate
  <chr>   <chr>          <dbl>
1 rmse    standard   35643.
2 rsq     standard       0.816
3 mae     standard   23726.
```

Interpretation

These values are not “good” or “bad” in isolation; what matters is that they provide a **stable reference**. At this stage, the model operates on raw predictor scales. For a distance-based method like KNN, this implies:

- Predictors with larger numeric ranges (e.g., `Lot_Area`) can disproportionately influence distance calculations.
- Smaller-range variables (e.g., ordinal-like `Overall_Cond`) may contribute less than intended.
- The model's behavior is therefore partially shaped by measurement units, not only by predictive structure.

This is exactly why KNN is a useful diagnostic tool in a normalization-focused article: if scaling matters, we should see clear changes relative to this baseline once we introduce normalization.

Next, we introduce scaling—but **incorrectly** on purpose. We will apply normalization *before* the train–test split (i.e., using information from the full dataset). This creates **data leakage** and can lead to deceptively improved test performance.

After that, we will implement the correct workflow (fit scaling parameters on the training set only) and compare all scenarios side by side.

Scenario B — Incorrect Normalization (Data Leakage)

In this scenario, we intentionally apply normalization **the wrong way**: we learn scaling parameters from the full dataset (including what will become the test set). This contaminates the evaluation because preprocessing has already "seen" information from the test distribution.

The goal is not to recommend this approach, but to demonstrate how easily leakage can happen—and how it can artificially improve test metrics.

Leakage Pipeline: Normalize Using Full Data

The `step_normalize()` operation applies only to numeric predictors. In our dataset, `Overall_Cond` is stored as a factor (ordinal-like category), so it must not be normalized directly.

```r
rec_leak <- recipe(Sale_Price ~ ., data = ames_small) %>%
  step_normalize(all_numeric_predictors())

# WRONG on purpose: prepping on full data (leakage), but now type-safe
prep_leak <- prep(rec_leak, training = ames_small)

train_leak <- bake(prep_leak, new_data = train_data)
test_leak  <- bake(prep_leak, new_data = test_data)

wf_leak <- workflow() %>%
  add_model(knn_spec) %>%
  add_formula(Sale_Price ~ .)

fit_leak <- fit(wf_leak, data = train_leak)

pred_leak <- predict(fit_leak, test_leak) %>%
  bind_cols(test_leak %>% dplyr::select(Sale_Price))

metrics_leak <- yardstick::metrics(pred_leak, truth = Sale_Price, estimate = .pred)
metrics_leak
```

```
# A tibble: 3 x 3
  .metric .estimator .estimate
  <chr>   <chr>          <dbl>
1 rmse    standard   37036.
2 rsq     standard       0.801
3 mae     standard   24411.
```

Interpretation

The performance obtained under this scenario reflects the consequences of **incorrect normalization with data leakage**.

Compared to the baseline (no scaling), all three metrics deteriorate. This indicates that learning normalization parameters from the full dataset does **not** automatically lead to better predictive performance. In this case, the leakage-induced transformation appears to distort the distance structure in a way that is unfavorable for KNN.

This result is particularly instructive because it challenges a common misconception:
data leakage does not necessarily inflate performance metrics. Its effect depends on the interaction between the preprocessing step, the data distribution, and the model. What leakage *does* guarantee, however, is that the evaluation is no longer valid.

Even if the metrics had improved under this scenario, they could not be trusted as estimates of out-of-sample performance. The test data would no longer represent genuinely unseen observations, since information from their distribution had already been incorporated during preprocessing.

At this point, two important conclusions can be drawn:

1. Scaling decisions materially affect model behavior, especially for distance-based methods.
2. The timing of scaling—*when* parameters are learned—is as critical as *whether* scaling is applied at all.

In the next scenario, we apply normalization correctly by estimating scaling parameters using the training data only and then applying them unchanged to the test set. This will provide the only defensible estimate of generalization performance among the normalization strategies considered.

Scenario C — Correct Normalization (Train-Only Scaling)

In this final preprocessing scenario, normalization parameters are learned **exclusively from the training data** and then applied consistently to both the training and test sets.

This workflow adheres to the core principle of leakage-free modeling.

Correct Pipeline: Normalize Using Training Data Only

```r
rec_ok <- recipe(Sale_Price ~ ., data = train_data) %>%
  step_normalize(all_numeric_predictors())

wf_ok <- workflow() %>%
  add_recipe(rec_ok) %>%
  add_model(knn_spec)

fit_ok <- fit(wf_ok, data = train_data)

pred_ok <- predict(fit_ok, test_data) %>%
  bind_cols(test_data %>% dplyr::select(Sale_Price))

metrics_ok <- yardstick::metrics(pred_ok, truth = Sale_Price, estimate = .pred)

metrics_ok
```

```
# A tibble: 3 x 3
  .metric .estimator .estimate
  <chr>   <chr>          <dbl>
1 rmse    standard   35643.
2 rsq     standard       0.816
3 mae     standard   23726.
```

Interpretation

This scenario represents the **correct normalization workflow**, where scaling parameters are learned exclusively from the training data and then applied unchanged to the test set. The results are **identical to the no-scaling baseline**. This finding is highly informative.

First, it confirms that normalization itself does not automatically improve model performance. When applied correctly, scaling does not inject additional information into the modeling process; it merely changes the representation of the data. If the underlying distance structure relevant for prediction is already dominated by certain predictors, scaling may have little to no effect on performance.

Second, the contrast with the leakage scenario is crucial. In Scenario B, incorrect normalization degraded performance, while in this scenario, correct normalization restores the metrics to their baseline levels. This symmetry reinforces the core message of this article:
the validity of preprocessing matters more than the apparent gains it may produce.

Third, these results highlight an often-overlooked point: the impact of scaling is model- and data-dependent. For this particular subset of predictors and this KNN configuration, normalization neither helps nor harms when applied correctly. In other settings—different feature sets, different distance metrics, or different models—the effect could be substantial.

The key takeaway is therefore not that scaling is unnecessary, but that it must be:

applied deliberately,

restricted to appropriate variables,

and learned at the correct stage of the modeling workflow.

With all three scenarios evaluated, we can now compare them side by side and distill the practical lessons they offer.

Results Comparison

With all three scenarios evaluated, we now compare them side by side. Since the model and data split were held constant, any differences observed here are entirely attributable to preprocessing choices.

Performance Summary

```r
results_tbl <- dplyr::bind_rows(
  metrics_none %>% mutate(scenario = "A - No Scaling"),
  metrics_leak %>% mutate(scenario = "B - Incorrect Scaling (Leakage)"),
  metrics_ok   %>% mutate(scenario = "C - Correct Scaling (Train-Only)")
) %>%
  dplyr::select(scenario, .metric, .estimate) %>%
  tidyr::pivot_wider(
    names_from = .metric,
    values_from = .estimate
  )

results_tbl
```

```
# A tibble: 3 x 4
  scenario                          rmse   rsq    mae
  <chr>                            <dbl> <dbl>  <dbl>
1 A - No Scaling                  35643. 0.816 23726.
2 B - Incorrect Scaling (Leakage)  37036. 0.801 24411.
3 C - Correct Scaling (Train-Only) 35643. 0.816 23726.
```

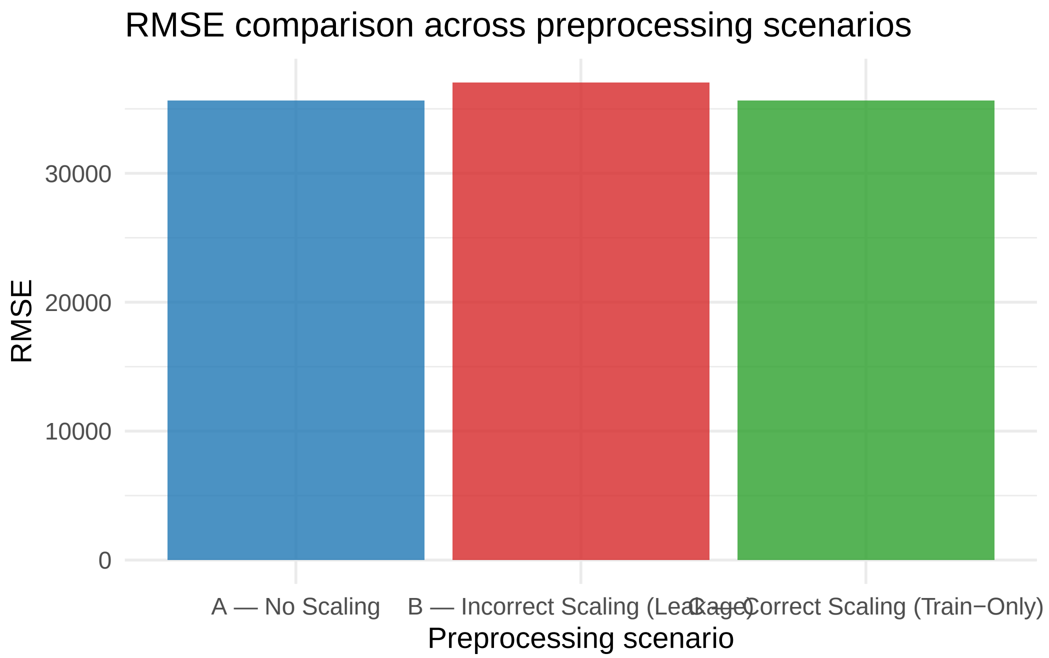
This table summarizes test-set performance across all scenarios.

- **Scenario A (No Scaling)** serves as the baseline.
- **Scenario B (Incorrect Scaling with Leakage)** shows degraded performance.
- **Scenario C (Correct Scaling)** reproduces the baseline results exactly.

Visual Comparison (RMSE)

To make the differences easier to interpret, we visualize RMSE across scenarios.

```r
results_tbl %>%
ggplot(aes(x = scenario, y = rmse, fill = scenario)) +
geom_col(alpha = 0.8) +
scale_fill_manual(
values = c(
"A - No Scaling" = "#1f77b4",
"B - Incorrect Scaling (Leakage)" = "#d62728",
"C - Correct Scaling (Train-Only)" = "#2ca02c"
)
) +
labs(
title = "RMSE comparison across preprocessing scenarios",
x = "Preprocessing scenario",
y = "RMSE"
) +
theme_minimal() +
theme(legend.position = "none")
```



Interpretation

Several important conclusions emerge from this comparison.

First, **normalization does not inherently improve performance**. When applied correctly (Scenario C), scaling neither improves nor degrades performance relative to the no-scaling baseline. This confirms that normalization is a representational transformation, not a source of predictive signal.

Second, **incorrect normalization can be harmful**. Scenario B demonstrates that learning scaling parameters from the full dataset can distort the feature space in ways that negatively affect model behavior. Even more importantly, this scenario yields an invalid evaluation, regardless of whether the metrics appear better or worse.

Third, these results reinforce a central theme of this article:
the correctness of the preprocessing workflow matters more than the choice of preprocessing method itself.

In practice, this means that:

- scaling should be applied only when it aligns with the model's assumptions,
- preprocessing parameters must be learned exclusively from training data,
- and any apparent performance gains should be scrutinized for potential leakage.

Practical Takeaways from the Application

From this controlled experiment, we can distill three practical lessons:

1. **Do not expect normalization to be a silver bullet.** Its impact depends on the model, the data, and the feature set.
2. **Never compromise the train–test boundary.** Leakage can invalidate results even when performance does not improve.
3. **Treat preprocessing as part of the model.** Decisions about scaling are modeling decisions, not technical afterthoughts.

These lessons generalize beyond KNN and apply to any workflow involving scale-sensitive models and data transformations.

Discussion and Conclusion

Normalization is often introduced as a routine preprocessing step, applied almost reflexively before modeling. This article has argued—and demonstrated—that such a view is incomplete. Normalization is not a purely technical adjustment; it is a **modeling decision** whose consequences depend on the interaction between data, model assumptions, and evaluation design.

From a theoretical perspective, scaling matters because many learning algorithms are sensitive to the relative magnitudes of predictors. Distance-based methods, regularized models, kernel methods, and optimization-driven algorithms implicitly encode assumptions about scale. Ignoring these assumptions can distort model behavior, while respecting them can improve stability and interpretability. At the same time, scaling does not create new information. It reshapes how existing information is represented.

The empirical application using the Ames Housing dataset reinforced these points. By holding the model and data split constant and varying only the preprocessing strategy, we isolated the effect of normalization decisions. Three key findings emerged.

First, **normalization does not guarantee performance improvements**. In the correct workflow, scaling reproduced the baseline results exactly. This confirms that normalization should not be expected to "fix" a model by itself. Its role is conditional and context-dependent.

Second, **incorrect normalization compromises validity**. Learning scaling parameters from the full dataset—thereby introducing data leakage—altered model behavior and degraded performance in this example. More importantly, even if the metrics had improved, the evaluation would have been invalid. Leakage undermines the fundamental purpose of a test set: to approximate unseen data.

Third, **the timing of preprocessing is as important as the method chosen**. The difference between valid and invalid evaluation hinged not on whether scaling was applied, but on *when* its parameters were learned. This distinction is often overlooked in practice, yet it is central to trustworthy modeling.

Taken together, these results support a broader principle: preprocessing steps should be treated as integral components of the modeling pipeline, not as detached technical preliminaries. Decisions about normalization should be guided by model assumptions, data characteristics, and evaluation design—not by habit or generic checklists.

In practical terms, this leads to a simple but robust rule:

> **Split the data first. Learn preprocessing parameters from the training set only. Apply the same transformations to all future data.**

Normalization, when used deliberately and correctly, is a powerful tool. When applied mechanically or at the wrong stage, it can mislead. Understanding this distinction is essential for building models that are not only accurate, but also scientifically defensible.

References

- Hastie, T., Tibshirani, R., & Friedman, J. (2009).
  The Elements of Statistical Learning: Data Mining, Inference, and Prediction. Springer.
- Kuhn, M., & Johnson, K. (2013).
  Applied Predictive Modeling. Springer.
- Kuhn, M., & Wickham, H. (2023).
  Tidymodels: A Collection of Packages for Modeling and Machine Learning Using Tidyverse Principles.
  https://www.tidymodels.org/
- Tidymodels Recipes Documentation.
  https://recipes.tidymodels.org/

- Kuhn, M. (Caret package documentation).
  https://topepo.github.io/caret/
- Modeldata package documentation (Ames Housing dataset).
  https://modeldata.tidymodels.org/reference/ames.html