Predicting House Prices with Elastic Net and Gradient Boosting

Daniela Nieto Prada

Introduction

Accurately predicting house prices is a critical task in real estate analytics, as it enables buyers, sellers, and investors to make informed decisions. This project aims to develop predictive models for home prices using two complementary approaches: linear regression with regularization and tree-based ensemble methods. Specifically, the project explores:

- Elastic Net Regression: a linear model that combines both Lasso and Ridge penalties to handle multicollinearity and perform feature selection automatically.
- Gradient Boosting Trees: a powerful ensemble technique that iteratively improves predictive performance by combining multiple weak learners, capturing complex non-linear relationships in the data.

A key aspect of this project is predicting home prices in the logarithmic scale, which stabilizes variance and improves model performance. The workflow includes preprocessing the data, fitting the two models independently, and generating predictions. Model performance is evaluated using the Root Mean Squared Error (RMSE) between the logarithm of actual and predicted prices. The target performance for this project is:

- RMSE less than 0.125 for initial splits of the dataset
- RMSE less than 0.135 for subsequent splits

Model performance is measured using the RMSE on the logarithm of actual versus predicted prices. This dual-model approach not only allows comparison between linear and non-linear modeling techniques but also demonstrates the effectiveness of regularization and ensemble methods in predictive analytics. This repository contains the full implementation in R, including data preprocessing, model training, and prediction generation, providing a reproducible workflow for house price prediction.

Technical Details

Data Preprocessing

The training and test datasets are processed separately to maintain a strict separation between model training and evaluation. The preprocessing workflow begins by transforming the training data using the `transform_data()` function, which simultaneously produces an object called *train_info*. This object stores key preprocessing details required to ensure that identical transformations are consistently applied to the test data.

Once the training data is processed and *train_info* is created, the script loads the test dataset and applies the same transformations using the stored preprocessing information.

The `transform_data()` function takes three inputs:

1. The dataset (training or test data)
2. A binary indicator (`1` for training data, `0` for test data)
3. *train_info*, a storage container for critical transformation parameters obtained from the training data preprocessing step

The preprocessing steps performed by `transform_data()` are as follows:

1. Variable Removal

- Categorical variables where the dominant category exceeds 95% were removed: *Street*, *Utilities*, *Condition_2*, *Roof_Matl*, *Heating*, *Pool_QC*, and *Misc_Feature*.
- An exception was made for *Land_Slope*, which has categories "Gtl", "Mod", and "Sev". "Gtl" comprises 95.01% of observations; the remaining categories were combined into "Other" to retain predictive information while reducing sparsity.
- Numerical variables with minimal variance or non-informative patterns were removed. *Longitude* and *Latitude* have very small standard deviations, covering a limited geographic area. *PID* is a unique property identifier and not predictive. *Low_Qual_Fin_SF*, *Pool_Area*, *Three_season_porch*, and *Misc_Val* are mostly zeros and remained non-informative even after transformations.

2. Categorical Variables and One-Hot Encoding

- Categorical variables are identified by checking for variables of type *factor* or *character* and stored in the subset *x_cat*.
- One-hot encoding is applied to all categorical variables using the `dummy_cols()` function from the **fastDummies** library.
- This function generates binary columns for each level of every categorical variable, removing the original categorical columns afterward.
- For the training data, encoded variable names are saved in *train_info$encoding_vars* for use during test preprocessing.
- For the test data, after one-hot encoding, the script checks for any missing encoding columns compared to *train_info$encoding_vars*. Missing columns are added with zeros, and only the encoding columns present in the training data are retained. This approach ensures consistency and prevents issues with unseen factor levels in the test data.

3. Numerical Variables and Missing Values

- Numerical variables are identified by checking for variables of type *numeric* and stored in the subset *x_num*.
- Missing values in *Garage_Yr_Blt* are replaced with zeros.

4. Skewness Correction

- For the training data, the skewness of each numerical variable was calculated using the `Skew()` function from the **DescTools** package.

- Variables with absolute skewness greater than 0.5 were transformed using the square root.
- The transformed variables are: *Bsmt_Unf_SF*, *Second_Flr_SF*, *Bsmt_Full_Bath*, *TotRms_AbvGrd*, *Fireplaces*, *Wood_Deck_SF*.
- For the test data, the same square root transformation was applied to these variables to ensure consistency.

5. Outlier Treatment (Winsorization)

- Several numerical variables are winsorized to reduce the influence of extreme outliers: *Lot_Frontage*, *Lot_Area*, *Mas_Vnr_Area*, *BsmtFin_SF_2*, *Bsmt_Unf_SF*, *Total_Bsmt_SF*, *Second_Flr_SF*, *First_Flr_SF*, *Gr_Liv_Area*, *Garage_Area*, *Wood_Deck_SF*, *Open_Porch_SF*, *Enclosed_Porch*, and *Screen_Porch*.
- For the training data, the 95th percentile (upper quantile) is calculated for each variable, and values exceeding this threshold are capped.
- These quantile values are stored in *train_info$quantile*.
- For the test data, each variable is capped using the reference quantiles stored in *train_info$quantile*.

6. Standardization

- Numerical variables are standardized by centering them around their mean.
- For the training data, means are subtracted and stored in *train_info$mean*.
- For the test data, centering is applied using the same means from the training data to maintain consistency.

7. Final Combination

The transformed *x_num* and *x_cat* subsets are combined into a single feature matrix using `cbind()`. This final preprocessed dataset is then passed on for model fitting and prediction.

Model Implementation

This project implements two distinct predictive models, a **regularized linear regression model (Elastic Net)** and a **tree-based ensemble model (Gradient Boosting)**, to predict house prices in logarithmic scale. Each model captures different patterns in the data, providing both interpretability and flexibility in the modeling approach.

1. Elastic Net Regression

The Elastic Net regression model is implemented using the `glmnet` package, which combines both *Lasso* (L1) and *Ridge* (L2) regularization.
An *alpha* value of 0.5 is used, providing an equal balance between the two penalties. This approach helps manage multicollinearity while performing automatic feature selection. To identify the optimal level of regularization, cross-validation is performed using a sequence of *lambda* values ranging from 10^{-10} to 10^{10} (on the log scale).
The value of *lambda* that minimizes the Mean Squared Error (MSE) across the folds is selected and stored as *best_lam*. The model is then refitted using *best_lam*, and predictions are generated on the processed test dataset.

2. Gradient Boosting Model

The second model is a Gradient Boosting Machine (GBM), implemented using the `gbm` package.
This approach builds an ensemble of decision trees sequentially, where each new tree aims to correct the residuals of the previous ones. Gradient boosting is particularly effective for capturing non-linear relationships and complex feature interactions.

The GBM model is trained using the following hyperparameters:

- Number of trees: 1000
- Learning rate: 0.05
- Maximum tree depth: 6

Once trained, the GBM model produces predictions for the test dataset, which are then used to compute the final performance metrics.

These two models, the Elastic Net Regression and GBM, together provide a balanced comparison between a regularized linear approach and a non-linear ensemble method, illustrating how different algorithmic strategies can capture distinct aspects of housing market dynamics.

Model Training

Load Libraries and Data.

```r
#install.packages("DescTools")
#install.packages("fastDummies")
#install.packages("glmnet")
#install.packages("gbm")
#install.packages("ggplot2")
library(DescTools)
library(fastDummies)
library(glmnet)
library(gbm)
library(ggplot2)
```

Create Functions to Preprocess the Data

```r
transform_data = function(data, is_train = 1, train_info) {
  # List of variables to remove
  vars_to_remove = c("Street", "Utilities", "Condition_2", "Roof_Matl",
                     "Heating", "Pool_QC", "Misc_Feature", "Low_Qual_Fin_SF",
                     "Pool_Area", "Three_season_porch", "Misc_Val",
                     "Longitude", "Latitude", "PID")

  # Remove specified variables from data
  data = data[, !names(data) %in% vars_to_remove]
```

```r
# Select categorical variables
categorical_variables = names(data)[sapply(data, function(data)
  is.factor(data) || is.character(data))]
x_cat = subset(data, select = categorical_variables)
# Combine rare levels in Land_Slope into "Other"
x_cat$Land_Slope = ifelse(x_cat$Land_Slope == "Gtl", "Gtl", "Other")

# Select numerical variables
numerical_variables = names(data)[sapply(data, function(data)
  is.numeric(data))]
x_num = subset(data, select = numerical_variables)

# The only missing values are in Garage_Yr_Blt for houses with no garage,
# so replace missing values with 0
x_num$Garage_Yr_Blt = ifelse(is.na(x_num$Garage_Yr_Blt), 0, x_num$Garage_Yr_Blt)

# List of variables to apply square root transformation
vars_to_sqrt = c("Bsmt_Unf_SF", "Second_Flr_SF", "Bsmt_Full_Bath",
                      "TotRms_AbvGrd", "Fireplaces", "Wood_Deck_SF")
for (col_name in vars_to_sqrt) {
  x_num[, col_name] = sqrt(x_num[, col_name])
}

# List of variables to winsorize
vars_to_winsorize = c("Lot_Frontage", "Lot_Area", "Mas_Vnr_Area",
                      "BsmtFin_SF_2", "Bsmt_Unf_SF", "Total_Bsmt_SF",
                      "Second_Flr_SF", 'First_Flr_SF', "Gr_Liv_Area",
                      "Garage_Area", "Wood_Deck_SF", "Open_Porch_SF",
                      "Enclosed_Porch", "Screen_Porch")
if (is_train == 1) {
  # Create an empty data frame to store quantiles
  quantile = data.frame(matrix(NA, nrow = 1, ncol = length(vars_to_winsorize)))
  colnames(quantile) = vars_to_winsorize
  # Calculate and apply winsorization to each variable in vars_to_winsorize
  for (col_name in vars_to_winsorize) {
    # Calculate the 95% upper quantile based on the training data for the
    # current variable
    quantile[, col_name] = quantile(x_num[, col_name], probs = 0.95,
                                    na.rm = TRUE)

    # Apply winsorization to data for the current variable
    x_num[, col_name] = pmin(x_num[, col_name], quantile[, col_name])
  }
  train_info$quantile = quantile
}
else {
  # Apply winsorization using the 95% upper quantile based on the training data
  for (col_name in vars_to_winsorize) {
    x_num[, col_name] = pmin(x_num[, col_name], train_info$quantile[, col_name])
  }
}

# Standardize data
```

```r
  if (is_train == 1) {
    # Substract the mean from the data
    x_num = scale(x_num, center = TRUE, scale = FALSE)
    train_info$mean = attr(x_num, "scaled:center")
  }
  else {
    # Use the mean from the training data to standardize the test data
    x_num = scale(x_num, train_info$mean, scale = FALSE)
  }

  # Perform one-hot encoding for categorical variables
  x_cat = dummy_cols(x_cat, select_columns = names(x_cat),
                     remove_selected_columns = TRUE)

  if (is_train == 1) {
    # Identify the encoding columns from the training data
    train_info$encoding_vars = names(x_cat)
  }
  else {
    # Create a data frame of zeros for the missing columns in x_cat
    missing_vars = setdiff(train_info$encoding_vars, names(x_cat))
    missing_data = data.frame(matrix(0, nrow = nrow(x_cat),
                                     ncol = length(missing_vars)))
    colnames(missing_data) = missing_vars
    # Add the missing columns to x_cat
    x_cat = cbind(x_cat, missing_data)
    # Ignore unseen levels in x_cat
    x_cat = x_cat[, train_info$encoding_vars]
  }

  # Combine categorical and numeric data frames using cbind
  x = as.matrix(cbind(x_cat, x_num))

  return(list(x = x, train_info = train_info))
}
```

Create Functions to Fit the Two Models

```r
fit_regression = function(x, y, alpha) {
  # Create a sequence of lambda values
  lambda_seq = exp(seq(-10, 10, length.out = 100))

  # Perform cross-validation for regression using glmnet
  cv_out = cv.glmnet(x, y, alpha = alpha,
                     lambda = lambda_seq)
  # Find the lambda value that minimizes the mean squared error
  best_lam = cv_out$lambda.min

  return(list(cv_out = cv_out, best_lam = best_lam))
}
```

```r
fit_boosting_tree = function(x, y, num_trees = 300, learning_rate = 0.05,
                             max_depth = 6) {
  # Fit a gradient boosting model using gbm
  model = gbm(
    formula = y ~ .,
    data = as.data.frame(cbind(x, y)),
    distribution = "gaussian", # For regression
    n.trees = num_trees,
    shrinkage = learning_rate,
    interaction.depth = max_depth,
  )
  return(model)
}
```

Load and Transform the Data, Train the Models, Make Predictions

```r
set.seed(1047)

# Create an empty list to store the errors
rmse_reg = rep(0,10)
rmse_tree = rep(0, 10)

# Loop through the folder names and read the corresponding train.csv files
for (idx in 1:10) {
  # Load training data
  file_path = file.path(paste0("Data/fold", idx ) , "train.csv")
  train_data = read.csv(file_path)
  # Store Sale_Price as y_train and remove it from x_train
  y_train = subset(train_data, select = Sale_Price)
  y_train = log(unname(unlist(y_train)))
  x_train = subset(train_data, select = -Sale_Price)
  # Transform training data
  train_trans = transform_data(x_train, is_train = 1, list())

  # Train first model
  first_model = fit_regression(train_trans$x, y_train, 0.5)
  # Train second model
  n_trees = 1000
  second_model = fit_boosting_tree(train_trans$x, y_train, n_trees)

  # Load test data
  file_path = file.path(paste0("Data/fold", idx ) , "test.csv")
  x_test = read.csv(file_path)
  test_trans =  transform_data(x_test, is_train = 0, train_trans$train_info)
  # Load y test
  file_path = file.path(paste0("Data/fold", idx ) , "test_y.csv")
  y_test = read.csv(file_path)
  y_test = subset(y_test, select = -PID)
  y_test = log(unname(unlist(y_test)))

  # Make predictions
  # Use the selected lambda to make predictions on the test data
  y_hat_reg = as.numeric(predict(first_model$cv_out, s = first_model$best_lam,
                              newx = test_trans$x))
  y_hat_tree = predict(second_model, newdata = as.data.frame(test_trans$x),
                    n.trees = n_trees)

  # Calculate the RMSE between predicted and actual values
  rmse_reg[idx] = round(sqrt(mean((y_test - y_hat_reg)^2)),3)
  rmse_tree[idx] = round(sqrt(mean((y_test - y_hat_tree)^2)),3)
}
```

Analysis and Model Evaluation

This section evaluates the performance of the two models using cross-validation and test set predictions. Model performance is measured by the RMSE between the logarithm of predicted and actual sale prices. For each model, RMSE values were computed across all 10 data folds. The results summarized in the table below show that both models meet the performance targets.

```r
fold_performance = as.table(rbind(rmse_reg, rmse_tree))
dimnames(fold_performance) = list(Model = c("Elastic Net Regression",
                                            "Gradient Boosting"),
                      Fold  = 1:10)
knitr::kable(
  fold_performance,
  caption = "Cross-Validation Performance for Each Fold
  (RMSE on Log-Transformed Prices)",
  digits = 3)
```

Table 1: Cross-Validation Performance for Each Fold (RMSE on Log-Transformed Prices)

	1	2	3	4	5	6	7	8	9	10
Elastic Net Regression	0.122	0.118	0.121	0.120	0.112	0.132	0.126	0.121	0.13	0.124
Gradient Boosting	0.122	0.123	0.116	0.119	0.114	0.129	0.132	0.128	0.13	0.124

A second table reports the mean RMSE and standard deviation across folds, providing a concise view of overall model accuracy and stability.

```r
performance_summary = data.frame(
  Model = c("Elastic Net Regression", "Gradient Boosting"),
  RMSE = c(mean(rmse_reg),mean(rmse_tree)),
  SD = c(sd(rmse_reg),sd(rmse_tree)))
knitr::kable(
  performance_summary,
  caption = "Cross-Validation Performance Summary
  (RMSE and SD on Log-Transformed Prices)",
  digits = 4)
```

Table 2: Cross-Validation Performance Summary (RMSE and SD on Log-Transformed Prices)

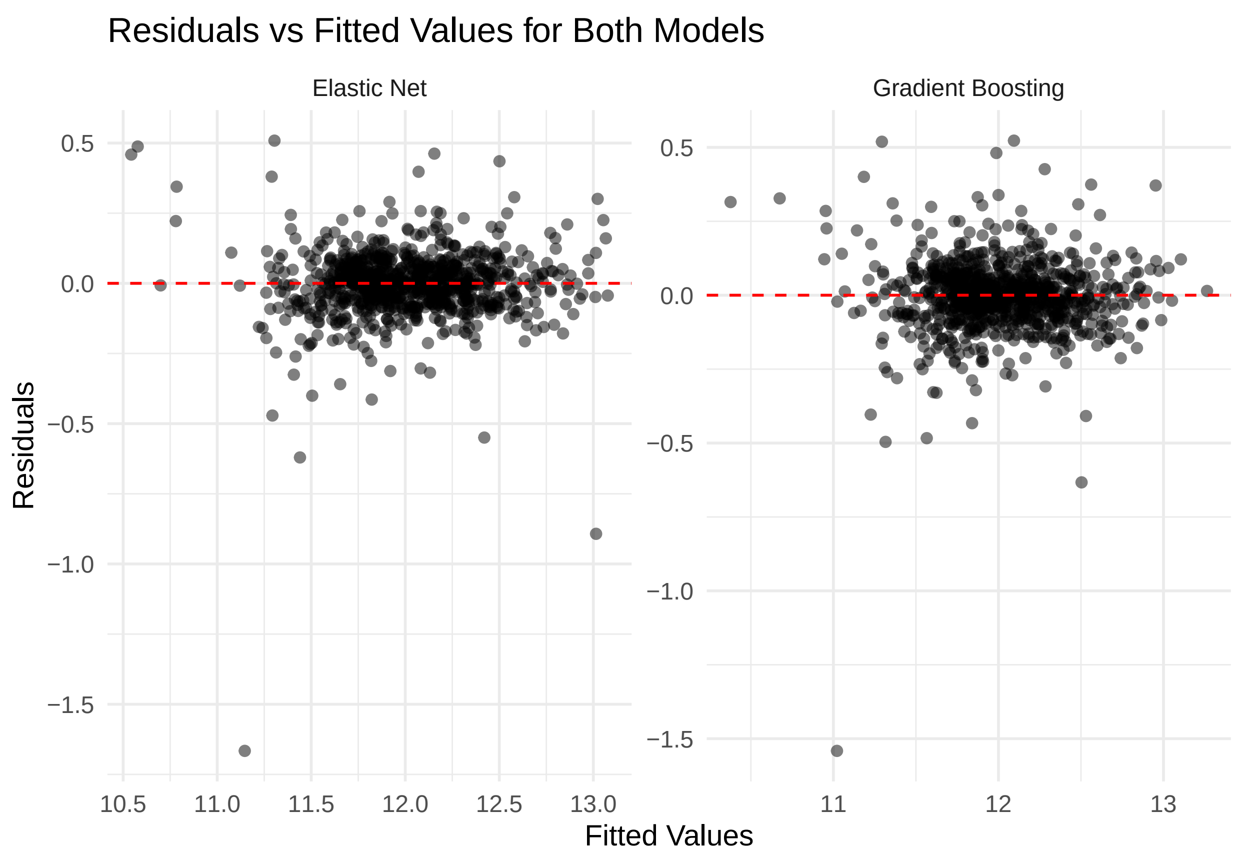
Model	RMSE	SD
Elastic Net Regression	0.1226	0.0058
Gradient Boosting	0.1237	0.0061

Both models performed well, achieving RMSE values near 0.123, which is below the target threshold. Several insights emerge from the comparison:

1. Elastic Net Regression Elastic Net achieved the lowest average RMSE, with very small variation across folds. This consistency suggests that the linear relationships in the data—paired with regularization generalize well. The combined L1/L2 penalty likely helped stabilize coefficients while performing implicit feature selection.
2. Gradient Boosting Model Gradient Boosting produced an RMSE only slightly higher than Elastic Net, with similarly low variance across folds. Although tree-based models typically capture nonlinear interactions, its performance being nearly identical suggests that nonlinearities in the dataset are limited or already handled effectively by preprocessing.
3. Model Comparison The small difference in performance confirms that both models learned the underlying structure of the housing prices effectively. Elastic Net's slight edge indicates that, after transformations and standardization, the feature space was especially compatible with a penalized linear model. Gradient Boosting's comparable performance reinforces that the engineered features were sufficient to capture most of the predictive signal.

Residual plots can provide insights into potential bias or heteroscedasticity.The side-by-side residual-vs-fitted plots below show that both models produce residuals centered around zero with similar spread. There are no strong patterns or funnel shapes, suggesting that variance is reasonably stable across fitted values.

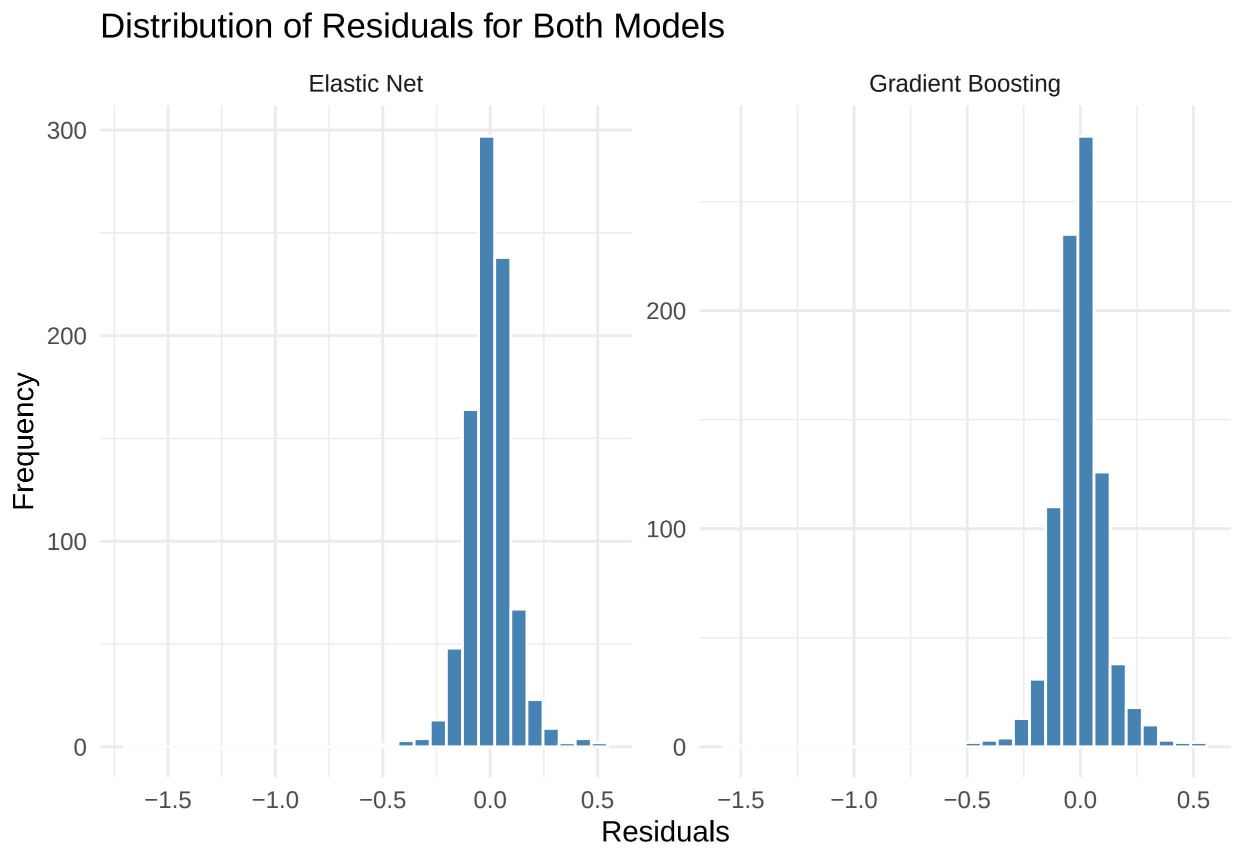
```r
# Residuals for each model
res_reg  = y_test - y_hat_reg
res_tree  = y_test - y_hat_tree
# Combined data frame for faceting
df_res = data.frame(
  Fitted = c(y_hat_reg, y_hat_tree),
  Residuals = c(res_reg, res_tree),
  Model = factor(rep(c("Elastic Net", "Gradient Boosting"),
                     each = length(y_hat_reg)))
)
ggplot(df_res, aes(x = Fitted, y = Residuals)) +
  geom_point(alpha = 0.5) +
  geom_hline(yintercept = 0, color = "red", linetype = "dashed") +
  facet_wrap(~ Model, scales = "free") +
  labs(
    title = "Residuals vs Fitted Values for Both Models",
    x = "Fitted Values",
    y = "Residuals"
  ) +
  theme_minimal()
```



Residual distributions for both models are nearly identical. They are approximately symmetric with a single outlier with a high-magnitude residual in both histograms, but the rest of the distribution is smooth and well-behaved. Together, these diagnostics indicate that the preprocessing steps successfully stabilized variance and reduced skewness, leaving only one observation that falls meaningfully outside the general trend.

```r
# Combined data frame
df_hist = data.frame(
  Residuals = c(res_reg, res_tree),
  Model = factor(rep(c("Elastic Net", "Gradient Boosting"),
                     each = length(res_reg)))
)

ggplot(df_hist, aes(x = Residuals)) +
  geom_histogram(bins = 30, fill = "steelblue", color = "white") +
  facet_wrap(~ Model, scales = "free") +
  labs(
    title = "Distribution of Residuals for Both Models",
    x = "Residuals",
    y = "Frequency"
  ) +
  theme_minimal()
```



Conclusions

This project developed and evaluated two predictive models for house price estimation. Through systematic data preprocessing, transformation, and model tuning, both approaches achieved strong predictive performance under the log-scale RMSE metric.

Both models performed strongly and met the performance thresholds, but Elastic Net Regression emerged as the best-performing model overall, offering the lowest RMSE and most stable results. The Gradient Boosting model remained a valuable complementary method, demonstrating competitive accuracy and offering robustness to nonlinear relationships. These results highlight that thoughtful preprocessing, including skewness correction, winsorizing outliers, careful feature removal, and standardized encoding, was essential to achieving reliable performance across both linear and tree-based models.

However, the dataset used in this analysis covers a very small geographic region, resulting in minimal variation in location-based predictors such as latitude and longitude. Because of this limited spatial diversity, the models were unable to leverage the type of rich location-driven patterns that typically play a major role in real estate pricing. In a broader and more varied dataset, one with neighborhoods, districts, and meaningful geographic gradients, location would likely emerge as one of the strongest predictors.

Exploring a larger, more diverse dataset would therefore be a valuable next step. In such a setting, tree-based models like Gradient Boosting could potentially outperform linear approaches by capturing complex interactions between location, property characteristics, and neighborhood-level context. This would provide a more realistic and informative assessment of model performance in real-world housing markets.