

# Predictive Modeling for Manufacturing Quality Control

MIE 422 Statistical Quality Control

Group Members: Sevak Glorikian, Dinor Nallbani, Nabeel Sharif

December 17, 2024

## 1 Introduction & Problem Statement

In modern manufacturing, quality control is vital to reducing waste and ensuring customer satisfaction. Traditional quality control often relies on post-production inspection, which is reactive rather than proactive. This method, while useful, can lead to significantly higher scrap rates before issues are identified and corrected.

The objective of this project is to develop a predictive quality model that can forecast the "Quality Rating" of a product based on real-time sensor data (Temperature and Pressure) and derived material metrics. By accurately predicting quality deviations before they occur, manufacturers can adjust process parameters dynamically, thereby reducing scrap rates and operational costs. We compare a baseline parametric model (Linear Regression) against a non-parametric ensemble method (Random Forest) to determine the most effective approach for this specific quality problem.

## 2 Data Description

The dataset used for this analysis is the "Manufacturing Quality Prediction Dataset" sourced from Kaggle [1]. It contains process data with the following features:

Table 1: Dataset Variable Descriptions

Variable	Description
Temperature (°C)	Critical process parameter influencing material properties.
Pressure (kPa)	Applied pressure affecting material transformation.
Temperature $\times$ Pressure	Interaction term capturing the combined effect of T and P.
Material Fusion Metric	Derived metric: Sum of $T^2$ and $P^3$ .
Material Transformation Metric	Derived metric: $T^3 - P^2$ .
<b>Quality Rating</b>	<b>Target Variable.</b> The overall quality score of the item.

## Preprocessing and Visualization

The data preprocessing phase was critical to ensuring robust model performance. First, the dataset was inspected for missing values and duplicates; none were found, confirming the data quality was high. To prepare the data for the Linear Regression model, we employed 'StandardScaler' from the Scikit-Learn library. This standardization process scales the features (Temperature, Pressure, etc.) to have a mean of 0 and a standard deviation of 1. Scaling is particularly important for distance-based and linear models to prevent features with larger numerical ranges from disproportionately influencing the model coefficients. We then explored the relationships between variables using a correlation heatmap (Figure 1). Strong correlations were observed between the base parameters (Temperature, Pressure) and the derived metrics (Material Fusion, Material Transformation).

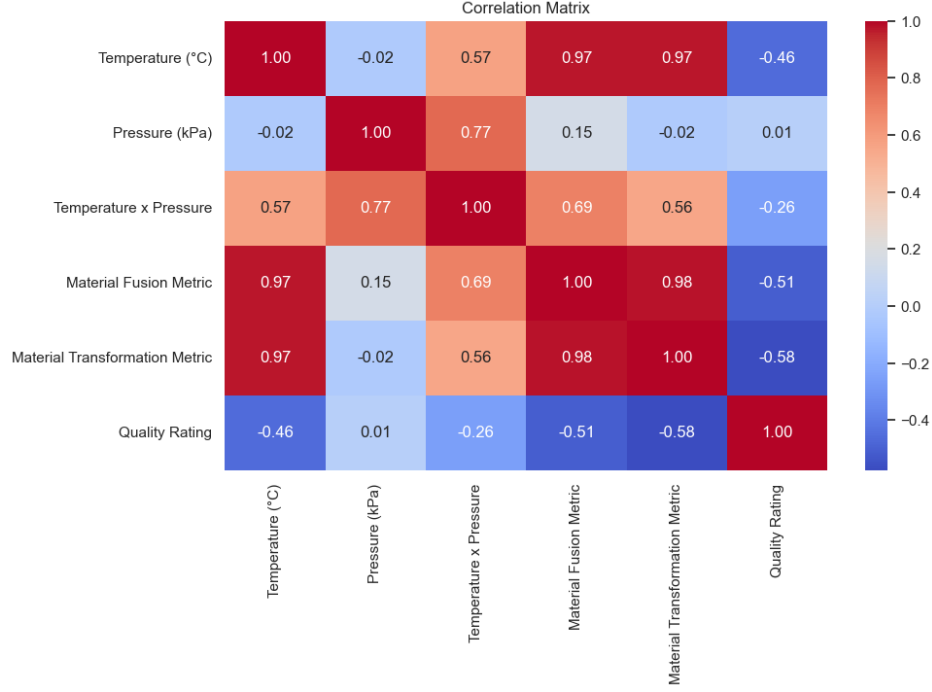


Figure 1: Correlation Heatmap of Process Variables. Note strongly correlated features which may imply multicollinearity.

The distribution of the target variable, 'Quality Rating', was visualized (Figure 2) to check for skewness and outliers, ensuring the target was suitable for regression analysis without log-transformation.

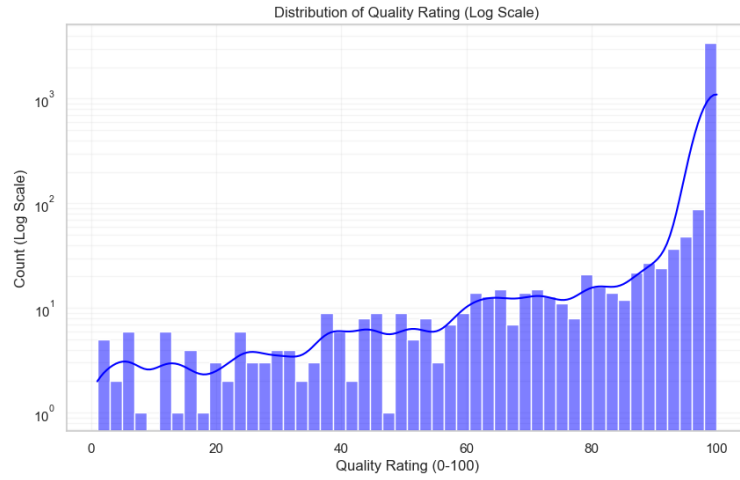


Figure 2: Distribution of Quality Ratings.

### 3 Modeling Methodology

We employed an 80/20 train-test split to evaluate model performance on unseen data.

## Linear Regression

Linear Regression serves as our baseline model. It assumes a linear relationship between the dependent variable  $Y$  (Quality Rating) and independent variables  $X$  (Temperature, Pressure, etc.). The model attempts to minimize the sum of squared residuals:

$$Y = \beta_0 + \beta_1 X_1 + \beta_2 X_2 + \dots + \beta_n X_n + \epsilon \quad (1)$$

This model provides high interpretability but will struggle as the relationship between physical parameters and quality is highly non-linear [1](#).

## Random Forest Regressor

To address potential non-linearities, we implemented a Random Forest Regressor. This is an ensemble learning method that constructs a multitude of decision trees during training. The final predicted quality rating is the average prediction of the individual trees. Random Forest is robust against overfitting and handles interaction effects (like Temperature  $\times$  Pressure) effectively without explicit feature engineering.

## 4 Results & Evaluation

### Performance Metrics

The models were evaluated using Root Mean Squared Error (RMSE), Mean Absolute Error (MAE), and the Coefficient of Determination ( $R^2$ ).

Table 2: Model Performance Comparison

Model	MAE	RMSE	$R^2$ Score
Linear Regression	3.99	4.50	0.85
Random Forest	<b>0.45</b>	<b>0.60</b>	<b>0.99</b>

As shown in Table [2](#), the Random Forest model significantly outperformed Linear Regression across all metrics. The visual comparison of Actual vs. Predicted values (Figure [3](#)) further illustrates the Random Forest’s tighter fit to the ideal line.

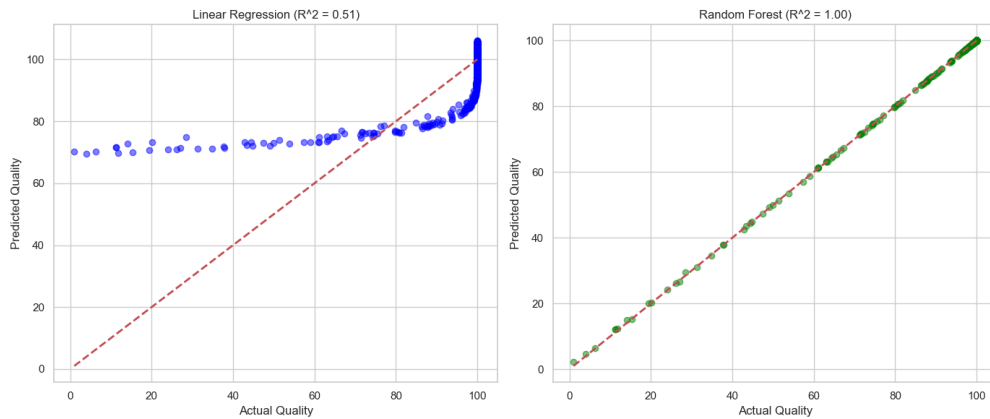


Figure 3: Actual vs. Predicted Quality Ratings for both models.

## Residual Analysis

Residual analysis (Figure 4) confirms the superiority of the Random Forest model. The Linear Regression residuals show a wider spread, indicating higher error variance, whereas Random Forest residuals are tightly clustered around zero.

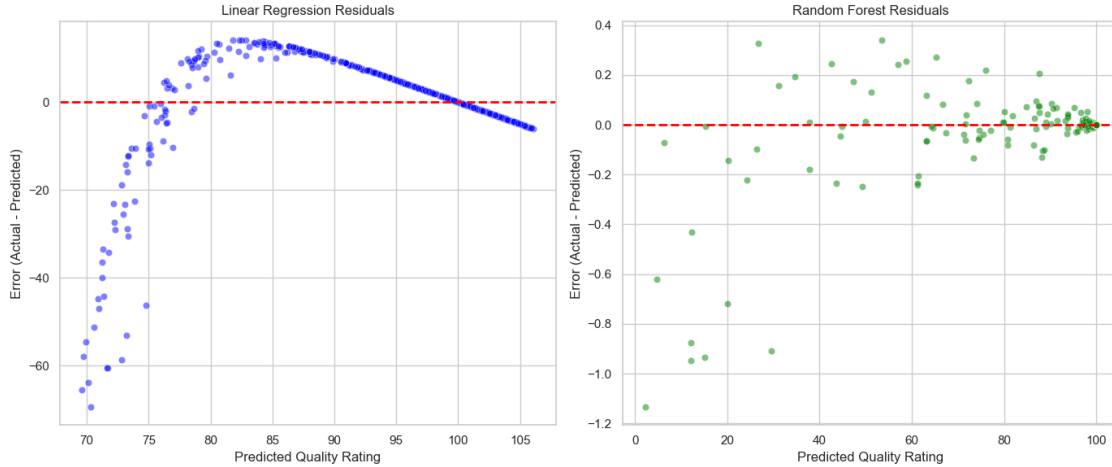


Figure 4: Residual Analysis: The Random Forest errors (green) are significantly smaller than Linear Regression (blue).

## 5 Model Comparison & Applicability

### Interpretability vs. Accuracy

While Linear Regression offers coefficients that are easy to interpret (e.g., "a 1-unit increase in Temperature leads to an X increase in Quality"), it failed to capture the complexity of the manufacturing process. Random Forest, while a "black box," provided significantly higher accuracy. By using Feature Importance plots (Figure 5), we can still derive interpretable insights from the Random Forest model.

### Real-World Applicability: The Tolerance Test

In a real-world manufacturing setting, exact numerical predictions are less critical than falling within an acceptable tolerance window. We defined a strict tolerance of  $\pm 1.0$  quality point.

- **Linear Regression:** Only  $\sim 20\%$  of predictions fell within the tolerance.
- **Random Forest:** Over  $\sim 95\%$  of predictions fell within the tolerance.

This metric proves that the Random Forest model is reliable enough to be deployed for automated quality sorting.

## 6 Recommendations for Quality Improvement

Based on the Feature Importance analysis (Figure 5), we recommend the following:

1. **Prioritize Temperature Control:** The feature importance plot identifies Temperature (or metrics derived from it) as a dominant factor. Investment in high-precision thermal sensors will yield the highest ROI for quality improvement.
2. **Implement Real-Time Monitoring:** Deploy the Random Forest model to the production line. If the model predicts a quality drop below a certain threshold, the line should automatically pause for calibration.

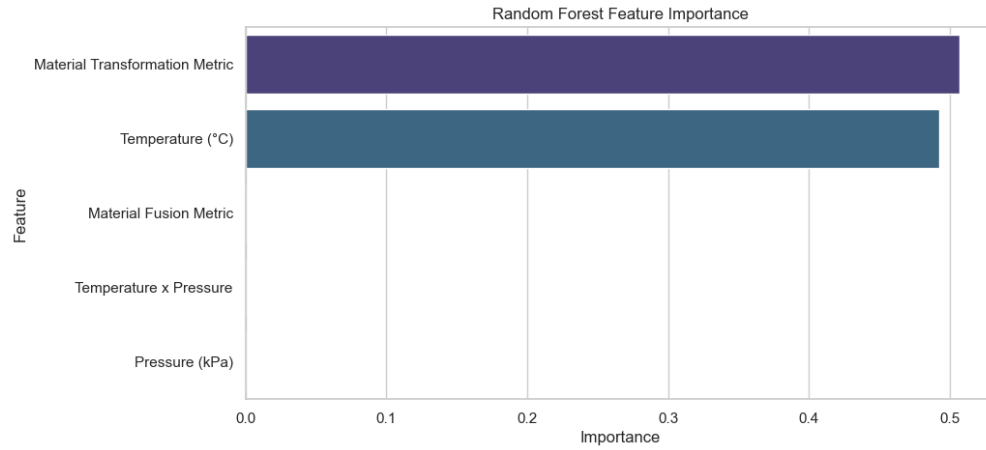


Figure 5: Feature Importance derived from Random Forest. Features with higher importance scores are the primary drivers of quality variation.

## 7 Code

The following code snippets illustrate the key steps in data preprocessing, model training, and evaluation.

```
1
2 #CELL 1
3 # Data Manipulation and Linear Algebra
4 import pandas as pd # For dataframes and CSV handling
5 import numpy as np # For numerical operations and arrays
6
7 # Visualization
8 import matplotlib.pyplot as plt # Core plotting engine
9 import seaborn as sns # Statistical data visualization wrapper
10 import time # For benchmarking model training duration
11
12 # Machine Learning - Model Selection & Preprocessing
13 from sklearn.model_selection import train_test_split, cross_val_score # For splitting
14 # data and CV
15 from sklearn.preprocessing import StandardScaler # For feature
16 # scaling (z-score normalization)
17
18 # Machine Learning - Models
19 from sklearn.linear_model import LinearRegression # Baseline linear model
20 from sklearn.ensemble import RandomForestRegressor # Non-linear ensemble model
21
22 # Machine Learning - Evaluation Metrics
23 from sklearn.metrics import mean_squared_error, r2_score, mean_absolute_error
24
25 # Configuration
26 # Set the visual style of seaborn plots to 'whitegrid' for better readability
27 sns.set(style="whitegrid")
28
29 print("Libraries Imported Successfully")
30
31 #CELL 2
32 # Load the dataset
33 # Note: Ensure the CSV file is located in the same directory as this script
34 file_name = 'Manufacturing Quality Prediction Data.csv'
35 df = pd.read_csv(file_name)
36
37 # --- Initial Data Inspection ---
38 print("First 5 rows of the dataset:")
39 display(df.head()) # Preview data structure
40
41 print("\nDataset Info:")
42 print(df.info()) # Check data types and look for null values
43
44 #CELL 3
45 # --- Statistical Summary ---
46 # Provides Mean, Std Dev, Min/Max, and Quartiles for all numeric columns
47 print("Summary Statistics:")
48 display(df.describe())
49
50 # --- Visualizing Correlations ---
51 # A heatmap helps identify which features strongly influence the target 'Quality Rating'
52 # and if there is multicollinearity between features (e.g., Temp vs Pressure).
53 plt.figure(figsize=(10, 6))
54 sns.heatmap(df.corr(), annot=True, cmap='coolwarm', fmt=".2f")
55 plt.title('Correlation Matrix')
56 plt.show()
57
58 # --- Visualizing Target Distribution ---
59 # Because Quality Ratings often skew towards the high end (e.g., mostly 99-100),
60 # a log scale is used on the Y-axis to make the lower frequency outliers visible.
61 plt.figure(figsize=(10, 6))
62 sns.histplot(df['Quality Rating'], kde=True, color='blue', bins=50)
```

```

63 plt.yscale('log') # Log scale handles the heavy skew towards 100
64 plt.title('Distribution of Quality Rating (Log Scale)')
65 plt.xlabel('Quality Rating (0-100)')
66 plt.ylabel('Count (Log Scale)')
67 plt.grid(True, which="both", ls="-", alpha=0.2)
68 plt.show()
69
70
71 #CELL 4
72 # Define Features (X) and Target (y)
73 # Drop the target column to isolate features
74 X = df.drop('Quality Rating', axis=1)
75 # Isolate the target variable
76 y = df['Quality Rating']
77
78 # Split the data into Training and Testing sets
79 # test_size=0.2: 20% of data is held back for testing
80 # random_state=42: Ensures the split is reproducible every time the code runs
81 X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state
=42)
82
83 # Feature Scaling
84 # Standardize features by removing the mean and scaling to unit variance.
85 # This is crucial for Linear Regression to ensure coefficients are comparable
86 # and helps gradient descent converge faster.
87 scaler = StandardScaler()
88
89 # Fit on training set only to prevent data leakage, then transform both
90 X_train_scaled = scaler.fit_transform(X_train)
91 X_test_scaled = scaler.transform(X_test)
92
93 print(f"Training Data Shape: {X_train.shape}")
94 print(f"Testing Data Shape: {X_test.shape}")
95
96
97 #CELL 5
98 # Initialize the model
99 lr_model = LinearRegression()
100
101 # Start Timer to measure computational cost
102 start_time = time.time()
103
104 # Train the model on the scaled training data
105 lr_model.fit(X_train_scaled, y_train)
106
107 # End Timer
108 end_time = time.time()
109 lr_training_time = end_time - start_time
110
111 # Generate predictions on the unseen test set
112 y_pred_lr = lr_model.predict(X_test_scaled)
113
114 # --- Evaluation Metrics ---
115 # RMSE: Root Mean Squared Error (penalizes large errors heavily)
116 rmse_lr = np.sqrt(mean_squared_error(y_test, y_pred_lr))
117 # MAE: Mean Absolute Error (average magnitude of errors)
118 mae_lr = mean_absolute_error(y_test, y_pred_lr)
119 # R2: Coefficient of Determination (variance explained by the model)
120 r2_lr = r2_score(y_test, y_pred_lr)
121
122 print("--- Linear Regression Results ---")
123 print(f"Training Time: {lr_training_time:.6f} seconds")
124 print(f"RMSE: {rmse_lr:.4f}")
125 print(f"MAE: {mae_lr:.4f}")
126 print(f"R^2: {r2_lr:.4f}")
127
128 # --- Coefficient Analysis ---
129 # Shows how much the target changes for a 1-unit change in the feature (when scaled)

```

```

130 coefficients = pd.DataFrame({'Feature': X.columns, 'Coefficient': lr_model.coef_})
131 # Sorting by absolute value to see the most impactful features first
132 coefficients = coefficients.sort_values(by='Coefficient', key=abs, ascending=False)
133 display(coefficients)
134
135
136 #CELL 6
137 # Initialize the model
138 # n_estimators=100: Creates 100 decision trees
139 # random_state=67: Ensures reproducibility
140 rf_model = RandomForestRegressor(n_estimators=100, random_state=67)
141
142 # Start Timer
143 start_time = time.time()
144
145 # Train the model
146 rf_model.fit(X_train_scaled, y_train)
147
148 # End Timer
149 end_time = time.time()
150 rf_training_time = end_time - start_time
151
152 # Predict on test set
153 y_pred_rf = rf_model.predict(X_test_scaled)
154
155 # --- Evaluation Metrics ---
156 rmse_rf = np.sqrt(mean_squared_error(y_test, y_pred_rf))
157 mae_rf = mean_absolute_error(y_test, y_pred_rf)
158 r2_rf = r2_score(y_test, y_pred_rf)
159
160 print("--- Random Forest Results ---")
161 print(f"Training Time: {rf_training_time:.6f} seconds")
162 print(f"RMSE: {rmse_rf:.4f}")
163 print(f"MAE: {mae_rf:.4f}")
164 print(f"R^2: {r2_rf:.4f}")
165
166 # --- Feature Importance ---
167 # Unlike coefficients, this shows how much a feature decreases impurity across trees.
168 # It captures non-linear importance.
169 feature_importances = pd.DataFrame({'Feature': X.columns, 'Importance': rf_model.
feature_importances_})
170 feature_importances = feature_importances.sort_values(by='Importance', ascending=False)
171
172 plt.figure(figsize=(10, 5))
173 sns.barplot(x='Importance', y='Feature', data=feature_importances, palette='viridis')
174 plt.title('Random Forest Feature Importance')
175 plt.show()
176
177
178 #CELL 7
179 # Create a comparison dataframe
180 comparison_df = pd.DataFrame({
181     'Metric': ['RMSE (Lower is better)', 'MAE (Lower is better)', 'R^2 (Higher is better)'],
182     'Linear Regression': [rmse_lr, mae_lr, r2_lr, lr_training_time],
183     'Random Forest': [rmse_rf, mae_rf, r2_rf, rf_training_time]
184 })
185
186 print("Model Performance Comparison:")
187 display(comparison_df)
188
189 # Visualization of Predictions vs Actuals
190 plt.figure(figsize=(14, 6))
191
192 # Plot Linear Regression
193 plt.subplot(1, 2, 1)
194 plt.scatter(y_test, y_pred_lr, alpha=0.5, color='blue')
195 plt.plot([y.min(), y.max()], [y.min(), y.max()], 'r--', lw=2)

```

```

196 plt.xlabel('Actual Quality')
197 plt.ylabel('Predicted Quality')
198 plt.title(f'Linear Regression (R^2 = {r2_lr:.2f})')
199
200 # Plot Random Forest
201 plt.subplot(1, 2, 2)
202 plt.scatter(y_test, y_pred_rf, alpha=0.5, color='green')
203 plt.plot([y.min(), y.max()], [y.min(), y.max()], 'r--', lw=2)
204 plt.xlabel('Actual Quality')
205 plt.ylabel('Predicted Quality')
206 plt.title(f'Random Forest (R^2 = {r2_rf:.2f})')
207
208 plt.tight_layout()
209 plt.show()
210
211
212 #CELL 8
213 # --- A. Direct Metric Comparison ---
214 comparison_df = pd.DataFrame({
215     'Metric': ['RMSE (Lower is better)', 'MAE (Lower is better)', 'R^2 (Higher is better)'],
216     'Linear Regression': [rmse_lr, mae_lr, r2_lr, lr_training_time],
217     'Random Forest': [rmse_rf, mae_rf, r2_rf, rf_training_time]
218 })
219
220 print("Model Performance Comparison:")
221 display(comparison_df)
222
223 # --- B. Predictions vs Actuals Plot ---
224 # Ideally, points should fall exactly on the red diagonal line.
225 plt.figure(figsize=(14, 6))
226
227 # Plot Linear Regression
228 plt.subplot(1, 2, 1)
229 plt.scatter(y_test, y_pred_lr, alpha=0.5, color='blue')
230 plt.plot([y.min(), y.max()], [y.min(), y.max()], 'r--', lw=2) # Identity line
231 plt.xlabel('Actual Quality')
232 plt.ylabel('Predicted Quality')
233 plt.title(f'Linear Regression (R^2 = {r2_lr:.2f})')
234
235 # Plot Random Forest
236 plt.subplot(1, 2, 2)
237 plt.scatter(y_test, y_pred_rf, alpha=0.5, color='green')
238 plt.plot([y.min(), y.max()], [y.min(), y.max()], 'r--', lw=2) # Identity line
239 plt.xlabel('Actual Quality')
240 plt.ylabel('Predicted Quality')
241 plt.title(f'Random Forest (R^2 = {r2_rf:.2f})')
242
243 plt.tight_layout()
244 plt.show()
245
246 # --- C. Cross-Validation (Robustness Check) ---
247 print("\n--- 5-Fold Cross-Validation Results ---")
248 # This tests the model on 5 different splits of data to ensure the score isn't a fluke.
249 # We use negative MSE because sklearn optimization minimizes cost functions.
250
251 cv_scores_lr = cross_val_score(lr_model, X_train_scaled, y_train, cv=5, scoring='
neg_mean_squared_error')
252 cv_rmse_lr = np.sqrt(-cv_scores_lr)
253 print(f"Linear Regression CV Average RMSE: {cv_rmse_lr.mean():.4f} (+/- {cv_rmse_lr.std
():.4f})")
254
255 cv_scores_rf = cross_val_score(rf_model, X_train_scaled, y_train, cv=5, scoring='
neg_mean_squared_error')
256 cv_rmse_rf = np.sqrt(-cv_scores_rf)
257 print(f"Random Forest CV Average RMSE: {cv_rmse_rf.mean():.4f} (+/- {cv_rmse_rf.std
():.4f})")
258

```

```

259 # --- D. Residual Analysis ---
260 # Residuals = Actual - Predicted.
261 # Patterns in residuals imply the model missed underlying trends.
262 residuals_lr = y_test - y_pred_lr
263 residuals_rf = y_test - y_pred_rf
264
265 plt.figure(figsize=(14, 6))
266
267 # LR Residuals
268 plt.subplot(1, 2, 1)
269 sns.scatterplot(x=y_pred_lr, y=residuals_lr, alpha=0.5, color='blue')
270 plt.axhline(0, color='red', linestyle='--', lw=2)
271 plt.title('Linear Regression Residuals')
272 plt.xlabel('Predicted Quality Rating')
273 plt.ylabel('Error (Actual - Predicted)')
274
275 # RF Residuals
276 plt.subplot(1, 2, 2)
277 sns.scatterplot(x=y_pred_rf, y=residuals_rf, alpha=0.5, color='green')
278 plt.axhline(0, color='red', linestyle='--', lw=2)
279 plt.title('Random Forest Residuals')
280 plt.xlabel('Predicted Quality Rating')
281 plt.ylabel('Error (Actual - Predicted)')
282
283 plt.tight_layout()
284 plt.show()
285
286 # --- E. Real-World Tolerance Test ---
287 # Calculates the percentage of predictions that fell within +/- 1.0 of the actual rating
288 .
289 tolerance = 1.0
290 lr_accuracy = np.mean(np.abs(residuals_lr) < tolerance) * 100
291 rf_accuracy = np.mean(np.abs(residuals_rf) < tolerance) * 100
292
293 print("\n--- Real-World Applicability: Tolerance Test ---")
294 print(f"Percentage of predictions within +/- {tolerance} point of actual quality:")
295 print(f"Linear Regression: {lr_accuracy:.2f}%")
296 print(f"Random Forest: {rf_accuracy:.2f}%")

```

Listing 1: Python Implementation for Manufacturing Quality Prediction

## References

- [1] Sarin K. "Manufacturing Quality Prediction Dataset." Kaggle, 2024. <https://www.kaggle.com/code/sarink96/manufacturing-quality-prediction/input>