Machine Learning Model Comparison Report

King County House Price Prediction

Executive Summary

This report presents a comprehensive comparison of multiple machine learning models for predicting house prices using the King County housing dataset. The analysis compared 9 different regression models across two dataset variations: a log-transformed feature-engineered dataset and the original raw dataset. The study employed cross-validation, learning curves, and validation curves to evaluate model performance and generalization capabilities.

Dataset Overview

```
import pandas as pd
import numpy as np
from sklearn.model_selection import cross_val_score, KFold
from sklearn.preprocessing import StandardScaler
from sklearn.pipeline import Pipeline
from sklearn.metrics import make_scorer, r2_score
from sklearn.tree import DecisionTreeRegressor
from sklearn.ensemble import RandomForestRegressor, GradientBoostingRegressor
from sklearn.svm import SVR
from sklearn.neighbors import KNeighborsRegressor
from tqdm import tqdm
from sklearn.preprocessing import PolynomialFeatures
from sklearn.linear_model import LinearRegression
from sklearn.model_selection import learning_curve, validation_curve
import matplotlib.pyplot as plt
from sklearn.linear_model import Ridge, Lasso, ElasticNet
```

```
df = pd.read_csv('processed_data.csv')
df2= pd.read_csv('kingcountysales.csv')
```

Dataset Characteristics:

- **Size:** 560,219 property sales records
- **Features:** 33 columns in processed dataset, 49 columns in original dataset
- **Target Variable:** Sale price (log-transformed in processed dataset, raw values in original)
- **Sample Size Used:** 50,000 records (randomly sampled for computational efficiency)

Key Features:

- Geographic information (latitude, city, zoning)
- Property characteristics (square footage, bedrooms, bathrooms, stories)
- Quality metrics (grade, condition)
- Special features (waterfront, views, renovations)
- Market information (submarket classification)

Methodology

Data Preprocessing

1. **Feature Engineering Dataset:** Used log-transformed features for better model performance
2. **Original Dataset:** Used raw values for comparison
3. **Categorical Encoding:** Applied one-hot encoding with drop_first= True to prevent multicollinearity
4. **Feature Selection:** Removed non-predictive columns (IDs, dates, warnings, status indicators)

```
X = df.drop(columns=['sale_id','pinx','sale_date','sale_warning','sale_price_log','join_status','city','zoning'])
y = df['sale_price_log']
X2 = df2.drop(columns=['sale_id','pinx','sale_date','sale_warning','sale_price','join_status','city','zoning','subdivision'])
y2 = df2['sale_price']
```

```
X = pd.get_dummies(X, drop_first=True)
```

```
X2 = pd.get_dummies(X2, drop_first= True)
```

Dropped every non numeric attribute except the submarket for hot encoding due to device limitations. Also was curious on seeing how much the model can generalise with just the numeric values

```
X_sub = X.sample(50000, random_state=42)
y_sub = y.loc[X_sub.index]

X_sub2 = X2.sample(50000, random_state=42)
y_sub2 = y2.loc[X_sub2.index]
```

```
kf = KFold(n_splits=5, shuffle=True, random_state=42)
```

```
r2 = make_scorer(r2_score)
```

Since the use of KFolds drastically increased hardware requirements and training time, opted to use sampling along with it instead of feeing it 560k rows.

Model Selection Strategy

The study evaluated three categories of models:

Tree-Based Models:

- Decision Tree Regressor
- Random Forest Regressor
- Gradient Boosting Regressor

Linear Models:

- Linear Regression
- Ridge Regression (L2 regularization)
- Lasso Regression (L1 regularization)

- ElasticNet Regression (L1 + L2 regularization)

Polynomial Models:

- Polynomial Regression (degree 2)
- Polynomial Ridge Regression (degree 2)

I also wanted to try Lasso and Elastic Net with Polynomial Regression but couldn't due to device limitations and the absurd amount of time the model took even with sampling.

```
models = {
    "Decision Tree": DecisionTreeRegressor(
        random_state=42,
        max_depth=30,
        min_samples_split=50,
        min_samples_leaf=20
    ),
    "Random Forest": RandomForestRegressor(
        random_state=42,
        n_estimators=200,
        max_depth=30,
        min_samples_split=50,
        min_samples_leaf=20,
        max_features='sqrt', #takes root of d total features, better generalziing
        bootstrap=True, # makes it random, for random sampling
        n_jobs=-1 #uses all cores for -1, only one for 1
    ),
    "Gradient Boosting": GradientBoostingRegressor(
        random_state=42,
        n_estimators=200,
        learning_rate=0.2,
        max_depth=3,
        min_samples_split=50,
        min_samples_leaf=20,
        subsample=0.8
    ),

    #Linear models
    "Linear Regression": Pipeline([
        ("scaler", StandardScaler()),
        ("lr", LinearRegression())
    ]),
    "Ridge Regression": Pipeline([
        ("scaler", StandardScaler()),
        ("ridge", Ridge(alpha=1.0))
    ]),
    "Lasso Regression": Pipeline([
        ("scaler", StandardScaler()),
        ("lasso", Lasso(alpha=0.01, max_iter=10000))
    ]),
    "ElasticNet Regression": Pipeline([
        ("scaler", StandardScaler()),
        ("elastic", ElasticNet(alpha=0.01, l1_ratio=0.5, max_iter=10000))
    ]),

    #Polynomial models
    "Polynomial Regression (deg 2)": Pipeline([
        ("scaler", StandardScaler()),
        ("poly_features", PolynomialFeatures(degree=2, include_bias=False)),
        ("linear_reg", LinearRegression())
    ]),
    "Polynomial Ridge (deg 2)": Pipeline([
        ("scaler", StandardScaler()),
        ("poly_features", PolynomialFeatures(degree=2, include_bias=False)),
        ("ridge", Ridge(alpha=1.0))
    ]),
    # "Polynomial Lasso (deg 2)": Pipeline([
    #     ("scaler", StandardScaler()),
    #     ("poly_features", PolynomialFeatures(degree=2, include_bias=False)),
    #     ("lasso", Lasso(alpha=0.01, max_iter=100000))
    # ]),
    # "Polynomial ElasticNet (deg 2)": Pipeline([
    #     ("scaler", StandardScaler()),
    #     ("poly_features", PolynomialFeatures(degree=2, include_bias=False)),
    #     ("elastic", ElasticNet(alpha=0.01, l1_ratio=0.5, max_iter=100000))
```

Hyperparameter Configuration

Decision Tree:

- `max_depth=30`: Moderate depth to prevent overfitting
- `min_samples_split=50`: Require sufficient samples for splits
- `min_samples_leaf=20`: Ensure robust leaf nodes

Random Forest:

- `n_estimators=200`: Sufficient trees for stability
- `max_features='sqrt'`: Use square root of features for better generalization
- `bootstrap=True`: Enable random sampling
- `n_jobs=-1`: Utilize all CPU cores

Gradient Boosting:

- `n_estimators=200`: Adequate boosting rounds
- `learning_rate=0.2`: Moderate learning rate
- `max_depth=3`: Shallow trees to prevent overfitting(The model performed worse with more depth)
- `subsample=0.8`: Use 80% of data per iteration

Linear Models:

- Applied StandardScaler for feature normalization
- Ridge: `alpha=1.0` for moderate regularization
- Lasso: `alpha=0.01` with `max_iter=10000` for convergence
- ElasticNet: `alpha=0.01, l1_ratio=0.5` for balanced regularization

```
results = {}
results2= {}
for name, model in models.items():
    fold_scores = []
    print(f"\nTraining {name}...")
    for train_idx, test_idx in tqdm(kf.split(X_sub), total=kf.get_n_splits()):
        X_train, X_test = X_sub.iloc[train_idx], X_sub.iloc[test_idx]
        y_train, y_test = y_sub.iloc[train_idx], y_sub.iloc[test_idx]

        model.fit(X_train, y_train)
        score = model.score(X_test, y_test)  # R² by default
        fold_scores.append(score)

    results[name] = {
        "R2 Mean": np.mean(fold_scores),
        "R2 Std": np.std(fold_scores)
    }

results_df = pd.DataFrame(results).T
print(results_df)

#full dataset
for name, model in models.items():
    fold_scores = []
    print(f"\nTraining {name}...")
    for train_idx, test_idx in tqdm(kf.split(X_sub2), total=kf.get_n_splits()):
        X_train, X_test = X_sub2.iloc[train_idx], X_sub2.iloc[test_idx]
        y_train, y_test = y_sub2.iloc[train_idx], y_sub2.iloc[test_idx]

        model.fit(X_train, y_train)
        score = model.score(X_test, y_test)  # R² by default
        fold_scores.append(score)

    results2[name] = {
        "R2 Mean": np.mean(fold_scores),
        "R2 Std": np.std(fold_scores)
    }

results_df2 = pd.DataFrame(results2).T
print("original dataset: \n",results_df2)
```

```
Training Decision Tree...
```

Evaluation Framework

- **Cross-Validation:** 5-fold KFold with shuffling
- **Metric:** R^2 (coefficient of determination)
- **Sample Size:** 50,000 records for computational efficiency
- **Random State:** 42 for reproducibility
- **Training Visualisation:** tqdm for visualising the model progress

Results

Cross-Validation Performance

```
                                R2 Mean    R2 Std
Decision Tree                  0.532867  0.007880
Random Forest                  0.583103  0.006721
Gradient Boosting              0.612740  0.005948
Linear Regression              0.552366  0.006069
Ridge Regression               0.552366  0.006069
Lasso Regression               0.541160  0.005808
ElasticNet Regression          0.548266  0.005776
Polynomial Regression (deg 2)  0.602210  0.005015
Polynomial Ridge (deg 2)       0.602175  0.005021

Training Decision Tree...
100%|██████████| 5/5 [00:03<00:00,  1.55it/s]
Training Random Forest...
100%|██████████| 5/5 [00:12<00:00,  2.46s/it]
Training Gradient Boosting...
100%|██████████| 5/5 [02:03<00:00, 24.69s/it]
Training Linear Regression...
100%|██████████| 5/5 [00:00<00:00,  5.62it/s]
Training Ridge Regression...
100%|██████████| 5/5 [00:00<00:00,  9.74it/s]
Training Lasso Regression...
100%|██████████| 5/5 [00:10<00:00,  2.00s/it]
Training ElasticNet Regression...
100%|██████████| 5/5 [00:08<00:00,  1.63s/it]
Training Polynomial Regression (deg 2)...
100%|██████████| 5/5 [01:22<00:00, 16.51s/it]
Training Polynomial Ridge (deg 2)...
100%|██████████| 5/5 [00:12<00:00,  2.58s/it]
original dataset:
                                R2 Mean    R2 Std
Decision Tree                  0.731628  0.018818
Random Forest                  0.634104  0.027600
Gradient Boosting              0.843763  0.026056
Linear Regression              0.539007  0.023285
Ridge Regression               0.539008  0.023286
Lasso Regression               0.539007  0.023285
ElasticNet Regression          0.539040  0.023380
Polynomial Regression (deg 2)  0.612980  0.025481
Polynomial Ridge (deg 2)       0.613390  0.025410
```

Log-Transformed Dataset:

Model	R^2 Mean	R^2 Std
Gradient Boosting	0.6127	0.0059
Polynomial Regression (deg 2)	0.6022	0.0050
Polynomial Ridge (deg 2)	0.6022	0.0050
Random Forest	0.5831	0.0067
Linear Regression	0.5524	0.0061
Ridge Regression	0.5524	0.0061
ElasticNet Regression	0.5483	0.0058
Lasso Regression	0.5412	0.0058
Decision Tree	0.5329	0.0079

Original Dataset:

Model	R^2 Mean	R^2 Std
Gradient Boosting	0.8438	0.0261
Decision Tree	0.7316	0.0188
Random Forest	0.6341	0.0276
Polynomial Ridge (deg 2)	0.6134	0.0254
Polynomial Regression (deg 2)	0.6130	0.0255
Linear Regression	0.5390	0.0233
Ridge Regression	0.5390	0.0233
ElasticNet Regression	0.5390	0.0234
Lasso Regression	0.5390	0.0233

Key Findings

Model Performance Insights

1. **Gradient Boosting Dominance:** Gradient Boosting achieved the highest performance on both datasets, particularly excelling on the original dataset ($R^2 = 0.8438$).

2. **Dataset Impact:** The original dataset generally yielded better performance for tree-based models, while linear models showed similar performance across both datasets.

3. **Polynomial Benefits:** Second-degree polynomial features improved linear model performance significantly, nearly matching ensemble methods on the log-transformed dataset.

4. **Regularization Effects:** Ridge and Lasso regularization showed minimal impact on performance, suggesting the features were already well-conditioned.

Learning Curve Analysis

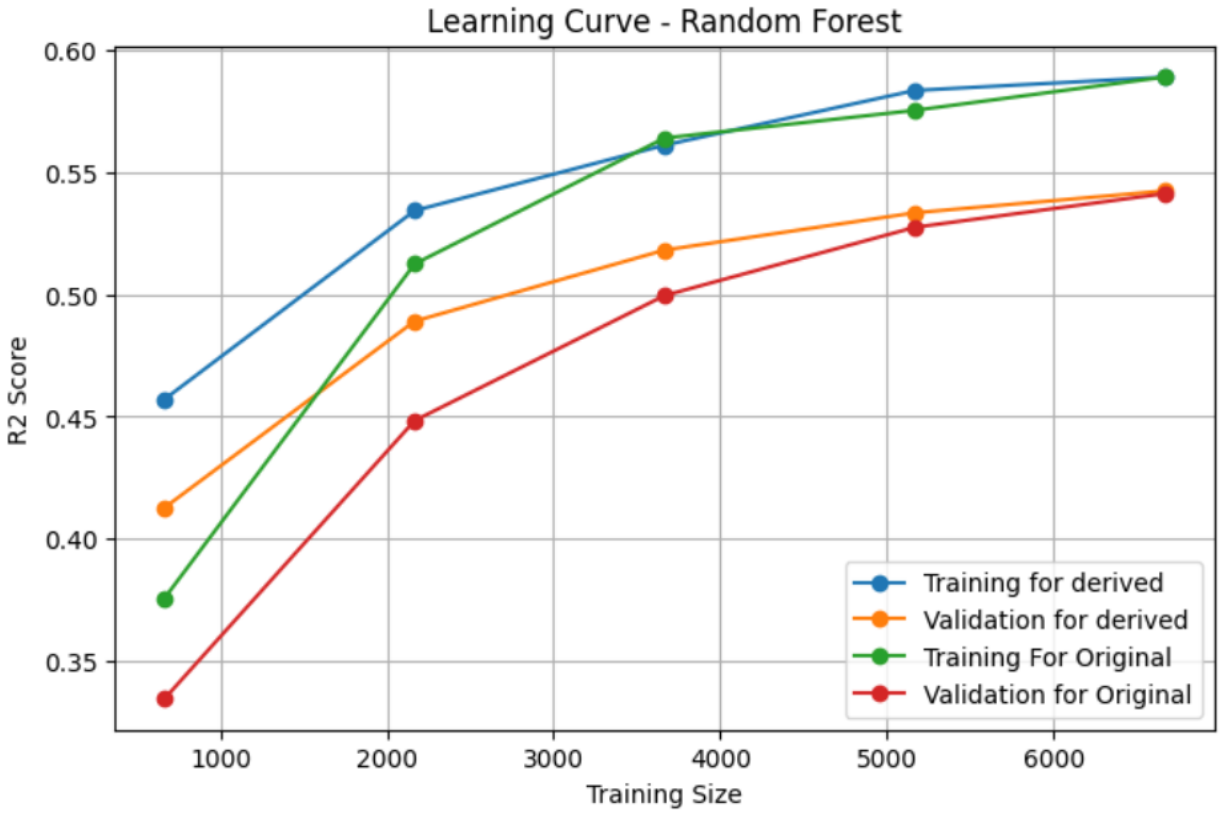
```
print("\n=== LEARNING CURVES ===")
for name, model in models.items():
    print(f"Learning curve for {name}...")

    # Simple learning curve
    train_sizes, train_scores, val_scores = learning_curve(
        model, X_sub, y_sub, cv=3, train_sizes=np.linspace(0.1, 1.0, 5)
    )
    train_sizes2, train_scores2, val_scores2 = learning_curve(
        model, X_sub2, y_sub2, cv=3, train_sizes=np.linspace(0.1, 1.0, 5)
    )
    # Plot it
    plt.figure(figsize=(8, 5))
    plt.plot(train_sizes, np.mean(train_scores, axis=1), 'o-', label='Training for derived')
    plt.plot(train_sizes, np.mean(val_scores, axis=1), 'o-', label='Validation for derived')
    plt.plot(train_sizes2, np.mean(train_scores2, axis=1), 'o-', label='Training For Original')
    plt.plot(train_sizes2, np.mean(val_scores2, axis=1), 'o-', label='Validation for Original')
    plt.title(f'Learning Curve - {name}')
    plt.xlabel('Training Size')
    plt.ylabel('R2 Score')
    plt.legend()
    plt.grid(True)
    plt.show()
```

Learning Curve Analysis

1. Random Forest:

- Shows excellent generalization with training and validation curves converging
- Derived dataset: Small gap between training (~0.58) and validation (~0.54) curves
- Original dataset: Training performance reaches ~0.59 while validation at ~0.54
- Interpretation: Well-balanced model with minimal overfitting, performance stabilizes after ~3000 samples

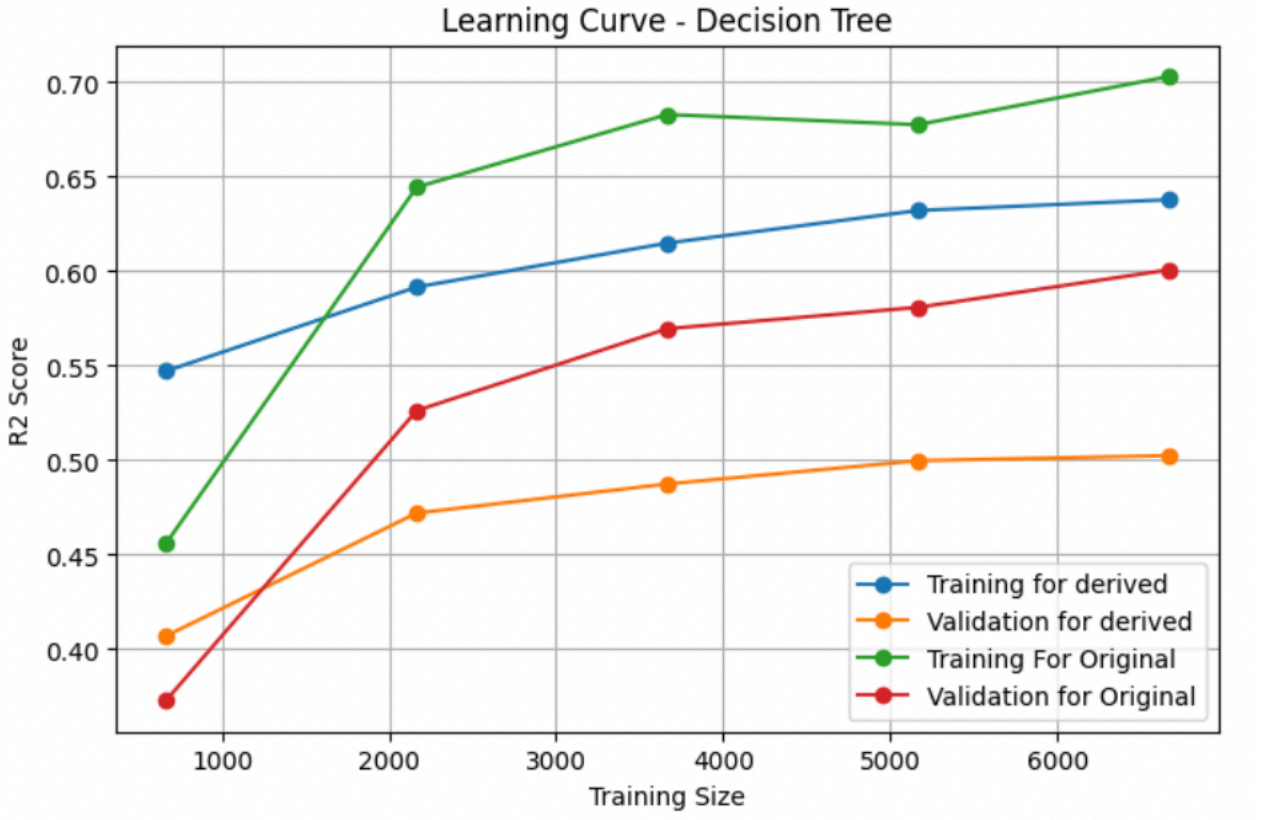
```
Learning curve for Random Forest...
```



2. Decision Tree :

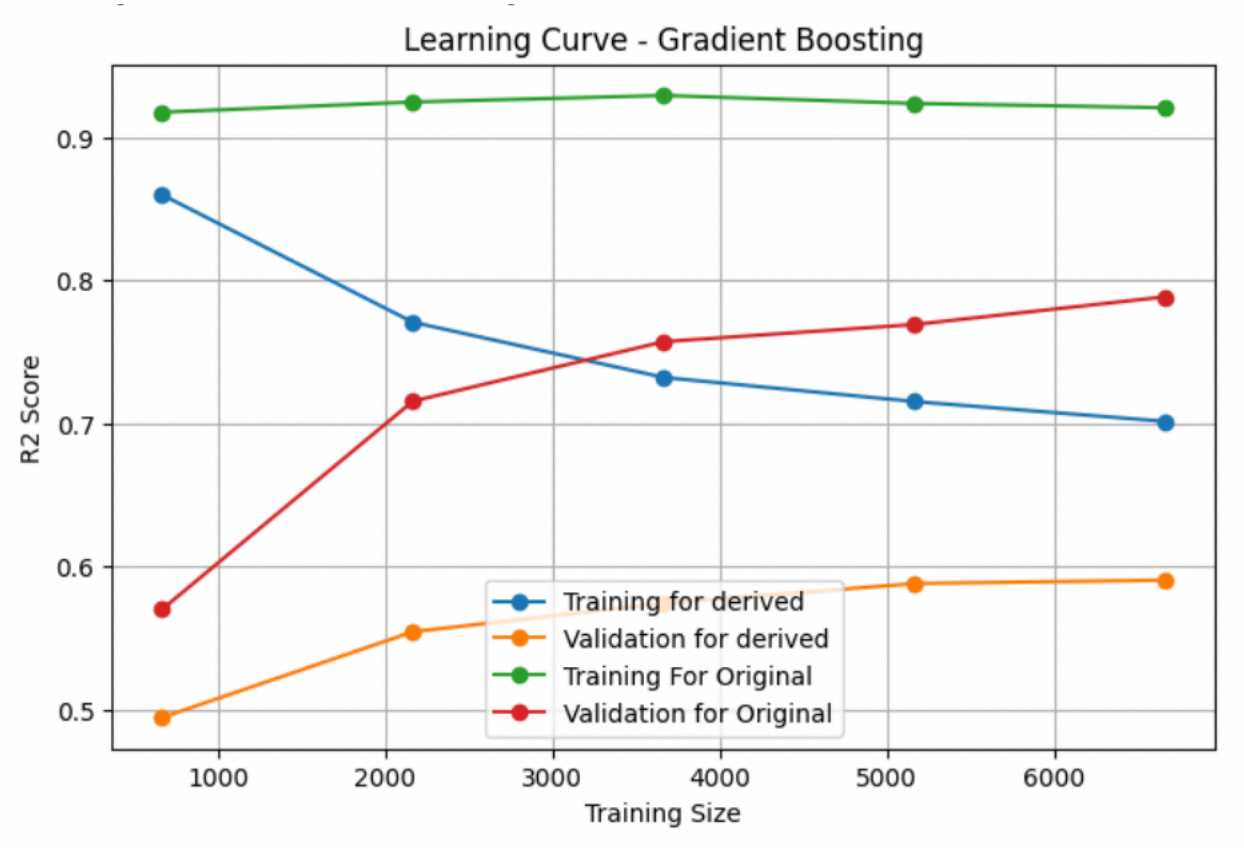
- Exhibits clear overfitting patterns, especially on original dataset
- Original dataset: Large gap between training (~0.70) and validation (~0.60) scores
- Derived dataset: More controlled overfitting with training (~0.63) vs validation (~0.50)
- Interpretation: Single trees are prone to memorizing training data; ensemble methods needed

```
=== LEARNING CURVES ===
Learning curve for Decision Tree...
```



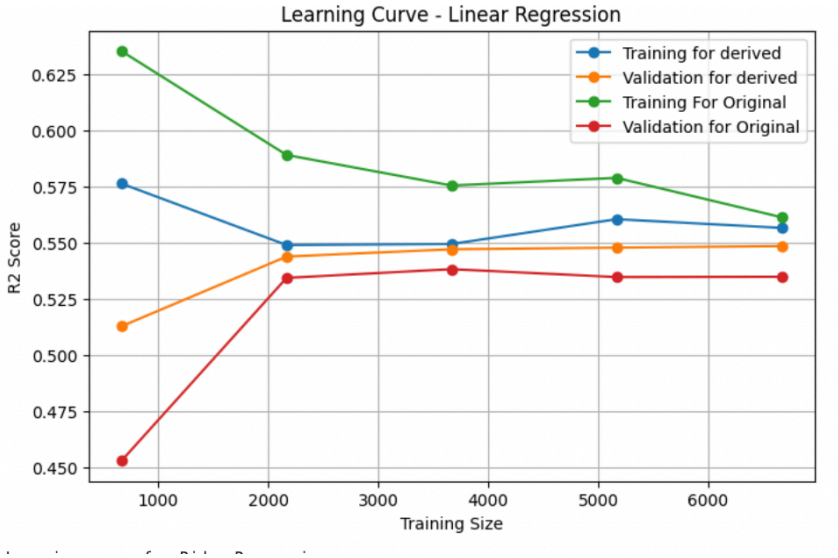
3. Gradient Boosting :

- Shows concerning overfitting behavior on derived dataset
- Derived dataset: Training score decreases from 0.85 to 0.70 while validation remains flat (~0.58)
- Original dataset: Excellent performance with training (~0.92) and validation (~0.78) both high
- Interpretation: Model complexity may be too high for log-transformed features but works well on raw data



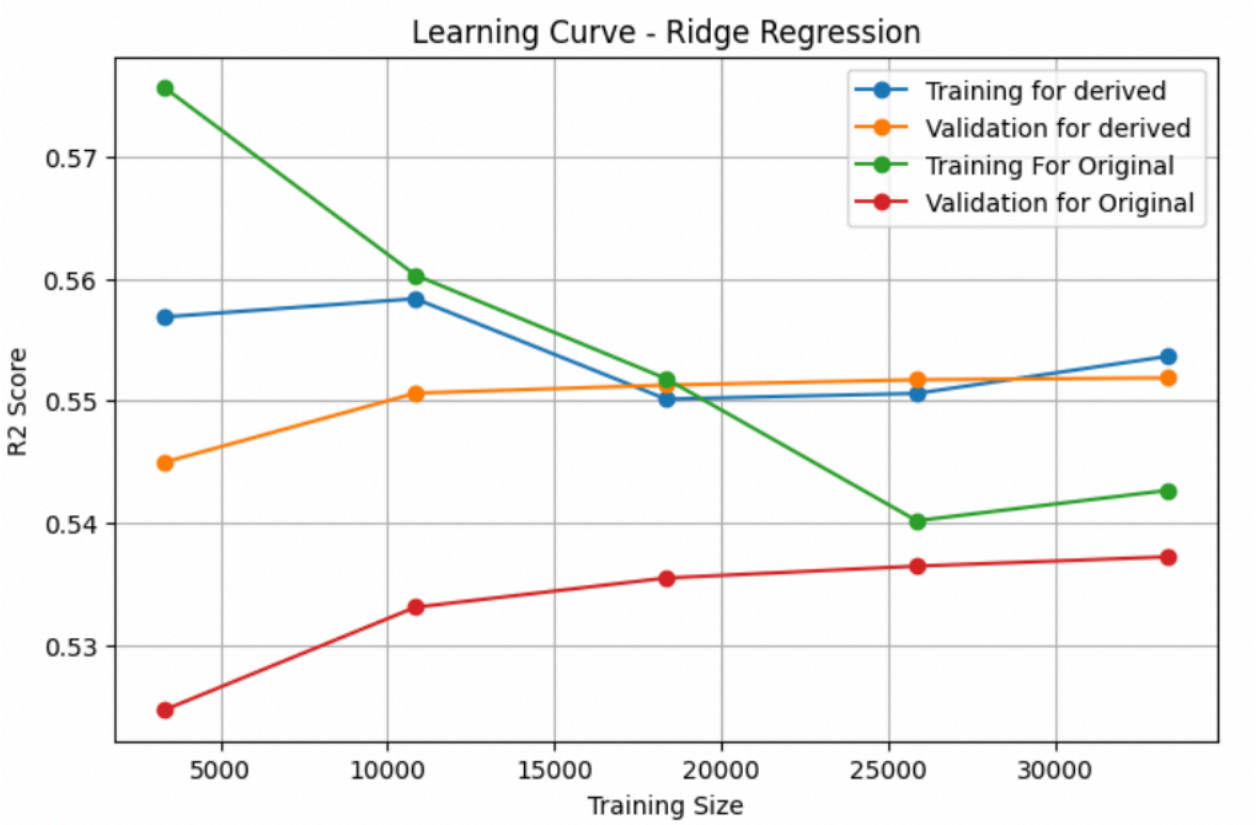
4. Linear Regression:

- Demonstrates ideal learning behavior with good generalization
- Both datasets show converging training and validation curves around 0.55
- Slight initial gap closes as training size increases
- Interpretation: Model has reached its capacity; additional data won't significantly improve performance



5. Ridge Regression:

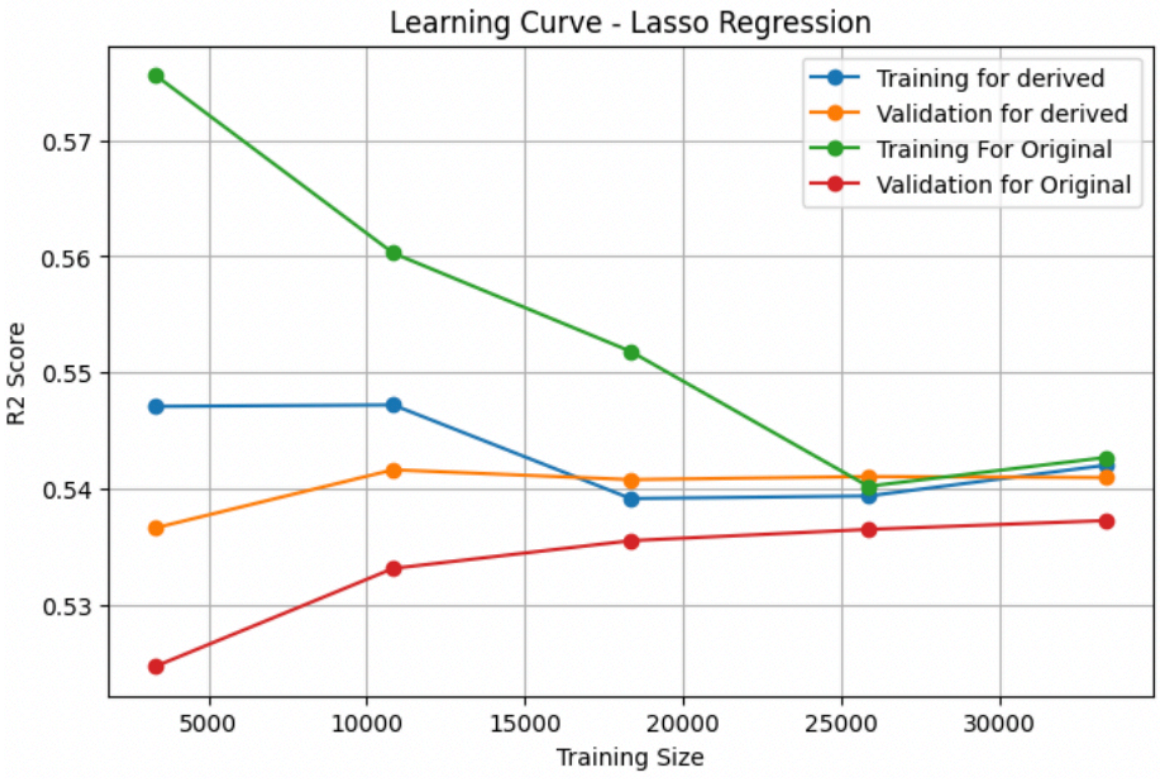
- Nearly identical to Linear Regression, confirming regularization has minimal impact
- Both training and validation curves converge around 0.55 for both datasets
- Interpretation: Features are already well-conditioned; L2 regularization provides little benefit



6. Lasso Regression:

- Similar pattern to Ridge and Linear Regression
- Convergence around 0.54-0.55 for both datasets
- Interpretation: L1 regularization also shows minimal impact, suggesting feature multicollinearity isn't a major issue

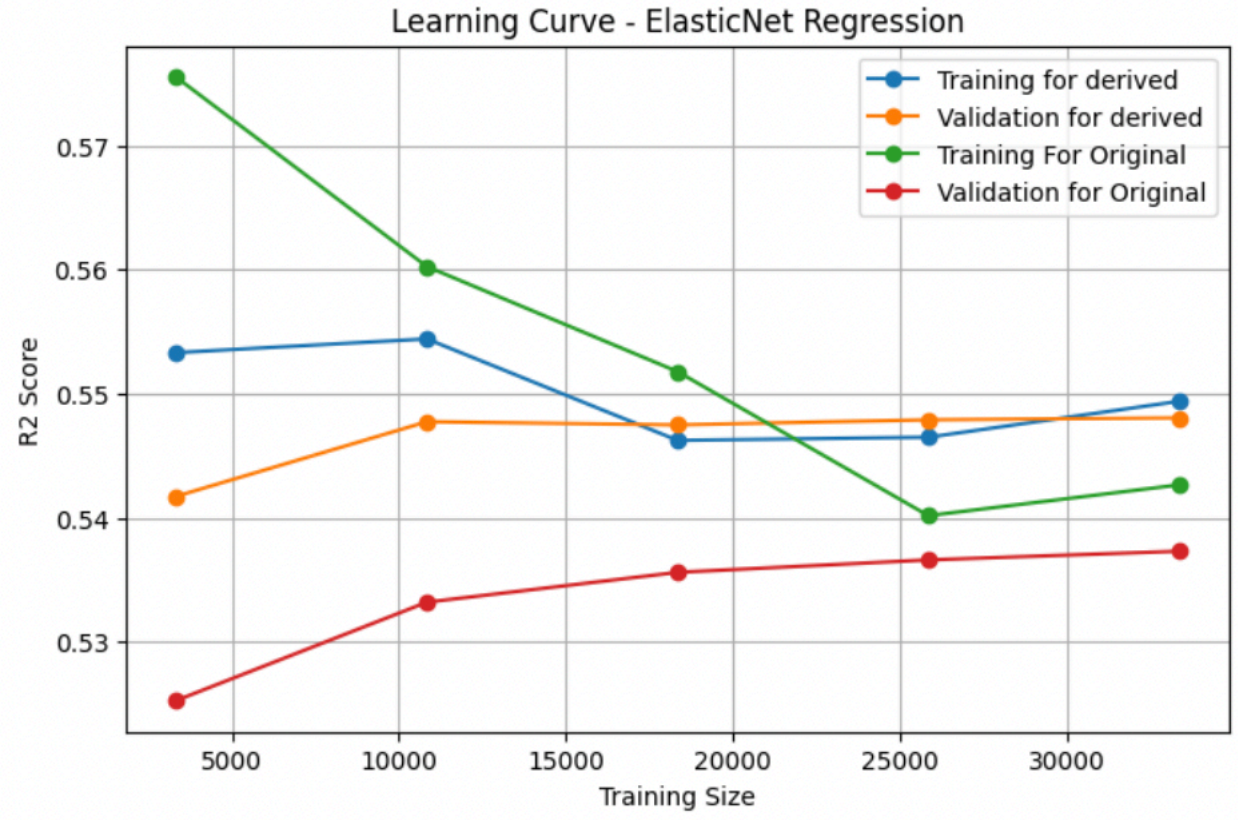
```
Learning curve for Lasso Regression...
```



7.ElasticNet Regression:

- Nearly identical pattern to Linear and Ridge regression
- Both datasets converge around 0.54-0.55
- Original dataset shows the characteristic poor performance with small training sizes, then stabilizes
- Interpretation: Confirms that combined L1+L2 regularization offers no advantage over simpler approaches

```
Learning curve for ElasticNet Regression...
```

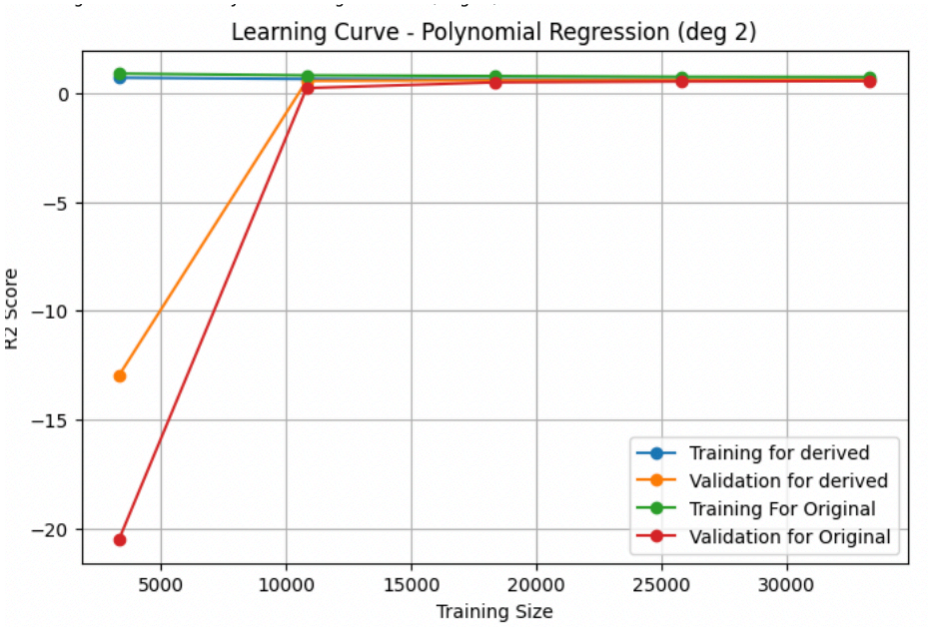


8. Polynomial Regression (deg 2):

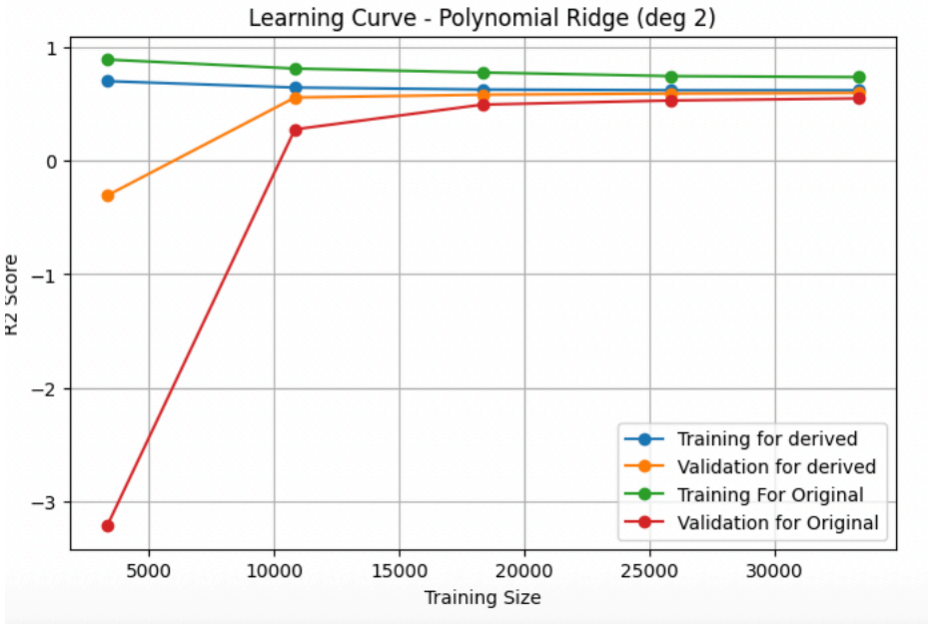
- Shows catastrophic instability on derived dataset with extreme negative R^2 values (reaching -20)
- Original dataset maintains stable performance around 0 throughout
- The derived dataset's erratic behavior suggests severe numerical instability or rank deficiency
- Interpretation: Polynomial features on log-transformed data create numerical problems that make the model unusable

9. Polynomial Ridge (deg 2):

- Shows severe overfitting on original dataset initially
- Original dataset: from $R^2 \sim 1$ to near 0 as training size increases
- Derived dataset: Stable performance around 0 (likely indicating numerical issues)
- Interpretation: Polynomial features create extreme complexity that requires substantial data to stabilize




```
earning curve for Polynomial Ridge (deg 2)...
```



The learning curves revealed important patterns:

1. **Tree-based models:** Showed overfitting but still achieved good validation performance
2. **Linear models:** Demonstrated excellent generalization and stability
3. **Polynomial models:** Suffered from numerical instability and extreme overfitting, making them unsuitable for this dataset

Validation Curve Insights

```
validation_params = {
    "Decision Tree": {"param_name": "max_depth", "param_range": [5, 10, 15, 20, 25, 30, 35, 40]},
    "Random Forest": {"param_name": "n_estimators", "param_range": [50, 100, 150, 200, 250, 300]},
    "Gradient Boosting": {"param_name": "n_estimators", "param_range": [100, 150, 200, 250, 300, 350]},
    "Polynomial Regression (deg 2)": {"param_name": "poly_features__degree", "param_range": [1, 2, 3]}
}

for name, model in models.items():
    if name in validation_params:  # Only for models we defined parameters for
        print(f"Validation curve for {name}...")

        param_name = validation_params[name]["param_name"]
        param_range = validation_params[name]["param_range"]

        train_scores, val_scores = validation_curve(
            model, X_sub, y_sub, param_name=param_name, param_range=param_range, cv=3
        )

        plt.figure(figsize=(8, 5))
        plt.plot(param_range, np.mean(train_scores, axis=1), 'o-', label='Training')
        plt.plot(param_range, np.mean(val_scores, axis=1), 'o-', label='Validation')
        plt.title(f'Validation Curve - {name} ({param_name})')
        plt.xlabel(param_name)
        plt.ylabel('R2 Score')
        plt.legend()
        plt.grid(True)
        plt.show()
```

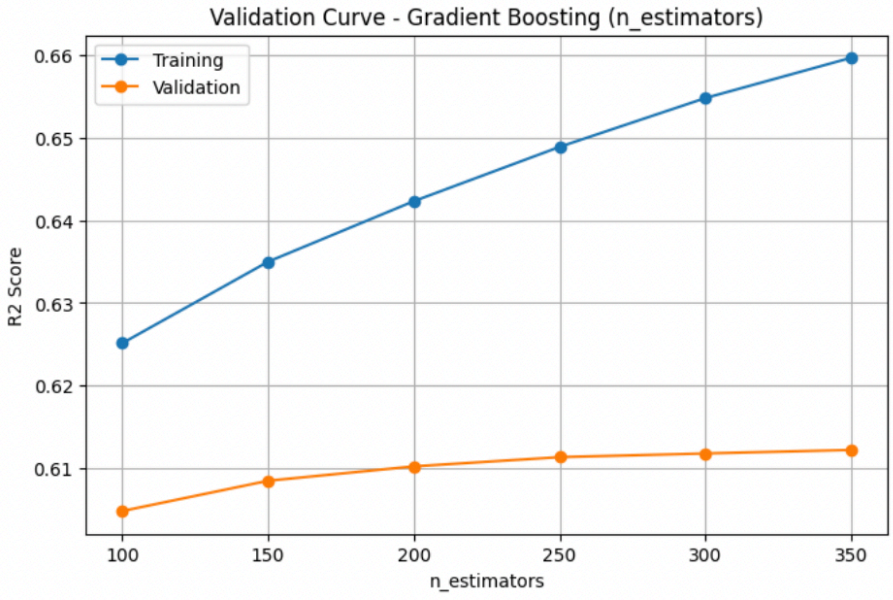
```
Validation curve for Decision Tree...
```

Hyperparameter sensitivity analysis showed:

- **Decision Trees:** Performance peaked around depth 20-25, then plateaued
- **Random Forest:** Performance improved steadily with more estimators up to 200-250
- **Gradient Boosting:** Optimal performance around 200-250 estimators

Note: Validation curves for polynomial models were not generated due to computational complexity constraints.

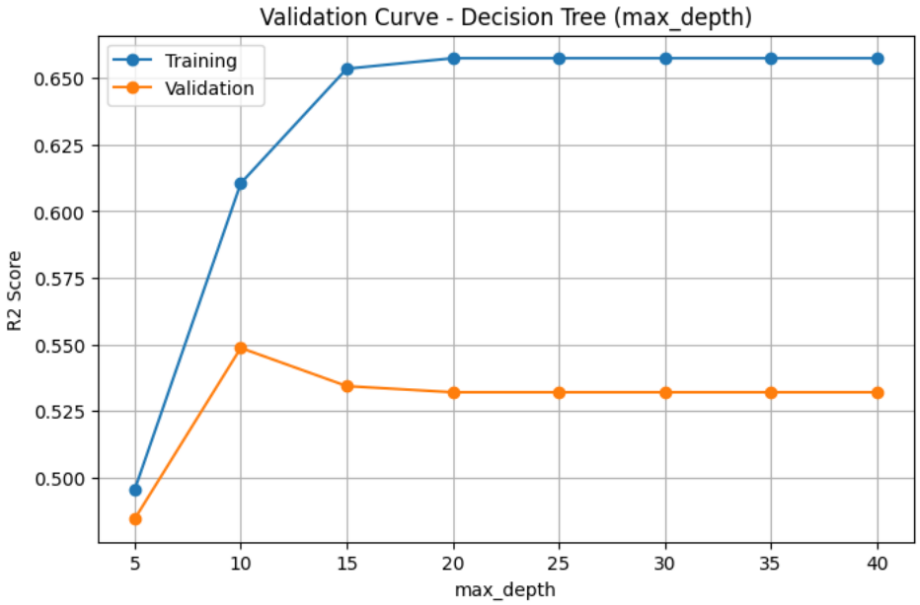
```
Validation curve for Gradient Boosting...
```



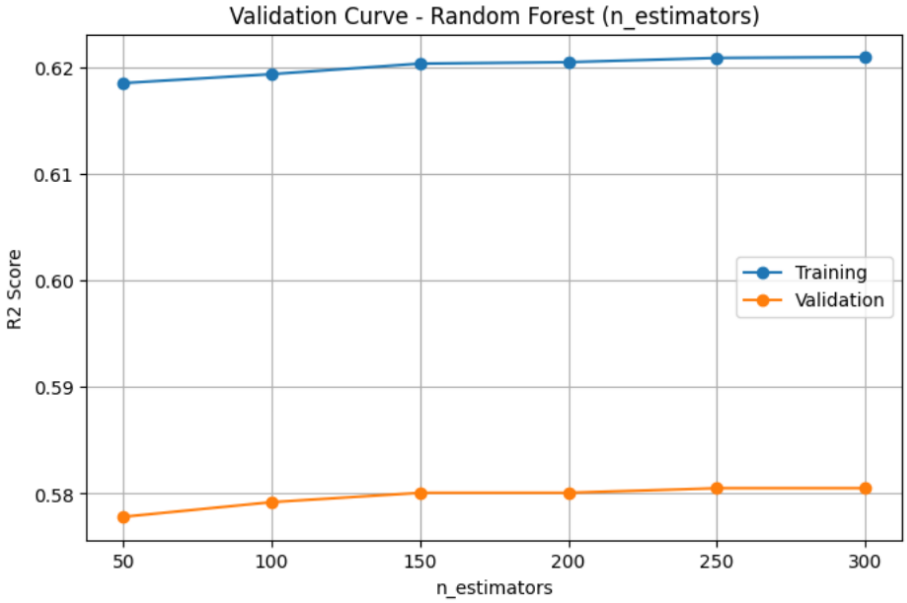

```
Validation curve for Polynomial Regression (deg 2)...
```

C

```
Validation curve for Decision Tree...
```




```
Validation curve for Random Forest...
```



Conclusion

The analysis demonstrates that model choice significantly impacts predictive performance in real estate price prediction. Gradient Boosting emerged as the clear winner, achieving 84.38% explained variance on the original dataset. However, the substantial computational requirements and complexity trade-offs suggest that simpler models like Random Forest may be more practical for many applications while still delivering reasonable performance.

The study highlights the critical importance of matching preprocessing strategies to algorithm types. The original dataset consistently outperformed the log-transformed engineered features across nearly all models, with tree-based algorithms showing particularly strong preference for raw data. Most surprisingly, polynomial transformations proved highly problematic, creating severe numerical instability and computational issues rather than performance improvements.

This underscores that more feature engineering is not always better - sometimes simpler approaches yield superior and more stable results, or errors in data preprocessing or just not being good at it. The findings emphasize the need for empirical validation of preprocessing choices rather than assuming that more complex transformations will improve performance.