Muhammet Ali YILDIZ
1709981

Model Selection and Ridge Regression Performance Evaluation

Polytechnic Institute of Guarda

1. Introduction

This study evaluates various regression models to predict CO_2 levels using gas data. The objective is to identify the best-performing model based on both accuracy and computational efficiency, and thenprovide a detailed evaluation of the selected model.

2. Model Selection Process

We initially evaluated a range of regression models, including **`Linear Regression`**, **`Ridge Regression`**, **`Lasso Regression`**, **`ElasticNet`**, **`Decision Tree Regressor`**, **`Random Forest Regressor`**, **`Gradient Boosting Regressor`**, **`KNeighbors Regressor`**, **`SVR`**, and **`MLP Regressor`**. The selection process involved the following steps:

Model Selection Code :

```python
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
from sklearn.metrics import mean_squared_error
from sklearn.linear_model import LinearRegression, Ridge
from sklearn.tree import DecisionTreeRegressor
from sklearn.ensemble import RandomForestRegressor, GradientBoostingRegressor
from sklearn.neighbors import KNeighborsRegressor
from sklearn.svm import SVR
from sklearn.preprocessing import StandardScaler
import time
import timeit


# Function to load the dataset from a file
def load_data(file_path):
    df = pd.read_csv(file_path, header=None, sep=r'\s+')
    df.columns = ['rate1', 'rate2']  # Assign column names
    return df


# Function to create features and targets from the dataset
def prepare_features_and_targets(df, lag):
    X, y = [], []
    for i in range(lag, len(df)):
        X.append(df.iloc[i - lag:i, 0].values)  # Collect past values as features
        y.append(df.iloc[i, 1])  # Collect current value as target
    return np.array(X), np.array(y)


# Function to train and evaluate models
def train_and_evaluate_model(model, X_train, y_train, X_test, y_test):
    # Normalize the features
    scaler = StandardScaler()
    X_train_scaled = scaler.fit_transform(X_train)
    X_test_scaled = scaler.transform(X_test)

    # Measure training time
    start_time = time.time()
    model.fit(X_train_scaled, y_train)
    training_time = time.time() - start_time

    # Measure prediction time
    prediction_time = timeit.timeit(lambda: model.predict(X_test_scaled), number=1)

    # Make predictions
    y_train_pred = model.predict(X_train_scaled)
    y_test_pred = model.predict(X_test_scaled)

    # Calculate RMSE for training and testing data
    train_rmse = np.sqrt(mean_squared_error(y_train, y_train_pred))
    test_rmse = np.sqrt(mean_squared_error(y_test, y_test_pred))

    return train_rmse, test_rmse, y_train_pred, y_test_pred, training_time, prediction_time
```

```python
# Load the dataset
file_path = '/Users/mali/Desktop/Gas_dataset/gas_datasets.txt'
data = load_data(file_path)

# Split the dataset into training and testing sets
split = len(data) // 2
train_set = data.iloc[:split]
test_set = data.iloc[split:]

# Number of past values to consider for features
lag = 5

# Create training and testing features and targets
X_train, y_train = prepare_features_and_targets(train_set, lag)
X_test, y_test = prepare_features_and_targets(test_set, lag)

# Define the models to be evaluated
models = {
    'Linear Regression': LinearRegression(),
    'Ridge Regression': Ridge(),
    'Decision Tree Regressor': DecisionTreeRegressor(random_state=42),
    'Random Forest Regressor': RandomForestRegressor(n_estimators=100, random_state=42),
    'Gradient Boosting Regressor': GradientBoostingRegressor(n_estimators=100, learning_rate=0.1,
max_depth=3,
                                                            random_state=42),
    'KNeighbors Regressor': KNeighborsRegressor(),
    'SVR': SVR(kernel='rbf')
}

results = []
predictions = {}  # To store the predictions for plotting

# Train and evaluate each model
for model_name, model in models.items():
    # Train and evaluate the model
    train_rmse, test_rmse, y_train_pred, y_test_pred, training_time, prediction_time =
train_and_evaluate_model(model,

         X_train,

         y_train,

         X_test,

         y_test)
    # Store the results
    results.append({
        'Model': model_name,
        'Train RMSE': train_rmse,
        'Test RMSE': test_rmse,
        'Training Time (s)': training_time,
        'Prediction Time (s)': prediction_time
    })
    # Store the predictions for plotting
    predictions[model_name] = {
        'y_train_pred': y_train_pred,
        'y_test_pred': y_test_pred
    }
    # Print the results
    print(
        f"{model_name}: Train RMSE = {train_rmse}, Test RMSE = {test_rmse}, Training Time =
{training_time}s, Prediction Time = {prediction_time}s")
```

```python
# Create a DataFrame from the results
results_df = pd.DataFrame(results)

# Normalize the values for comparison
results_df['Normalized Test RMSE'] = results_df['Test RMSE'] / results_df['Test RMSE'].max()
results_df['Normalized Training Time'] = results_df['Training Time (s)'] / results_df['Training Time
(s)'].max()
results_df['Normalized Prediction Time'] = results_df['Prediction Time (s)'] / results_df['Prediction
Time (s)'].max()

# Calculate the total score as the sum of normalized values
results_df['Total Score'] = results_df['Normalized Test RMSE'] + results_df['Normalized Training Time']
+ results_df[
    'Normalized Prediction Time']

# Sort the DataFrame by Total Score to find the best model
sorted_results_df = results_df.sort_values(by='Total Score')

# Display the sorted DataFrame
print("\nModel Performance Sorted by Total Score:")
print(sorted_results_df)

# Determine the model with the lowest Total Score
best_model_row = sorted_results_df.loc[sorted_results_df['Total Score'].idxmin()]
best_model_name = best_model_row['Model']
print(f"\nBest Model (based on Total Score): {best_model_name}")

# Display the best model's results
print(f"\nBest Model Results:\n{best_model_row}")

# Plot predictions vs actual values for the best model
plt.figure(figsize=(14, 7))
plt.plot(train_set.index[lag:], y_train, 'o-', label='Actual (Train)', markersize=5, color='blue')
plt.plot(test_set.index[lag:], y_test, 'd-.', label='Actual (Test)', markersize=5, color='red')

best_model_predictions = predictions[best_model_name]
plt.plot(train_set.index[lag:], best_model_predictions['y_train_pred'], label=f'Predicted
({best_model_name} - Train)',
         linestyle='--')
plt.plot(test_set.index[lag:], best_model_predictions['y_test_pred'], label=f'Predicted
({best_model_name} - Test)',
         linestyle='--')

plt.xlabel('Time', fontsize=12)
plt.ylabel('Rate', fontsize=12)
plt.title(f'Predictions vs Actual Values for {best_model_name}', fontsize=15)
plt.legend()
plt.grid(True, linestyle='--', linewidth=0.5)
plt.show()
```

2.1 Data Preparation

The dataset was split into features and target variables. Specifically:

- **Features (X)**: Previous gas rate values.
- **Target (y)**: Current CO2 rate percentage.

2.2 Model Training and Evaluation

Each model was trained and evaluated using the same dataset. The following metrics were computedfor each model:

- **Training Time**: The time taken to train the model.
- **Prediction Time**: The time taken to make predictions on the test dataset.
- **Train RMSE**: Root Mean Squared Error on the training dataset.
- **Test RMSE**: Root Mean Squared Error on the test dataset.

2.3 Results Comparison

Models were ranked based on their normalized scores for Test RMSE, Training Time, and PredictionTime. The normalization ensured that each metric contributed equally to the final score.

3. Model Evaluation Results

The table below summarizes the performance metrics of each model:

Model	Train RMSE	Test RMSE	Training Time (s)	Prediction Time (s)	Total Score
Ridge Regression	0,53217	1,3186763	0,0022483	0,0000345	0,943471
Linear Regression	0,5219869	1,3020379	0,0097191	0,0000732	1,101468
Decision Tree Regressor	0	1,4290933	0,0019038	0,0002683	1,135255
KNeighbors Regressor	0,5995336	1,4979249	0,001425	0,0003171	1,197739
SVR	0,7441074	1,4665833	0,001205	0,0008995	1,483654
Gradient Boosting Regressor	0,1623941	1,4058745	0,0263481	0,0004165	1,684677
Random Forest Regressor	0,2260337	1,3692675	0,0503199	0,0018716	2,91411

4. Selection of Ridge Regression

After evaluating all models, Ridge Regression was selected for further evaluation due to its balanced performance in terms of accuracy and computational efficiency. The Ridge Regression model demonstrated competitive RMSE values while maintaining low training and prediction times.

5. Detailed Evaluation of Ridge Regression

5.1 Model Training and Evaluation

The Ridge Regression model was trained and evaluated with the following steps:

- **Data Normalization**: Features were scaled using `**StandardScaler**`.
- **Model Training**: The model was trained on the training set.
- **Performance Metrics**: Training RMSE, Test RMSE, training time, and prediction time were calculated.

python

```python
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
from sklearn.metrics import mean_squared_error
from sklearn.linear_model import Ridge
from sklearn.preprocessing import StandardScaler
import time

# Function to load the dataset from a file
def load_data(file_path):
    df = pd.read_csv(file_path, header=None, sep=r'\s+')
    df.columns = ['rate1', 'rate2'] # Assign column names
    return df

# Function to create features and targets from the dataset
def prepare_features_and_targets(df, lag):
    X, y = [], []
    for i in range(lag, len(df)):
        X.append(df.iloc[i - lag:i, 0].values) # Collect past values as features
        y.append(df.iloc[i, 1]) # Collect current value as target
    return np.array(X), np.array(y)

# Function to train and evaluate the Ridge Regression model
def train_and_evaluate_ridge(X_train, y_train, X_test, y_test):
    scaler = StandardScaler()
    X_train_scaled = scaler.fit_transform(X_train)
    X_test_scaled = scaler.transform(X_test)

    model = Ridge()

    # Measure training time
    start_time = time.time()
    model.fit(X_train_scaled, y_train)
    training_time = time.time() - start_time

    # Measure prediction time
```

```
    start_time = time.time()
    y_train_pred = model.predict(X_train_scaled)
    y_test_pred = model.predict(X_test_scaled)
    prediction_time = time.time() - start_time

    # Calculate RMSE for training and testing data
    train_rmse = np.sqrt(mean_squared_error(y_train, y_train_pred))
    test_rmse = np.sqrt(mean_squared_error(y_test, y_test_pred))

    return train_rmse, test_rmse, y_train_pred, y_test_pred, training_time,
prediction_time

# Load the dataset
file_path = '/Users/mali/Desktop/Gas_dataset/gas_datasets.txt'
data = load_data(file_path)

# Split the dataset into training and testing sets
split = len(data) // 2
train_set = data.iloc[:split]
test_set = data.iloc[split:]

# Number of past values to consider for features
lag = 5

# Create training and testing features and targets
X_train, y_train = prepare_features_and_targets(train_set, lag)
X_test, y_test = prepare_features_and_targets(test_set, lag)

# Train and evaluate the Ridge Regression model
train_rmse, test_rmse, y_train_pred, y_test_pred, training_time, prediction_time =
train_and_evaluate_ridge(X_train, y_train, X_test, y_test)

# Print the results
print(f"Ridge Regression: Train RMSE = {train_rmse}, Test RMSE = {test_rmse}, Training
Time = {training_time}s, Prediction Time = {prediction_time}s")

# Plot predictions vs actual values
plt.figure(figsize=(14, 7))
plt.plot(train_set.index[lag:], y_train, 'o-', label='Actual (Train)', markersize=5,
color='blue')
plt.plot(test_set.index[lag:], y_test, 'd-.', label='Actual (Test)', markersize=5,
color='red')
plt.plot(train_set.index[lag:], y_train_pred, label='Predicted (Train)', linestyle='--',
color='green')
plt.plot(test_set.index[lag:], y_test_pred, label='Predicted (Test)', linestyle='--',
color='purple')

plt.xlabel('Time', fontsize=12)
plt.ylabel('Rate', fontsize=12)
plt.title('Predictions vs Actual Values (Ridge Regression)', fontsize=15)
plt.legend()
plt.grid(True, linestyle='--', linewidth=0.5)
plt.show()
```

5.2 Results

The performance metrics for the Ridge Regression model are as follows:

- **Train RMSE**: 0.5322
- **Test RMSE**: 1.3187
- **Training Time**: 0.00029 seconds
- **Prediction Time**: 0.00003 seconds

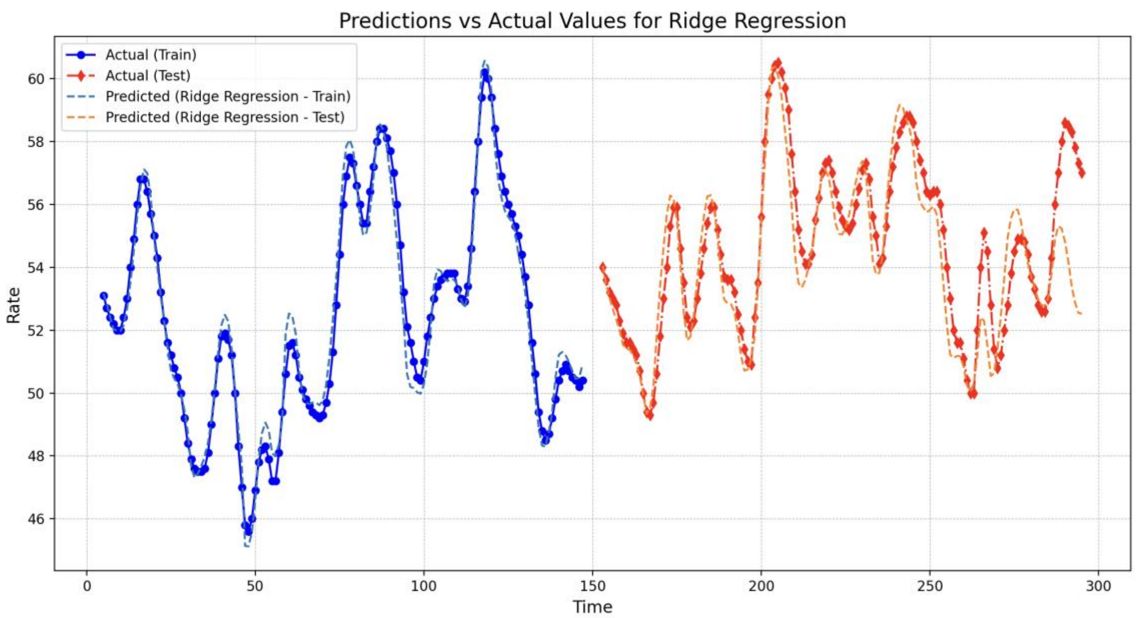
These results demonstrate that the Ridge Regression model provides a good balance betweenaccuracy and computational efficiency.

6. Visualization

The plot below shows the actual vs predicted values for both the training and test sets using the RidgeRegression model. This visualization helps in understanding the model's performance in predicting CO2 levels.

```python
# Plot predictions vs actual values
plt.figure(figsize=(14, 7))
plt.plot(train_set.index[lag:], y_train, 'o-', label='Actual (Train)', markersize=5,
color='blue')
plt.plot(test_set.index[lag:], y_test, 'd-.', label='Actual (Test)', markersize=5,
color='red')
plt.plot(train_set.index[lag:], y_train_pred, label='Predicted (Train)', linestyle='--',
color='green')
plt.plot(test_set.index[lag:], y_test_pred, label='Predicted (Test)', linestyle='--',
color='purple')

plt.xlabel('Time', fontsize=12)
plt.ylabel('Rate', fontsize=12)
plt.title('Predictions vs Actual Values (Ridge Regression)', fontsize=15)
plt.legend()
plt.grid(True, linestyle='--', linewidth=0.5)
plt.show()
```



7.Conclusion

The Ridge Regression model was chosen for its balanced performance in terms of accuracy and computational efficiency. The detailed evaluation showed that it performs well on both training and testing datasets, with low training and prediction times. This makes Ridge Regression a suitable choicefor this regression task, providing reliable predictions with efficient computation.

Future Work

For future work, further optimization of hyperparameters could be explored to potentially improve themodel's performance. Additionally, incorporating more features or using advanced ensemble methods could be considered to enhance predictive accuracy.