Using Regression Analysis for House Price Prediction

Abstract

Non-parametric regression methods are highly effective in addressing complex nonlinear relationships, while parametric regression methods are widely applied in predictive modeling for their simplicity and interpretability. This project aims to analyze the performance differences among three non-parametric regression methods (Nadaraya-Watson regression, k-Nearest Neighbors, and Neural Networks) and one parametric regression method (Ridge regression) in the context of house price prediction. Using the Boston Housing dataset, the study evaluates the predictive accuracy and computational efficiency of these methods under different conditions. The results indicate that Neural Networks exhibit significant advantages in modeling complex data relationships, whereas Ridge regression performs relatively poorly in low-dimensional data settings. This research highlights the importance of model selection in enhancing regression performance and provides valuable insights for data-driven decision-making.

1 Introduction

Regression analysis is a common method for predicting relationships between variables. This project compares non-parametric methods, such as Nadaraya-Watson regression, k-Nearest Neighbors (k-NN), and Neural Networks, with a parametric method, Ridge regression, for house price prediction.

The Boston Housing dataset[1] is used, which includes 506 samples and 13 features, such as crime rates, distance to highways, and median house prices. The models are evaluated using Mean Squared Error (MSE) for accuracy and computation time for efficiency. Feature selection and dimension reduction are also applied to test model performance.

Results show that Neural Networks perform the best in capturing complex data relationships, while Ridge regression has weaker results in low-dimensional settings. This study highlights the importance of choosing the right model based on data characteristics.

2 Dataset Description

The Boston Housing dataset originates from housing data collected in the Boston area of Massachusetts by the U.S. Census Bureau. It comprises 506 observations, each representing statistical information about a residential area along with its corresponding median housing price. The dataset includes 13 predictive features, such as the crime rate, distance to highways, and etc. Detailed descriptions of these features are provided in Table 1. Some features in the dataset, including CRIM, ZN, and INDUS, have 20 missing values, while other features, such as the target variable MEDV, are complete. Missing values will be imputed using the median. Additionally, the correlation coefficients between columns will be calculated, and the results are shown in Figure 1.

Table 1: Description of the Boston Housing Dataset

Feature	Description
CRIM	Per capita crime rate by town
ZN	Proportion of residential land zoned for lots over 25,000 sq.ft.
INDUS	Proportion of non-retail business acres per town
CHAS	Charles River dummy variable (1 if tract bounds river; 0 otherwise)
NOX	Nitric oxides concentration (parts per 10 million)
RM	Average number of rooms per dwelling
AGE	Proportion of owner-occupied units built prior to 1940
DIS	Weighted distances to five Boston employment centres
RAD	Index of accessibility to radial highways
TAX	Full-value property-tax rate per $10,000$
PTRATIO	Pupil-teacher ratio by town
B	$1000(B_k - 0.63)^2$ where B_k is the proportion of blacks by town
LSTAT	Percentage of lower status of the population
MEDV	Median value of owner-occupied homes in $1000's

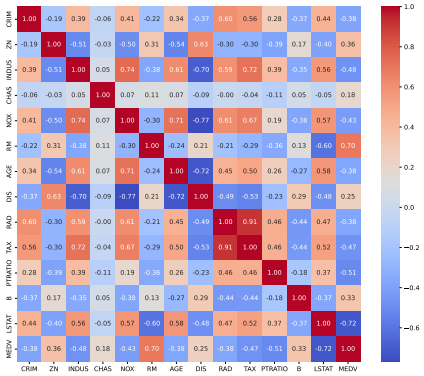

Figure 1: Correlation Heatmap

3 Methods

3.1 Nadaraya-Watson Estimator(Multivariate)

We consider the case of random X-variables and assume that the data points are observation from an i.i.d. sample $((X_1, Y_1), \cdots, (X_n, Y_n))$ from (X, Y). We supposed that the model is

$$\begin{aligned} m(x) &= E[Y \mid X = x] \\ &= \int y \frac{f_{(X,Y)}(x, y)}{f_X(x)} dy. \end{aligned}$$

We can be easily extended to $x \in \mathbb{R}^d$ and we can estimate $f_X(x)$ and $f_{(X,Y)}(x, y)$by

$$\hat{f}_X(x) = \frac{1}{n \det(H)} \sum_{i=1}^{n} K_x \left[H^{-1}(x - X_i)\right]$$

$$\hat{f}_{(X,Y)}(x, y) = \frac{1}{nh \det(H)} \sum_{i=1}^{n} K_y \left(\frac{y - Y_i}{h}\right) K_x \left[H^{-1}(x - X_i)\right].$$

The kernel regression estimation $\hat{m}(x)$ is called Nadaraya-Watson estimation and is given by

$$\hat{m}_H(x) = \int y \frac{\hat{f}_{(X,Y)}(x, y)}{\hat{f}_X(x)} dy = \frac{\sum_{i=1}^{n} K \left[H^{-1}(x - X_i)\right] Y_i}{\sum_{i=1}^{n} K \left[H^{-1}(x - X_i)\right]}.$$

3.2 k-Nearest Neighbors Estimator

The k-Nearest Neighbors (k-NN) regression approach relies on local averaging of the observed responses near the point of interest. Now we have an independent and identically distributed training dataset $\{(X_1, Y_1), (X_2, Y_2), \cdots, (X_n, Y_n)\}$. We define a distance metric is

$$\text{Distance} = \|x - x'\|$$

and identify the set

$$N_k(x) = \{i : X_i \text{ is among the } k \text{ closest points to } x\}$$

The k-NN estimator is then given by the average of the outputs corresponding to these nearest neighbors

$$\hat{m}(x) = \frac{1}{k} \sum_{i \in N_k(x)} Y_i.$$

3.3 Neural Networks

A simple mathematical model for a neuron can be expressed as

$$y = f\left(\sum_{i=1}^{n} w_i x_i + b\right),$$

where x_i represents the input features, w_i are the weights associated with these inputs, b is a bias term, and $f(\cdot)$ is the activation function.

A neural network usually consists of multiple layers of neurons. For the l-th layer, the output can be representated as

$$h^{(l)} = f^{(l)}(W^{(l)} h^{(l-1)} + b^{(l)})$$

where

- $h^{(l-1)}$ is the output of the previous layer.
- $W^{(l)}$ is the weight for layer l.
- $b^{(l)}$ is the bias for layer l.
- $f^{(l)}(\cdot)$ is the activation function.

We want to train this model, so we define a loss function

$$J(W, b) = \frac{1}{m} \sum_{j=1}^{m} L\left(h^{(L)}(x^{(j)}; W, b), y^{(j)}\right)$$

To minimize this loss, we compute the gradient of J with respect to the parameters: $\Delta_{W,b} J(W, b)$. Using the gradient descent, we iteratively update the parameters

$$W \leftarrow W - \eta \frac{\partial J}{\partial W}, \quad b \leftarrow b - \eta \frac{\partial J}{\partial b},$$

where η is the learning rate.

3.4 Ridge Regression

We want to approximate $m(x)$ by a linear function (in β) as

$$g(x) = \sum_{k=1}^{K} \beta_k g_k(x),$$

where the function forms of $\{g_k(x)\}$ are pre-determined. If we suppose that all variables have been centered, so we know the ridge regression minimizes

$$\begin{aligned} L &= \sum_{i=1}^{n} \left[Y_i - \sum_{k=1}^{K} \beta_k g_k(X_i) \right]^2 + \lambda \sum_{k=1}^{K} \beta_k^2 \\ &= (Y - G\beta)^T (Y - G\beta) + \lambda \beta^T \beta. \end{aligned}$$

So the ridge estimator is

$$\hat{\beta}^{ridge} = (G^T G + \lambda I)^{-1} G^T Y.$$

4 Experiments

4.1 Evaluation Metric

The model's performance was evaluated using two metrics: Mean Squared Error (MSE) for accuracy and computation time for efficiency.

Mean Squared Error (MSE) is an important metric for evaluating the difference between predicted and actual values. It is computed using the formula:

$$\text{MSE} = \frac{1}{n} \sum_{i=1}^{n} (\hat{y}_i - y_i)^2,$$

where $\hat{y}_i$ is the predicted value for the i-th sample, y_i is the true value for the i-th sample, and n is the total number of samples. A lower MSE indicates better alignment between predictions and true values, reflecting higher model accuracy.

Computation time is used to measure the model's efficiency. This metric records the time (in seconds) required for the model to process inputs and generate outputs, providing a clear indication of its execution speed and practical usability.

4.2 Experiments

4.2.1 Nadaraya-Watson Regression

The NW method cannot handle categorical or discrete data, as these types of variables do not have a probability density function. Therefore, columns such as CHAS, RM, and RAD were excluded from the dataset for this experiment. To select the most relevant features for the model, I used correlation coefficients to identify the top five features. Among them, INDUS, PTRATIO, and LSTAT are negatively correlated with the target variable, while ZN and B show positive correlations. Additionally, to ensure a comprehensive evaluation, I also trained the model using all available features.

4.2.2 k-Nearest Neighbors Regression

The k-Nearest Neighbors (k-NN) regression method is capable of handling any type of data. To evaluate its performance, I experimented with varying values of k and conducted comparisons across different feature subsets.

4.2.3 Neural Networks

The architecture consisted of 3 fully connected layers, each utilizing the ReLU activation function. The training process employed the Adam optimizer, with the mean squared error (MSE) as the loss function. The model was trained for 700 epochs, with the learning rate set at 0.01.

4.2.4 Ridge Regression

When training a Ridge regression model, the feature selection follows the same as employed for k-NN regression. The Ridge regression hyperparameter λ is set to two values: 1 and 3.

4.3 Results

The results of the experiment are shown in the Table 2.

Table 2: Results

Method	MSE	Time(s)
NW($d = 10$)	17.8930	0.9800
NW($d = 5$)	24.6781	0.8594
k-NN($d = 13, k = 3$)	19.1165	**0.0249**
k-NN($d = 13, k = 5$)	22.2142	0.0596
k-NN($d = 5, k = 3$)	17.3679	0.0547
k-NN($d = 5, k = 5$)	17.3370	0.0381
Ridge($d = 13, \lambda = 1$)	25.0027	0.0431
Ridge($d = 13, \lambda = 3$)	25.0108	0.0555
Ridge($d = 5, \lambda = 1$)	27.8886	0.0848
Ridge($d = 5, \lambda = 3$)	27.8377	0.0455
NN($d = 13$)	**14.3443**	0.4154
NN($d = 5$)	15.9074	0.4188

5 Discussion

For the Nadaraya-Watson (NW) method, when the feature dimension is reduced from 10 to 5, the model's mean squared error (MSE) significantly increases from 17.8930 to 24.6781. This indicates that feature dimensionality reduction in the NW model leads to the loss of critical information.

In contrast, the k-NN method shows greater adaptability to changes in feature dimensions and the hyperparameter k. In high-dimensional space ($d = 13$), the MSE for $k = 3$ and $k = 5$ is 19.1165 and 22.2142, respectively. After dimensionality reduction to $d = 5$, the MSE decreases to 17.3679 ($k = 3$) and 17.3370 ($k = 5$), suggesting that lower-dimensional spaces improve similarity measures. In addition, k-NN achieves good time efficiency, with runtime ranging from 0.0249s to 0.0596s.

For the Ridge regression method, dimensionality reduction has a clear negative impact on performance. At $d = 13$, the MSE remains around 25.0. However, when the dimensionality is reduced to $d = 5$, the MSE rises above 27.8, indicating that critical information not fully learned in the discarded dimensions weakens the model's predictive capability.

Neural networks (NN) achieve the lowest MSE among all models. At $d = 13$, the MSE is 14.3443, significantly outperforming other methods. Even after dimensionality reduction to $d = 5$, the MSE remains relatively low at 15.9074. This shows the better modeling ability of neural networks in preserving and extracting feature information. However, this performance comes at the cost of higher training time, which is higher than methods like k-NN.

6 Conclusion

This project examined four regression models under different settings, and the neural network model was found to perform best overall. When selecting an appropriate model, it is essential to consider the characteristics of the dataset. For instance, the Nadaraya-Watson (NW) method may not be suitable for discrete or categorical variables. Moreover, differences in data dimensionality can significantly affect model performance, emphasizing the importance of proper feature engineering. As computational power advances and the availability of high-dimensional datasets grows, neural networks increasingly represent a promising choice for regression analysis.

References

[1] David Harrison and Daniel L Rubinfeld. Hedonic prices and the demand for clean air. *Journal of Environmental Economics and Management*, 5:81–102, 1978.

A Appendix

```python
# loading dataset
import numpy as np
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler
from sklearn.neighbors import KNeighborsRegressor
from sklearn.metrics import mean_squared_error, r2_score
import warnings
import pandas as pd

#load data
housing = pd.read_csv('/Users/max/Computer-Intensive-Statistics-and-Applications-1MS049/Individual Project/HousingData.csv')
housing.head()
# fill data
data = pd.read_csv('/Users/max/Computer-Intensive-Statistics-and-Applications-1MS049/Individual Project/HousingData.csv')
data = data.fillna(data.median())
data.head()
#calculate correlation cofficients
import matplotlib.pyplot as plt
import seaborn as sns
correlation_matrix = data.corr()
correlation_with_medv = correlation_matrix['MEDV'].sort_values(ascending=False)
print(correlation_with_medv)
plt.figure(figsize=(12, 10))
sns.heatmap(correlation_matrix, annot=True, fmt='.2f', cmap='coolwarm')
plt.savefig('correlation_heatmap.pdf', format='pdf', bbox_inches='tight')
plt.show()
```

```python
#NW Estimator
import time
import numpy as np
import pandas as pd
from sklearn.model_selection import train_test_split
from sklearn.metrics import mean_squared_error
import matplotlib.pyplot as plt
from sklearn.preprocessing import StandardScaler

start_time = time.time()

X_NW = data.drop(["MEDV","CHAS","RM","RAD"], axis=1).values
X_special = data[['ZN','B','INDUS','PTRATIO','LSTAT']].values
y_NW = data["MEDV"].values
X_train, X_test, y_train, y_test = train_test_split(X_special, y_NW, test_size=0.2, random_state=42)

scaler = StandardScaler()
X_train = scaler.fit_transform(X_train)
X_test = scaler.transform(X_test)
def gaussian_kernel(x, x_i, bandwidth):
    return np.exp(-0.5 * np.sum(((x - x_i) / bandwidth) ** 2)) / ((2 * np.pi) ** (x.shape[0] / 2) * np.prod(bandwidth))
def nadaraya_watson(X_train, y_train, x_query, bandwidth):
    weights = np.array([gaussian_kernel(x_query, x_i, bandwidth) for x_i in X_train])
    return np.sum(weights * y_train) / np.sum(weights)
bandwidth = np.std(X_train, axis=0) * 0.5
train_predictions = [nadaraya_watson(X_train, y_train, x, bandwidth) for x in X_train]
```

```
test_predictions = [nadaraya_watson(X_train, y_train, x, bandwidth)
    for x in X_test]
train_mse = mean_squared_error(y_train, train_predictions)
test_mse = mean_squared_error(y_test, test_predictions)
print("Train Mean Squared Error (MSE):", train_mse)
print("Test Mean Squared Error (MSE):", test_mse)
end_time = time.time()
print("Time taken:", end_time - start_time)
```

```
# k-Nearest Neighbors Regression
import time
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler
from sklearn.neighbors import KNeighborsRegressor
from sklearn.metrics import mean_squared_error, r2_score
import matplotlib.pyplot as plt
start_time = time.time()

X = data.drop("MEDV", axis=1).values
y = data["MEDV"].values
X_special = data[["RM","LSTAT","PTRATIO","INDUS","TAX"]].values
X= X_special

X_train, X_test, y_train, y_test = train_test_split(X, y, test_size
    =0.2, random_state=42)


scaler = StandardScaler()
X_train = scaler.fit_transform(X_train)
X_test = scaler.transform(X_test)


knn = KNeighborsRegressor(n_neighbors=3)  # k = 5
knn.fit(X_train, y_train)


y_train_pred = knn.predict(X_train)
y_test_pred = knn.predict(X_test)


train_mse = mean_squared_error(y_train, y_train_pred)
test_mse = mean_squared_error(y_test, y_test_pred)

print(f"Train Mean Squared Error: {train_mse}")
print(f"Test Mean Squared Error: {test_mse}")
end_time = time.time()
print("Time taken:", end_time - start_time)
```

```
# Ridge Regression
import numpy as np
import pandas as pd
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler
from sklearn.linear_model import Ridge
from sklearn.metrics import mean_squared_error
import matplotlib.pyplot as plt
import time
start_time = time.time()

X = data.drop("MEDV", axis=1).values
y = data["MEDV"].values
X_special = data[["RM","LSTAT","PTRATIO","INDUS","TAX"]].values
X = X_special

```

```
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size
    =0.2, random_state=42)


scaler = StandardScaler()
X_train_scaled = scaler.fit_transform(X_train)
X_test_scaled = scaler.transform(X_test)


alpha = 1.0
ridge_model = Ridge(alpha=alpha)
ridge_model.fit(X_train_scaled, y_train)


y_pred = ridge_model.predict(X_test_scaled)


mse = mean_squared_error(y_test, y_pred)
print("Mean␣Squared␣Error␣(MSE):", mse)
end_time = time.time()
print("Time␣taken:", end_time - start_time)
```

```
# Neural Networks Regression
import torch
import torch.nn as nn
import torch.optim as optim
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler
import matplotlib.pyplot as plt
import time
start_time = time.time()

device = torch.device("cuda" if torch.cuda.is_available() else "cpu")

X= data.drop("MEDV", axis=1).values
y = data["MEDV"].values
X_special = data[["RM","LSTAT","PTRATIO","INDUS","TAX"]].values
X= X_special

scaler = StandardScaler()
X = scaler.fit_transform(X)


X = torch.tensor(X, dtype=torch.float32).to(device)
y = torch.tensor(y, dtype=torch.float32).view(-1, 1).to(device)


X_train, X_test, y_train, y_test = train_test_split(X, y, test_size
    =0.2, random_state=42)


class BostonHousingModel(nn.Module):
    def __init__(self):
        super(BostonHousingModel, self).__init__()
        self.fc1 = nn.Linear(X.shape[1], 64)
        self.fc2 = nn.Linear(64, 32)
        self.fc3 = nn.Linear(32, 1)
        self.relu = nn.ReLU()

    def forward(self, x):
        x = self.relu(self.fc1(x))
        x = self.relu(self.fc2(x))
        x = self.fc3(x)
        return x
```

```
42
43 model = BostonHousingModel().to(device)
44
45
46 criterion = nn.MSELoss()
47 optimizer = optim.Adam(model.parameters(), lr=0.01)
48
49
50 num_epochs = 700
51 train_losses = []
52 test_losses = []
53
54 for epoch in range(num_epochs):
55
56     model.train()
57     predictions = model(X_train)
58     loss = criterion(predictions, y_train)
59
60
61     optimizer.zero_grad()
62     loss.backward()
63     optimizer.step()
64
65
66     train_losses.append(loss.item())
67     model.eval()
68     with torch.no_grad():
69         test_predictions = model(X_test)
70         test_loss = criterion(test_predictions, y_test).item()
71         test_losses.append(test_loss)
72
73
74     if (epoch + 1) % 10 == 0:
75         print(f"Epoch [{epoch+1}/{num_epochs}], Train Loss: {loss.
              item():.4f}, Test Loss: {test_loss:.4f}")
76
77
78 model.eval()
79 with torch.no_grad():
80     y_pred = model(X_test)
81 mse = mean_squared_error(y_test, y_pred)
82 print(f"Mean Squared Error (MSE): {mse:.4f}")
83 end_time = time.time()
84 print("Time taken:", end_time - start_time)
```