CS 577 Project

April 21, 2024

1 Olympics Project

1.0.1 Introduction

In this project, we are attempting to count the number of time the US has won a golden medal in the Olympics using a regression model. The data set being used is derived from the dataset_olympics.csv found on the Kaggle.com. However, since our goal has narrowed down our data to be focused on the U.S., we used MySQL and filtered the data to only include information in regards to the U.S. We also feature-engineered a new column called Gold_count which includes a numeric representation of the golden medal count. Our final dataset used is called Olympics_dataset.csv

1.0.2 Reading the Data Set

[17]:
```
#we will start by importing necessary libraries to implement our goal
import pandas as pd
import numpy as np
```

[31]:
```
df = pd.read_csv("Olympics_dataset.csv")
```

[32]:
```
# a quick representation of the data set
df. head()
```

[32]:
```
      ID                          Name Sex  Age  Height  Weight  \
0  22700  James Brendan Bennet Connolly   M   27     175    72.0
1  22700  James Brendan Bennet Connolly   M   27     175    72.0
2  22700  James Brendan Bennet Connolly   M   27     175    72.0
3  16616      Thomas Edmund Tom" Burke"   M   21     183    66.0
4  16616      Thomas Edmund Tom" Burke"   M   21     183    66.0

            Team  NOC        Games  Year  Season    City      Sport  \
0  United States  USA  1896 Summer  1896  Summer  Athina  Athletics
1  United States  USA  1896 Summer  1896  Summer  Athina  Athletics
2  United States  USA  1896 Summer  1896  Summer  Athina  Athletics
3  United States  USA  1896 Summer  1896  Summer  Athina  Athletics
4  United States  USA  1896 Summer  1896  Summer  Athina  Athletics

                         Event   Medal  Gold_Count
```

```
0    Athletics Men's High Jump  Silver          0
1    Athletics Men's Long Jump  Bronze          0
2  Athletics Men's Triple Jump    Gold          1
3   Athletics Men's 100 metres    Gold          1
4   Athletics Men's 400 metres    Gold          1
```

[33]:
```
df.tail()
```

[33]:
```
        ID                                     Name Sex  Age  Height  Weight  \
3851  8093                            Danny Barrett   M   26     188   102.0
3852  8128  Jennifer Mae Jenny" Barringer-Simpson"   F   29     166    53.0
3853  8173                       Thomas Barrows III   M   28     186    82.0
3854  6317         Anthony Lawrence Tony" Azevedo"   M   34     186    90.0
3855  6911                             Tavis Bailey   M   24     191   125.0

               Team  NOC        Games  Year  Season            City  \
3851  United States  USA  2016 Summer  2016  Summer  Rio de Janeiro
3852  United States  USA  2016 Summer  2016  Summer  Rio de Janeiro
3853  United States  USA  2016 Summer  2016  Summer  Rio de Janeiro
3854  United States  USA  2016 Summer  2016  Summer  Rio de Janeiro
3855  United States  USA  2016 Summer  2016  Summer  Rio de Janeiro

              Sport                           Event   Medal  Gold_Count
3851  Rugby Sevens  Rugby Sevens Men's Rugby Sevens     NaN           0
3852     Athletics   Athletics Women's 1,500 metres  Bronze           0
3853       Sailing               Sailing Men's Skiff     NaN           0
3854    Water Polo       Water Polo Men's Water Polo     NaN           0
3855     Athletics     Athletics Men's Discus Throw     NaN           0
```

-> In total there are 3857 records with 15 features (X) and one target variable (y)

1.0.3 Splitting The Data

[35]:
```
from sklearn.model_selection import train_test_split
```

[36]:
```
#Specifying the features/independent variable (X) and target/dependet variable␣
↪(y)

X = df[['ID','Name', 'Sex','Age','Height','Weight', 'Team', 'NOC', 'Games',␣
↪'Year','Season', 'City', 'Sport', 'Event','Medal']]
y = df['Gold_Count']

#Splitting X and y into training and temporary tuning sets using train-test␣
↪split
#Stratifying to ensure y is preserved
```

```
X_train, X_tun, y_train, y_tun = train_test_split(X, y, test_size=0.3,␣
↪random_state=42, stratify=y)

#Splitting the temporary tuning set into a validation set and a test set
X_valid, X_test, y_valid, y_test = train_test_split(X_tun, y_tun, test_size=0.
↪5, random_state=42)
```

Since our data set has a mix of both numerical and categorical variables, an issue will occur since our goal is to solve a regression question. So, we have to represent the categorical variables numerically using proper encoding. Therefore, we have to preprocess the data.

[37]:
```
from sklearn.compose import ColumnTransformer
from sklearn.pipeline import Pipeline
from sklearn.preprocessing import StandardScaler, OneHotEncoder
```

[38]:
```
#First, we define the relevant numerical features and regularize them
numeric_features = ['Age', 'Height', 'Weight']
transform_num = StandardScaler()

#Second, we define the relevant categorical features and apply one hot encoding

categorical_features = ['Sex', 'Season', 'City', 'Sport', 'Event']
transform_cat = OneHotEncoder(handle_unknown='ignore')
```

Now, we combine both steps to make sure our data that we are using is appropraitely combined

[39]:
```
combined_data = ColumnTransformer(transformers=[('num', transform_num,␣
↪numeric_features),('cat', transform_cat, categorical_features)])
```

1.0.4 Establishing a Baseline

Since we are solving for a regression problem, the best baseline to be established is a mean and median baseline.

[41]:
```
from sklearn.metrics import r2_score, mean_squared_error
```

[92]:
```
# finding the Mean and Median of the target variable
y_median = y_train.median()
y_mean = y_train.mean()

#Predicting mean and median
y_pred_median = [y_median] * len(y_test)
y_pred_mean = [y_mean] * len(y_test)

#Evalutaing the Median Baseline
median_mse = mean_squared_error(y_test, y_pred_median)
median_rmse = np.sqrt(median_mse)
median_r2 = r2_score(y_test, y_pred_median)
```

```
#Evaluating the Mean Baseline
mean_mse = mean_squared_error(y_test, y_pred_mean)
mean_rmse = np.sqrt(mean_mse)
mean_r2 = r2_score(y_test, y_pred_mean)

# Displaying the Results
print("Performance of Median Baseline")
print(f"MSE: {median_mse:.4f}")
print(f"RMSE: {median_rmse:.4f}")
print(f"R² score: {median_r2:.4f}\n")

print("Performance of Mean Baseline")
print(f"MSE: {mean_mse:.4f}")
print(f"RMSE: {mean_rmse:.4f}")
print(f"R² score: {mean_r2:.4f}")
```

```
Performance of Median Baseline
MSE: 0.1520
RMSE: 0.3899
R² score: -0.1792

Performance of Mean Baseline
MSE: 0.1289
RMSE: 0.3590
R² score: -0.0000
```

1.0.5 Model Implementation

Our Main goal is to utilize the Linear Regression model to predict the output of how many gold medals the U.S. won in the Olympics over the years. However, further research indicated that there could be more powerful models that might be more successful when predicting the same output. Therefore, we will test out other models in the process and compare the outcome to that of linear regression. Our performance metrics will evaluated by the the measurements of MSE (Mean Square Error), RMSE (Root Mean Square Error), and R^2 (R-squared)

Linear Regression

[43]:
```
from sklearn.pipeline import Pipeline
from sklearn.linear_model import LinearRegression
```

[44]:
```
#Defining the model using Pipelining to preserve the data transformation

LR_model = Pipeline(steps=[('combined_data', combined_data),('Model',␣
↪LinearRegression())])

#Training the model
LR_model.fit(X_train, y_train)
```

[44]:
```
Pipeline(steps=[('combined_data',
                 ColumnTransformer(transformers=[('num', StandardScaler(),
                                                  ['Age', 'Height', 'Weight']),
                                                 ('cat',
OneHotEncoder(handle_unknown='ignore'),
                                                  ['Sex', 'Season', 'City',
                                                   'Sport', 'Event'])])),
                ('Model', LinearRegression())])
```

[45]:
```
#Evaluating the Linear Regression Model
y_pred = LR_model.predict(X_test)

#Finding MSE
val_MSE = mean_squared_error(y_test, y_pred)

#Finding RMSE
r_MSE = np.sqrt(val_MSE)

#Finding R^2 on both training and testing sets
r2_train = LR_model.score(X_train, y_train)
r2 = r2_score(y_test, y_pred)

#Displaying the results
print("Performance of Linear Regression Model")
print(f"MSE: {val_MSE:.4f}")
print(f"RMSE: {r_MSE:.4f}" )
print(f"Testing Data R² score: {r2:.4f}" )
```

```
Performance of Linear Regression Model
MSE: 0.0929
RMSE: 0.3047
Testing Data R² score: 0.2795
```

KNN

[46]:
```
from sklearn.model_selection import GridSearchCV
from sklearn.neighbors import KNeighborsRegressor
```

[47]:
```
# Defining a range of k values
k = range(1,20)

# Using Cross Validation and GridSearch to find the best value for k
#Defining the parameter grid
param_grid = {'Model__n_neighbors': k}

# Creating a KNN model that is based on Regression
model = KNeighborsRegressor()

knn_model = Pipeline(steps=[('combined_data', combined_data), ('Model', model)])
```

```python
# Using GridSearch and cv of 5
grid_search = GridSearchCV(knn_model, param_grid, cv=5)

grid_search.fit(X_train, y_train)
```

[47]:
```
GridSearchCV(cv=5,
             estimator=Pipeline(steps=[('combined_data',
                                        ColumnTransformer(transformers=[('num',
StandardScaler(),
                                                                         ['Age',
'Height',
'Weight']),
                                                                        ('cat',
OneHotEncoder(handle_unknown='ignore'),
                                                                         ['Sex',
'Season',
'City',
'Sport',
'Event'])])),
                                       ('Model', KNeighborsRegressor())]),
             param_grid={'Model__n_neighbors': range(1, 20)})
```

[48]:
```python
#Evaluating the Knn model

#Finding MSE
val_MSE = grid_search.best_score_

#Finding RMSE
r_MSE = np.sqrt(val_MSE)

#Finding R^2 on both training and testing sets
r2 = r2_score(y_test, y_pred)

#Displaying the results
print("Performance of KNN Model")
print(f"MSE: {val_MSE:.4f}")
print(f"RMSE: {r_MSE:.4f}" )
print(f"Testing Data R² score: {r2:.4f}" )
```

```
Performance of KNN Model
MSE: 0.2787
RMSE: 0.5279
Testing Data R² score: 0.2795
```

Decision Trees

[18]:
```python
from sklearn.tree import DecisionTreeRegressor
```

[49]:
```
DT_model = Pipeline(steps=[('combined_data', combined_data),('Model',␣
↪DecisionTreeRegressor())])
```

[50]:
```
#Defining grid parameters with tree depth, number of examples for splitting,␣
↪and number of examples for leaf nodes
param_grid = {
    'Model__max_depth': [3, 5, 7, None],
    'Model__min_samples_split': [2, 5, 10],
    'Model__min_samples_leaf': [1, 2, 4]
}

# Using GridSearch and cv of 5
grid_search = GridSearchCV(DT_model, param_grid, cv=5)

#Training the Model
grid_search.fit(X_train, y_train)
```

[50]:
```
GridSearchCV(cv=5,
             estimator=Pipeline(steps=[('combined_data',
                                        ColumnTransformer(transformers=[('num',
StandardScaler(),
                                                                         ['Age',
'Height',
'Weight']),
                                                                        ('cat',
OneHotEncoder(handle_unknown='ignore'),
                                                                         ['Sex',
'Season',
'City',
'Sport',
'Event'])])),
                                       ('Model', DecisionTreeRegressor())]),
             param_grid={'Model__max_depth': [3, 5, 7, None],
                         'Model__min_samples_leaf': [1, 2, 4],
                         'Model__min_samples_split': [2, 5, 10]})
```

[51]:
```
#Evaluating the Decision Tree Model

#Finding MSE
val_MSE = grid_search.best_score_

#Finding RMSE
r_MSE = np.sqrt(val_MSE)

#Finding R^2 on both training and testing sets
r2 = r2_score(y_test, y_pred)
```

```
#Displaying the results
print("Performance of Decision Tree Model")
print(f"MSE: {val_MSE:.4f}")
print(f"RMSE: {r_MSE:.4f}" )
print(f"Testing Data R² score: {r2:.4f}" )
```

```
Performance of Decision Tree Model
MSE: 0.2969
RMSE: 0.5449
Testing Data R² score: 0.2795
```

Random Forest

[52]:
```
from sklearn.ensemble import RandomForestRegressor
```

[53]:
```
RF_model = Pipeline(steps=[('combined_data', combined_data),('Model',␣
↪RandomForestRegressor())])

# Training the model
RF_model.fit(X_train, y_train)
```

[53]:
```
Pipeline(steps=[('combined_data',
                 ColumnTransformer(transformers=[('num', StandardScaler(),
                                                  ['Age', 'Height', 'Weight']),
                                                 ('cat',
OneHotEncoder(handle_unknown='ignore'),
                                                  ['Sex', 'Season', 'City',
                                                   'Sport', 'Event'])])),
                ('Model', RandomForestRegressor())])
```

[54]:
```
#Evaluating the Random Forest Model
y_pred = RF_model.predict(X_test)

#Finding MSE
val_MSE = mean_squared_error(y_test, y_pred)

#Finding RMSE
r_MSE = np.sqrt(val_MSE)

#Finding R^2 on both training and testing sets
r2_train = RF_model.score(X_train, y_train)
r2 = r2_score(y_test, y_pred)

#Displaying the results
print("Performance of Random Forest Model")
print(f"MSE: {val_MSE:.4f}")
print(f"RMSE: {r_MSE:.4f}" )
print(f"Testing Data R² score: {r2:.4f}" )
```

```
Performance of Random Forest Model
MSE: 0.0749
RMSE: 0.2737
Testing Data R² score: 0.4187
```

Artificial Neural Networks

[31]:
```python
from keras.models import Sequential
from keras.layers import Dense
from keras.layers import Input
```

[ ]:
```python
X_train = combined_data.fit_transform(X_train)

# Transforming the data accordingly
X_valid= combined_data.transform(X_valid)
X_test = combined_data.transform(X_test)
shape = X_train.shape[1]
```

[61]:
```python
# Defining the Neural model
Neural_model = Sequential()
Neural_model.add(Dense(64, activation='relu', input_shape=(shape,)))
Neural_model.add(Dense(32, activation='relu'))
Neural_model.add(Dense(1))

# Compiling the Model
Neural_model.compile(optimizer='adam', loss='mean_squared_error')

# Training the Neural model
Neural_model.fit(X_train, y_train,validation_data=(X_valid,␣
 ↪y_valid),epochs=100, batch_size=32)

# Evaluating the Neural Model
test = Neural_model.evaluate(X_test, y_test)
print('Test :', test)

y_pred = Neural_model.predict(X_test)

# Finding the MSE
val_MSE = mean_squared_error(y_test, y_pred)

# Finding RMSE
r_MSE = np.sqrt(val_MSE)

# Finding R^2
r2 = r2_score(y_test, y_pred)

print("Performance of Neural Network Model")
print(f"MSE: {val_MSE:.4f}")
```

```python
print(f"RMSE: {r_MSE:.4f}" )
print(f"Testing Data R² score: {r2:.4f}" )
```

```
Epoch 1/100
85/85                2s 5ms/step - loss:
0.1202 - val_loss: 0.0926
Epoch 2/100
85/85                0s 2ms/step - loss:
0.0854 - val_loss: 0.0820
Epoch 3/100
85/85                0s 2ms/step - loss:
0.0694 - val_loss: 0.0808
Epoch 4/100
85/85                0s 2ms/step - loss:
0.0517 - val_loss: 0.0781
Epoch 5/100
85/85                0s 2ms/step - loss:
0.0501 - val_loss: 0.0795
Epoch 6/100
85/85                0s 2ms/step - loss:
0.0421 - val_loss: 0.0744
Epoch 7/100
85/85                0s 2ms/step - loss:
0.0371 - val_loss: 0.0754
Epoch 8/100
85/85                0s 2ms/step - loss:
0.0324 - val_loss: 0.0778
Epoch 9/100
85/85                0s 2ms/step - loss:
0.0304 - val_loss: 0.0774
Epoch 10/100
85/85                0s 2ms/step - loss:
0.0255 - val_loss: 0.0771
Epoch 11/100
85/85                0s 3ms/step - loss:
0.0201 - val_loss: 0.0791
Epoch 12/100
85/85                0s 2ms/step - loss:
0.0165 - val_loss: 0.0814
Epoch 13/100
85/85                0s 2ms/step - loss:
0.0149 - val_loss: 0.0832
Epoch 14/100
85/85                0s 2ms/step - loss:
0.0148 - val_loss: 0.0811
Epoch 15/100
85/85                0s 2ms/step - loss:
0.0124 - val_loss: 0.0788
```

```
Epoch 16/100
85/85            0s 2ms/step - loss:
0.0125 - val_loss: 0.0814
Epoch 17/100
85/85            0s 2ms/step - loss:
0.0108 - val_loss: 0.0823
Epoch 18/100
85/85            0s 2ms/step - loss:
0.0095 - val_loss: 0.0823
Epoch 19/100
85/85            0s 2ms/step - loss:
0.0092 - val_loss: 0.0792
Epoch 20/100
85/85            0s 2ms/step - loss:
0.0080 - val_loss: 0.0781
Epoch 21/100
85/85            0s 2ms/step - loss:
0.0074 - val_loss: 0.0799
Epoch 22/100
85/85            0s 2ms/step - loss:
0.0068 - val_loss: 0.0821
Epoch 23/100
85/85            0s 2ms/step - loss:
0.0080 - val_loss: 0.0787
Epoch 24/100
85/85            0s 3ms/step - loss:
0.0070 - val_loss: 0.0807
Epoch 25/100
85/85            0s 2ms/step - loss:
0.0067 - val_loss: 0.0794
Epoch 26/100
85/85            0s 2ms/step - loss:
0.0066 - val_loss: 0.0765
Epoch 27/100
85/85            0s 2ms/step - loss:
0.0050 - val_loss: 0.0825
Epoch 28/100
85/85            0s 2ms/step - loss:
0.0061 - val_loss: 0.0778
Epoch 29/100
85/85            0s 2ms/step - loss:
0.0052 - val_loss: 0.0817
Epoch 30/100
85/85            0s 2ms/step - loss:
0.0046 - val_loss: 0.0801
Epoch 31/100
85/85            0s 2ms/step - loss:
0.0059 - val_loss: 0.0789
```

```
Epoch 32/100
85/85             0s 2ms/step - loss:
0.0050 - val_loss: 0.0792
Epoch 33/100
85/85             0s 2ms/step - loss:
0.0054 - val_loss: 0.0818
Epoch 34/100
85/85             0s 2ms/step - loss:
0.0054 - val_loss: 0.0764
Epoch 35/100
85/85             0s 2ms/step - loss:
0.0034 - val_loss: 0.0806
Epoch 36/100
85/85             0s 2ms/step - loss:
0.0046 - val_loss: 0.0818
Epoch 37/100
85/85             0s 2ms/step - loss:
0.0041 - val_loss: 0.0773
Epoch 38/100
85/85             0s 3ms/step - loss:
0.0043 - val_loss: 0.0760
Epoch 39/100
85/85             0s 2ms/step - loss:
0.0042 - val_loss: 0.0819
Epoch 40/100
85/85             0s 2ms/step - loss:
0.0038 - val_loss: 0.0795
Epoch 41/100
85/85             0s 2ms/step - loss:
0.0038 - val_loss: 0.0787
Epoch 42/100
85/85             0s 2ms/step - loss:
0.0041 - val_loss: 0.0768
Epoch 43/100
85/85             0s 2ms/step - loss:
0.0037 - val_loss: 0.0792
Epoch 44/100
85/85             0s 2ms/step - loss:
0.0043 - val_loss: 0.0809
Epoch 45/100
85/85             0s 2ms/step - loss:
0.0044 - val_loss: 0.0787
Epoch 46/100
85/85             0s 2ms/step - loss:
0.0050 - val_loss: 0.0752
Epoch 47/100
85/85             0s 2ms/step - loss:
0.0035 - val_loss: 0.0773
```

```
Epoch 48/100
85/85              0s 2ms/step - loss:
0.0044 - val_loss: 0.0798
Epoch 49/100
85/85              0s 2ms/step - loss:
0.0033 - val_loss: 0.0756
Epoch 50/100
85/85              0s 2ms/step - loss:
0.0043 - val_loss: 0.0780
Epoch 51/100
85/85              0s 2ms/step - loss:
0.0032 - val_loss: 0.0767
Epoch 52/100
85/85              0s 2ms/step - loss:
0.0029 - val_loss: 0.0768
Epoch 53/100
85/85              0s 2ms/step - loss:
0.0033 - val_loss: 0.0772
Epoch 54/100
85/85              0s 3ms/step - loss:
0.0039 - val_loss: 0.0762
Epoch 55/100
85/85              0s 2ms/step - loss:
0.0047 - val_loss: 0.0768
Epoch 56/100
85/85              0s 2ms/step - loss:
0.0035 - val_loss: 0.0757
Epoch 57/100
85/85              0s 2ms/step - loss:
0.0028 - val_loss: 0.0759
Epoch 58/100
85/85              0s 2ms/step - loss:
0.0030 - val_loss: 0.0782
Epoch 59/100
85/85              0s 2ms/step - loss:
0.0030 - val_loss: 0.0759
Epoch 60/100
85/85              0s 2ms/step - loss:
0.0037 - val_loss: 0.0814
Epoch 61/100
85/85              0s 2ms/step - loss:
0.0028 - val_loss: 0.0751
Epoch 62/100
85/85              0s 2ms/step - loss:
0.0036 - val_loss: 0.0739
Epoch 63/100
85/85              0s 2ms/step - loss:
0.0031 - val_loss: 0.0749
```

```
Epoch 64/100
85/85            0s 2ms/step - loss:
0.0038 - val_loss: 0.0773
Epoch 65/100
85/85            0s 2ms/step - loss:
0.0034 - val_loss: 0.0780
Epoch 66/100
85/85            0s 3ms/step - loss:
0.0035 - val_loss: 0.0768
Epoch 67/100
85/85            0s 2ms/step - loss:
0.0037 - val_loss: 0.0754
Epoch 68/100
85/85            0s 2ms/step - loss:
0.0031 - val_loss: 0.0766
Epoch 69/100
85/85            0s 2ms/step - loss:
0.0032 - val_loss: 0.0738
Epoch 70/100
85/85            0s 3ms/step - loss:
0.0032 - val_loss: 0.0741
Epoch 71/100
85/85            0s 2ms/step - loss:
0.0034 - val_loss: 0.0786
Epoch 72/100
85/85            0s 2ms/step - loss:
0.0032 - val_loss: 0.0737
Epoch 73/100
85/85            0s 2ms/step - loss:
0.0032 - val_loss: 0.0773
Epoch 74/100
85/85            0s 2ms/step - loss:
0.0022 - val_loss: 0.0714
Epoch 75/100
85/85            0s 3ms/step - loss:
0.0030 - val_loss: 0.0742
Epoch 76/100
85/85            0s 2ms/step - loss:
0.0021 - val_loss: 0.0773
Epoch 77/100
85/85            0s 2ms/step - loss:
0.0025 - val_loss: 0.0750
Epoch 78/100
85/85            0s 2ms/step - loss:
0.0025 - val_loss: 0.0754
Epoch 79/100
85/85            0s 2ms/step - loss:
0.0028 - val_loss: 0.0749
```

```
Epoch 80/100
85/85            0s 2ms/step - loss:
0.0030 - val_loss: 0.0760
Epoch 81/100
85/85            0s 2ms/step - loss:
0.0027 - val_loss: 0.0738
Epoch 82/100
85/85            0s 3ms/step - loss:
0.0026 - val_loss: 0.0768
Epoch 83/100
85/85            0s 2ms/step - loss:
0.0026 - val_loss: 0.0757
Epoch 84/100
85/85            0s 2ms/step - loss:
0.0033 - val_loss: 0.0747
Epoch 85/100
85/85            0s 3ms/step - loss:
0.0023 - val_loss: 0.0738
Epoch 86/100
85/85            0s 3ms/step - loss:
0.0030 - val_loss: 0.0755
Epoch 87/100
85/85            0s 3ms/step - loss:
0.0027 - val_loss: 0.0763
Epoch 88/100
85/85            0s 2ms/step - loss:
0.0028 - val_loss: 0.0754
Epoch 89/100
85/85            0s 3ms/step - loss:
0.0028 - val_loss: 0.0756
Epoch 90/100
85/85            0s 2ms/step - loss:
0.0020 - val_loss: 0.0726
Epoch 91/100
85/85            0s 2ms/step - loss:
0.0022 - val_loss: 0.0756
Epoch 92/100
85/85            0s 2ms/step - loss:
0.0023 - val_loss: 0.0734
Epoch 93/100
85/85            0s 2ms/step - loss:
0.0024 - val_loss: 0.0739
Epoch 94/100
85/85            0s 2ms/step - loss:
0.0025 - val_loss: 0.0758
Epoch 95/100
85/85            0s 2ms/step - loss:
0.0029 - val_loss: 0.0720
```

```
Epoch 96/100
85/85             0s 2ms/step - loss:
0.0024 - val_loss: 0.0768
Epoch 97/100
85/85             0s 2ms/step - loss:
0.0021 - val_loss: 0.0708
Epoch 98/100
85/85             0s 2ms/step - loss:
0.0029 - val_loss: 0.0749
Epoch 99/100
85/85             0s 2ms/step - loss:
0.0021 - val_loss: 0.0751
Epoch 100/100
85/85             0s 2ms/step - loss:
0.0017 - val_loss: 0.0754
19/19             0s 1ms/step - loss:
0.0891
Test : 0.085304394364357
19/19             0s 5ms/step
Performance of Neural Network Model
MSE: 0.0882
RMSE: 0.2970
Testing Data R² score: 0.3156
```

1.0.6 Summary

Model	MSE	RMSE	R2
Linear Regression	0.0929	0.3047	0.2795
KNN	0.2787	0.5279	0.2795
Decision Tree	0.2969	0.5449	0.2795
Random Forest	0.0749	0.2737	0.4187
Neural Network	0.0882	0.2970	0.3156

Conclusion In order to evaluate each model's performance, generally lower MSE and RMSE values are preferred with a higher R^2 value. Among the implemented models above, the Random Forest Model has outperformed the Linear regression model as well as all of the others because it has the lowest MSE and RMSE scores accompanied by the highest R^2 score. The MSE score of 0.0749 indicates that the model's performance predictions are averagely close to the actual values. The RMSE score of 0.2737 indicates that the predictions of the Random Forest model are generally off that score from the actual values. The R^2 score of 0.4187 indicates a variance percentage of 41.87%, meaning that the model could detect that much resulted in variance in the data attributed to the independent variables. In second place comes the Linear Regression model, which performed comparatively well. Then, the Neural networks mode