Syam Evani
MECE 6397: HW 1

The problem statement can be summarized according to the key changes needed for the provided sample script below. Please note, my full python code is provided at the end of this document and I have also created a github repo that has my hw1 files here

1. Model Replacement: The LinearRegression model is replaced with

- **DecisionTreeRegressor**
- **RandomForestRegressor**
- **SupportVectorRegression (SVR)**
- **K-Nearest Neighbors Regressor (KNN)**

I have created a simple dictionary and loop structure that fits our transformed x data with our y training data to evaluate all regressors. These are added to appropriate keys in our *models* dictionary which is used downstream to evaluate our error.

```
#--------------------------------------------------------------------
# Apply different regression approaches
#--------------------------------------------------------------------
# Placeholder for results and best transformed features
results = {"lr": [],        # Linear regression
           "dtr": [],       # Decision tree regression
           "rfr": [],       # Random forest regression
           "svr": [],       # Support vector regression
           "knn": []        # K-nearest neighbors regression
            }
models = {}
predictions = {}
best_transformed_features = None

for i in range(len(design)):
    # Apply transformations based on the design matrix
    X_train_transformed = np.column_stack([levels[design[i, j]](X_train[:, j]) for j in range(X.shape[1])])
    X_test_transformed = np.column_stack([levels[design[i, j]](X_test[:, j]) for j in range(X.shape[1])])

    # Train a linear regression model
    models["lr"] = LinearRegression().fit(X_train_transformed, y_train)
    models["dtr"] = DecisionTreeRegressor().fit(X_train_transformed, y_train)
    models["rfr"] = RandomForestRegressor().fit(X_train_transformed, y_train)
    models["svr"] = SVR().fit(X_train_transformed, y_train)                         # Default rbf kernel, 3 degr
    models["knn"]  = KNeighborsRegressor().fit(X_train_transformed, y_train)        # By default will use 5 neig

    # Predict and calculate MSE
    for regressor in models:
        predictions[regressor] = models[regressor].predict(X_test_transformed)
        mse = mean_squared_error(y_test, predictions[regressor])

        # Store the results
        results[regressor].append((design[i], mse, predictions[regressor]))

```

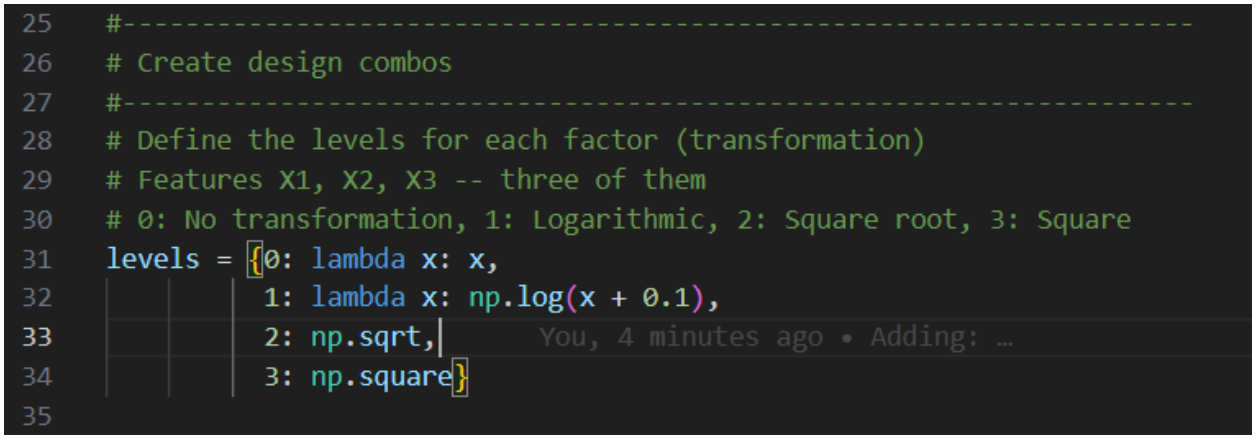
2. Three-Level Design: Each factor in the design matrix now has three levels, accommodating different transformations: no transformation, logarithmic, and square

I have updated our design to have 3 factors, each with 3 levels.

```
36  # Generating a full factorial design for 3 factors, each with 3 levels
37  design = fullfact([3, 3, 3])
```

3. Safety in Transformations: A small constant (0.1) is added to X where logarithmic transformations are used to avoid taking the logarithm of zero

As seen in line 32, I have added a small constant to X to ensure logarithmic transformations avoid taking log of zero



4. This setup allows you to explore the impact of different nonlinear transformations on the dataset using a model that can capture such nonlinearities effectively

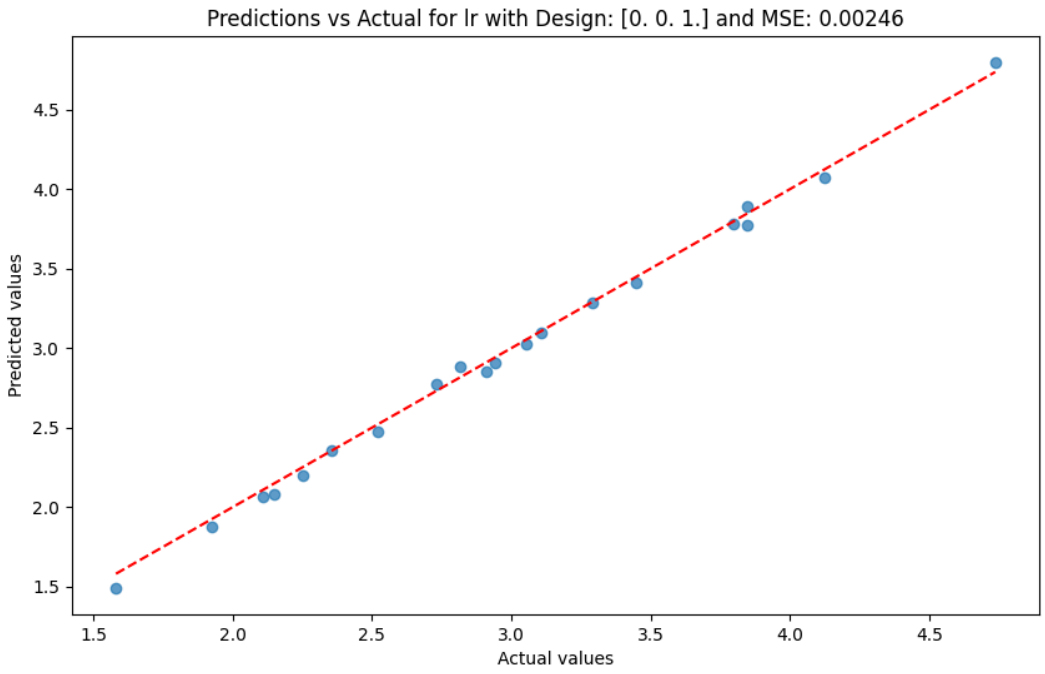
No commentary needed here, agreed!

5. Compare the results for MSE for the regressor models explored

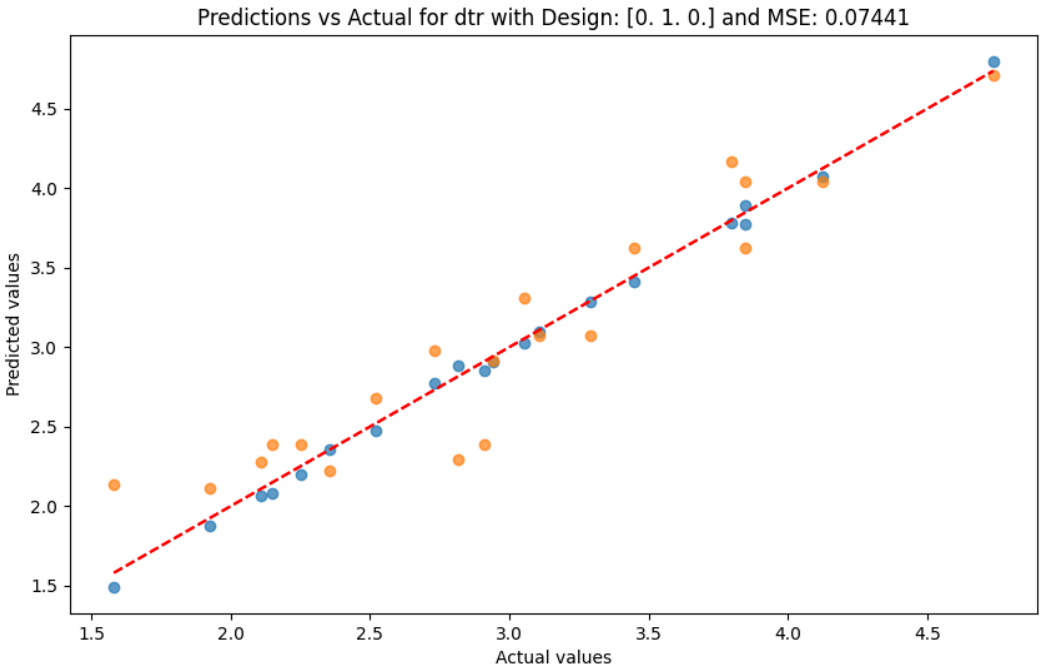
Regressor Type	Design with Lowest MSE	MSE
Linear	0 0 1	0.0024587061343516194
Decision Tree	0 1 1	0.07440568140632621
Random Forest	1 1 1	0.05500079319645723
Support Vector	0 1 2	0.01248389784244165
K-Nearest Neighbors	1 1 2	0.046979637634902816

I've also visualized these different regressors by plotting the expected values vs our predicted values (along with a red trendline to indicate what a perfect prediction would be). Those plots are provided below with some commentary that concludes this report.

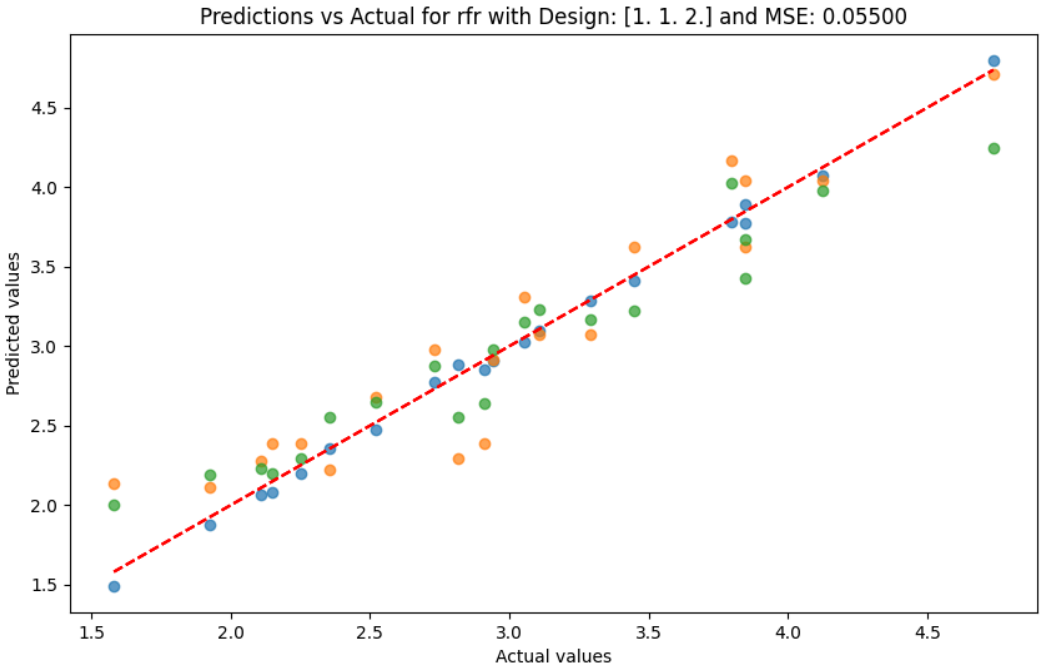
Linear



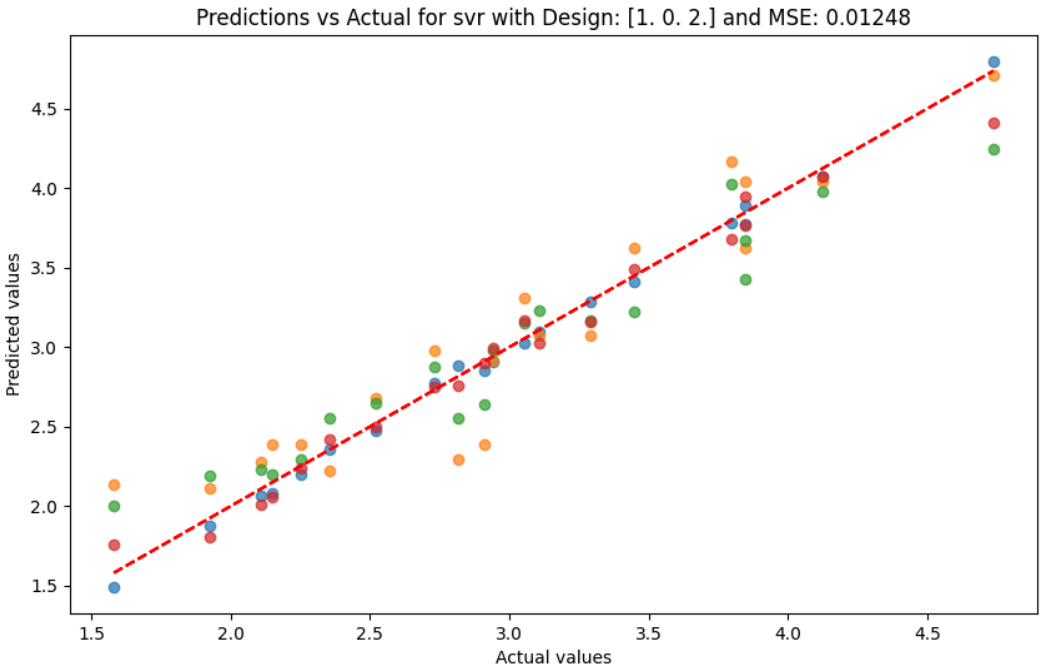
Decision Tree



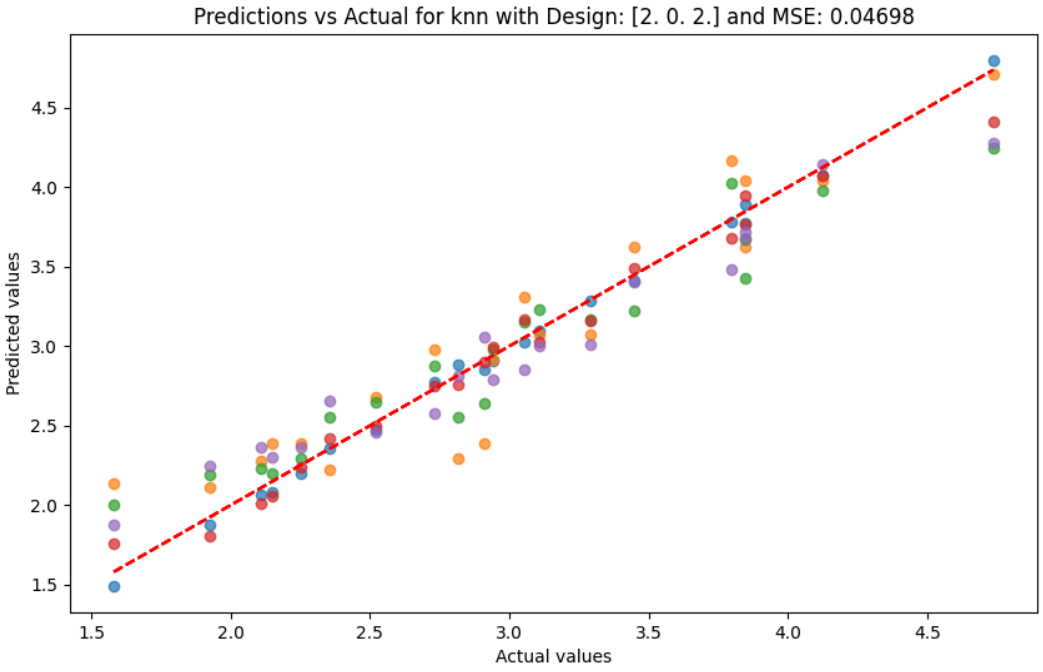
Random Forest



Support Vector



K-Nearest Neighbors



Looking at the results from both our table and plots, interestingly we see that for this relatively simple data linear regression provides the best predictions for our training data outputs. Following this, all models perform relatively similar but grade out in the following order (of least MSE aka best predictions):

1. Linear
2. Support Vector
3. K Nearest Neighbor
4. Random Forest
5. Decision Tree

A linear regression model as it might imply in the name, excels in linear relationships, has fast training and prediction, but can be prone to underfitting complex data. A decision tree regression model can capture nonlinear relationships and special interactions between features. However, it can be prone to overfitting in deep tree structures. Random forest regression constructs multiple decision trees and averages all trees to create predictions. This reduces overfitting, handles high dimension data well, but can be more computationally challenging and understandable than a single decision tree regression model. Support vector regression is a computationally advantageous model that is effective in high dimensional spaces (in this model we used the default, RBF kernel). Kernel choice does greatly impact the results of predictions. Finally, K-Nearest Neighbor predicts the value of a single data point by averaging values of its nearest neighbors. This can capture complex patterns however is sensitive to the distance metric selected.

```
"""
Purpose: HW1 - Evaluate different regressors for a 3 factor, 3 level model with
different design combos
Author: Syam Evani
"""

# Standard imports
import os

# Additional imports
import pandas as pd
import numpy as np
from sklearn.model_selection import train_test_split
from sklearn.linear_model import LinearRegression
from sklearn.tree import DecisionTreeRegressor
from sklearn.ensemble import RandomForestRegressor
from sklearn.svm import SVR
from sklearn.neighbors import KNeighborsRegressor
from sklearn.metrics import mean_squared_error
from pyDOE3 import fullfact
import matplotlib.pyplot as plt

# Local imports
# None

#----------------------------------------------------------------------
# Create design combos
#----------------------------------------------------------------------
# Define the levels for each factor (transformation)
# Features X1, X2, X3 -- three of them
# 0: No transformation, 1: Logarithmic, 2: Square root, 3: Square
levels = {0: lambda x: x,
          1: lambda x: np.log(x + 0.1),
          2: np.sqrt,
          3: np.square}

# Generating a full factorial design for 3 factors, each with 3 levels
design = fullfact([3, 3, 3])
```

```
# Show the design matrix
# print(design)
print(f"Number of Design:",len(design))

#-----------------------------------------------------------------
# Create sample dataset
#-----------------------------------------------------------------
# Sample dataset
# Replace this with your actual dataset
X = np.random.rand(100, 3)  # 100 samples, 3 features
y = 2*X[:, 0] + 3*np.log(X[:, 1] + 1) + np.sqrt(X[:, 2])   # Sample target
variable

# Splitting the dataset into training and testing sets
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2,
random_state=42)

#-----------------------------------------------------------------
# Apply different regression approaches
#-----------------------------------------------------------------
# Placeholder for results and best transformed features
results = {"lr": [],        # Linear regression
           "dtr": [],       # Decision tree regression
           "rfr": [],       # Random forest regression
           "svr": [],       # Support vector regression
           "knn": []        # K-nearest neighbors regression
            }
models = {}
predictions = {}
best_transformed_features = None

for i in range(len(design)):
    # Apply transformations based on the design matrix
     X_train_transformed = np.column_stack([levels[design[i, j]](X_train[:, j])
for j in range(X.shape[1])])
    X_test_transformed = np.column_stack([levels[design[i, j]](X_test[:, j]) for
j in range(X.shape[1])])

    # Train a linear regression model
```

```
    models["lr"] = LinearRegression().fit(X_train_transformed, y_train)
    models["dtr"] = DecisionTreeRegressor().fit(X_train_transformed, y_train)
    models["rfr"] = RandomForestRegressor().fit(X_train_transformed, y_train)
                models["svr"]    =    SVR().fit(X_train_transformed,    y_train)
# Default rbf kernel, 3 degree polynomial kernel function, uses 1/(n_features) *
X.var()) as gamma
      models["knn"]   =  KNeighborsRegressor().fit(X_train_transformed,  y_train)
# By default will use 5 neighbors

    # Predict and calculate MSE
    for regressor in models:
        predictions[regressor] = models[regressor].predict(X_test_transformed)
        mse = mean_squared_error(y_test, predictions[regressor])

        # Store the results
        results[regressor].append((design[i], mse, predictions[regressor]))

#----------------------------------------------------------------------
# Post process and plot different regressors for comparison
#----------------------------------------------------------------------
 # Plotting predictions against actual values
plt.figure(figsize=(10, 6))

# Post-process different regression approaches
for regressor in results:
    # Find the design with the lowest MSE
         min_mse_design,  min_mse,  best_predictions  =  min(results[regressor],
key=lambda x: x[1])

    # Print the design, MSE, and feature values with the lowest error
    print(f"Regressor: {regressor}")
    print(f"Design with lowest MSE: {min_mse_design}, MSE: {min_mse}")

    plt.scatter(y_test, best_predictions, alpha=0.7)
    plt.plot([min(y_test), max(y_test)], [min(y_test), max(y_test)], color='red',
linestyle='--')
    plt.xlabel("Actual values")
    plt.ylabel("Predicted values")
```

```
        plt.title(f"Predictions vs Actual for {regressor} with Design:
{min_mse_design} and MSE: {"{:.5f}".format(min_mse)}")
    plt.savefig(os.path.join('hw1', 'output', regressor + ".png"))
```