07zwe2mta

November 27, 2023

1 Assignment 1

2 1. Regression

The goal of this part of assignment is to apply regression analysis using splines and advanced decision tree regression techniques on a real-world dataset. You will explore the dataset, preprocess it, apply different regression models, and compare their performance.

2.1 Dataset

For this assignment, we will use the "California Housing Prices" dataset from the `sklearn.datasets` module for ease of access.

2.2 Tasks

2.2.1 1. Data Exploration and Preprocessing

- Load the California Housing Prices dataset.
- Perform exploratory data analysis (EDA) to understand the dataset.
- Visualize the distribution of the target variable and other features (plotting a histogram or a density plot of the target variable to see its distribution. This can reveal if the target is normally distributed, skewed, or has any unusual patterns.)
- Handle missing values if any (for now you can remove the observation).
- Normalize/standardize the features if required.

2.2.2 2. Regression with Splines

- Fit a spline model to the data. You may use the `patsy` library in Python for creating spline features.
- Experiment with different degrees of freedom to see how the model complexity affects the performance.
- Evaluate the model using appropriate metrics (e.g., RMSE, R^2).

2.2.3 3. Regression with ensemble Decision Trees

- Apply Random Forest, AdaBoost and GradientBoosting regression techniques (from `sklearn`)
- For each method, tune hyperparameters using cross-validation.
- Evaluate the models using appropriate metrics (e.g., RMSE, R^2).

2.2.4 4. Model Comparison

- Compare the performance of the spline models with the advanced decision tree regression models.
- Use visualizations to compare the predicted vs actual values for each model.
- Discuss the bias-variance tradeoff for each model based on your results.

2.2.5 5. Analysis and Discussion

- Discuss the performance of each model and the impact of hyperparameters on the outcome.
- Provide insights on which features are most important for predicting housing prices.

3 2. Classification

In this assignment, you will apply various classification techniques on a dataset to predict categorical outcomes. You will use Support Vector Machines (SVM), advanced decision tree classifiers, and Generalized Additive Models (GAMs) to build predictive models and compare their performance.

3.1 Dataset

We will use the "Breast Cancer Wisconsin (Diagnostic)" dataset for this assignment. This dataset is included in the `sklearn.datasets` module.

3.2 Tasks

3.2.1 1. Data Exploration and Preprocessing

- Begin by loading the dataset and conducting exploratory data analysis (EDA).
- Visualize the distribution of the classes (malignant and benign) and the features.
- Preprocess the data by handling missing values, encoding categorical variables if necessary, and scaling the features.

3.2.2 2. Classification with Support Vector Machines (SVM)

- Train an SVM classifier using the preprocessed data.
- Experiment with different kernels (linear, polynomial, and radial basis function) and regularization parameters.
- Evaluate the model using appropriate metrics (accuracy, precision, recall, F1-score, and ROC-AUC).

3.2.3 3. Classification with Advanced Decision Trees

- Apply advanced decision tree classification techniques such as Random Forest and Gradient Boosting.
- Tune hyperparameters with cross-validation.
- Evaluate the models using the same metrics as for SVM.

3.2.4 4. Classification with Generalized Additive Models (GAMs)

- Fit a GAM for classification using the `pyGAM` library.
- Select appropriate link functions and distribution families for the binary classification task.
- Visualize the contribution of each feature to the model using partial dependency plots.
- Evaluate the model using the same metrics as for SVM and decision trees.

3.2.5 5. Model Comparison and Analysis

- Compare the performance of SVM, advanced decision trees, and GAMs.
- Use confusion matrices and ROC curves to visualize the performance differences.
- Discuss the strengths and weaknesses of each model in the context of the dataset.

3.2.6 6. Conclusion

- Summarize the findings from the model comparisons.
- Provide insights into which model performed best and hypothesize why.
- Discuss any potential improvements or alternative approaches that could be explored.

3.3 How to Submit

- First, a Jupyter Notebook containing all the code, comments, and analysis.
- Second report cells in the same Jupyter Notebook, summarizing your findings, including results and a discussion of the results.
- Finally convert the Jupyter Notebook to PDF.
- **Don't write your name**.
- Upload the PDF into convas.

3.4 Evaluation Criteria (peer grading)

- Correctness of the implementation of all regression and classification models. (2 points)
- Quality of the EDA and preprocessing steps. (1 point)
- Depth of the analysis in comparing the models.(1 point)
- Clarity and organization of the submitted report and Jupyter Notebook. (1 point)

4 Task 1

[1]:
```
import pandas as pd
import numpy as np
import seaborn as sns
import matplotlib.pyplot as plt
import warnings
warnings.filterwarnings('ignore')
from statsmodels.stats.outliers_influence import variance_inflation_factor
from patsy import dmatrix
from sklearn.metrics import mean_squared_error, r2_score
from sklearn.linear_model import LinearRegression
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler
```

[2]:
```
# Load the California housing dataset
from sklearn.datasets import fetch_california_housing
california_housing = fetch_california_housing()
# Create a Pandas DataFrame from the dataset
ch_df = pd.DataFrame(data=california_housing.data, columns=california_housing.
 ↪feature_names)
```

[3]:
```
ch_df.head()
```

[3]:
```
   MedInc  HouseAge  AveRooms  AveBedrms  Population  AveOccup  Latitude  \
0  8.3252      41.0  6.984127   1.023810       322.0  2.555556     37.88
1  8.3014      21.0  6.238137   0.971880      2401.0  2.109842     37.86
2  7.2574      52.0  8.288136   1.073446       496.0  2.802260     37.85
3  5.6431      52.0  5.817352   1.073059       558.0  2.547945     37.85
4  3.8462      52.0  6.281853   1.081081       565.0  2.181467     37.85

   Longitude
0    -122.23
1    -122.22
2    -122.24
3    -122.25
4    -122.25
```

[4]:
```
# make a data frame with only the target variable
med_house_val_df = pd.DataFrame(data=california_housing.target,␣
 ↪columns=['MedHouseVal'])
med_house_val_df.head()
```

[4]:
```
   MedHouseVal
0        4.526
1        3.585
2        3.521
3        3.413
4        3.422
```

[5]:
```
med_house_val_df.mean()
```

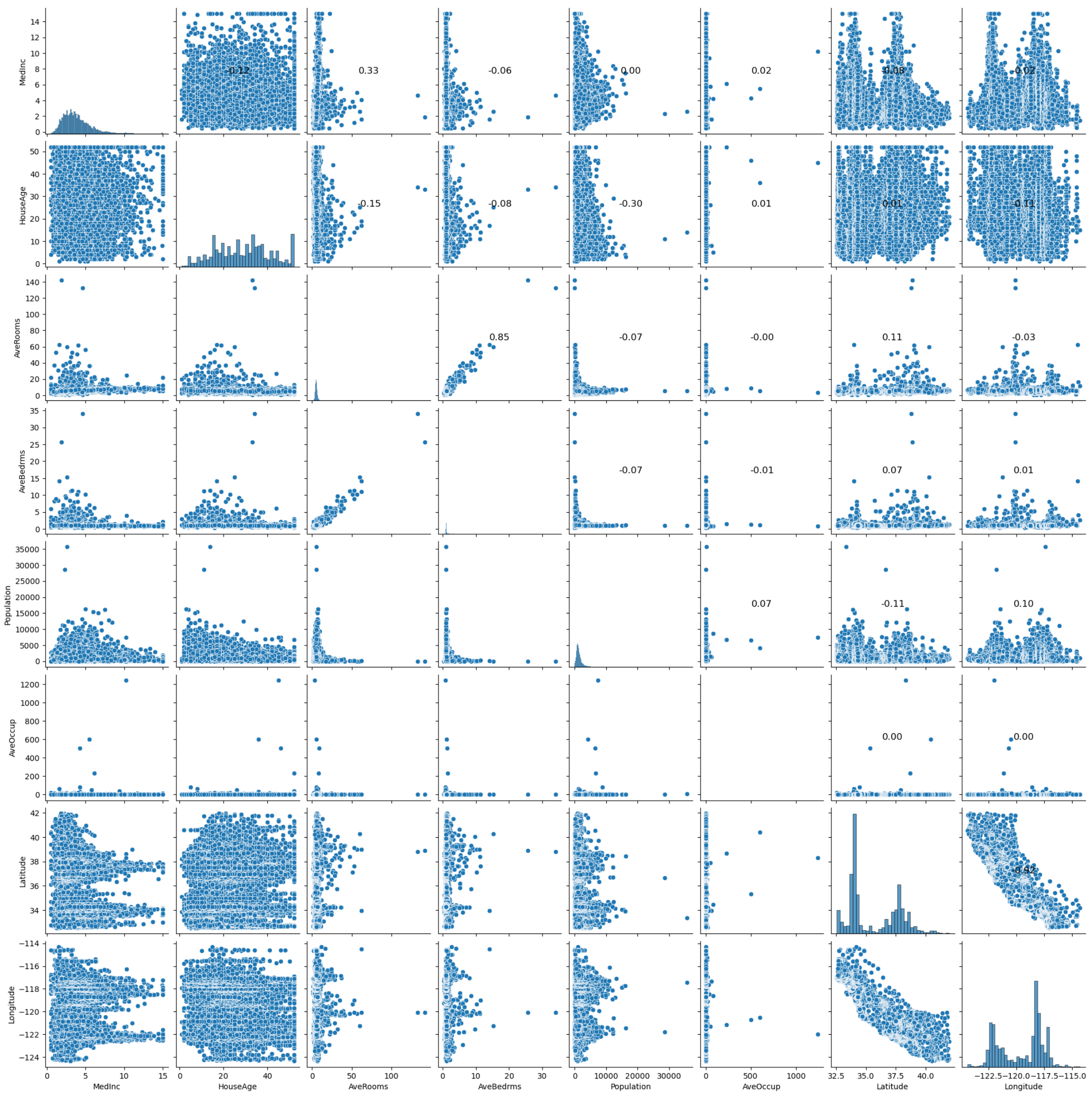
[5]:
```
MedHouseVal    2.068558
dtype: float64
```

[6]:
```
ch_df.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 20640 entries, 0 to 20639
```

```
Data columns (total 8 columns):
 #   Column      Non-Null Count  Dtype  
---  ------      --------------  -----  
 0   MedInc      20640 non-null  float64
 1   HouseAge    20640 non-null  float64
 2   AveRooms    20640 non-null  float64
 3   AveBedrms   20640 non-null  float64
 4   Population  20640 non-null  float64
 5   AveOccup    20640 non-null  float64
 6   Latitude    20640 non-null  float64
 7   Longitude   20640 non-null  float64
dtypes: float64(8)
memory usage: 1.3 MB
```

[7]:
```python
def pairplot_with_correlations(data):
    # Create a pairplot
    g = sns.pairplot(data, diag_kind='hist')

    # Loop through each axes
    for i in range(g.axes.shape[0]):
        for j in range(i+1, g.axes.shape[1]):
            ax = g.axes[i, j]
            # Compute and annotate the correlation value
            corr_value = data.corr().iloc[i, j]
            ax.annotate(f'{corr_value:.2f}', xy=(.5, .5), xycoords='axes␣
 ↪fraction',
                        ha='center', va='center', fontsize=12)

# Call the function with your data
pairplot_with_correlations(ch_df)

# Show the plot
plt.show()
```

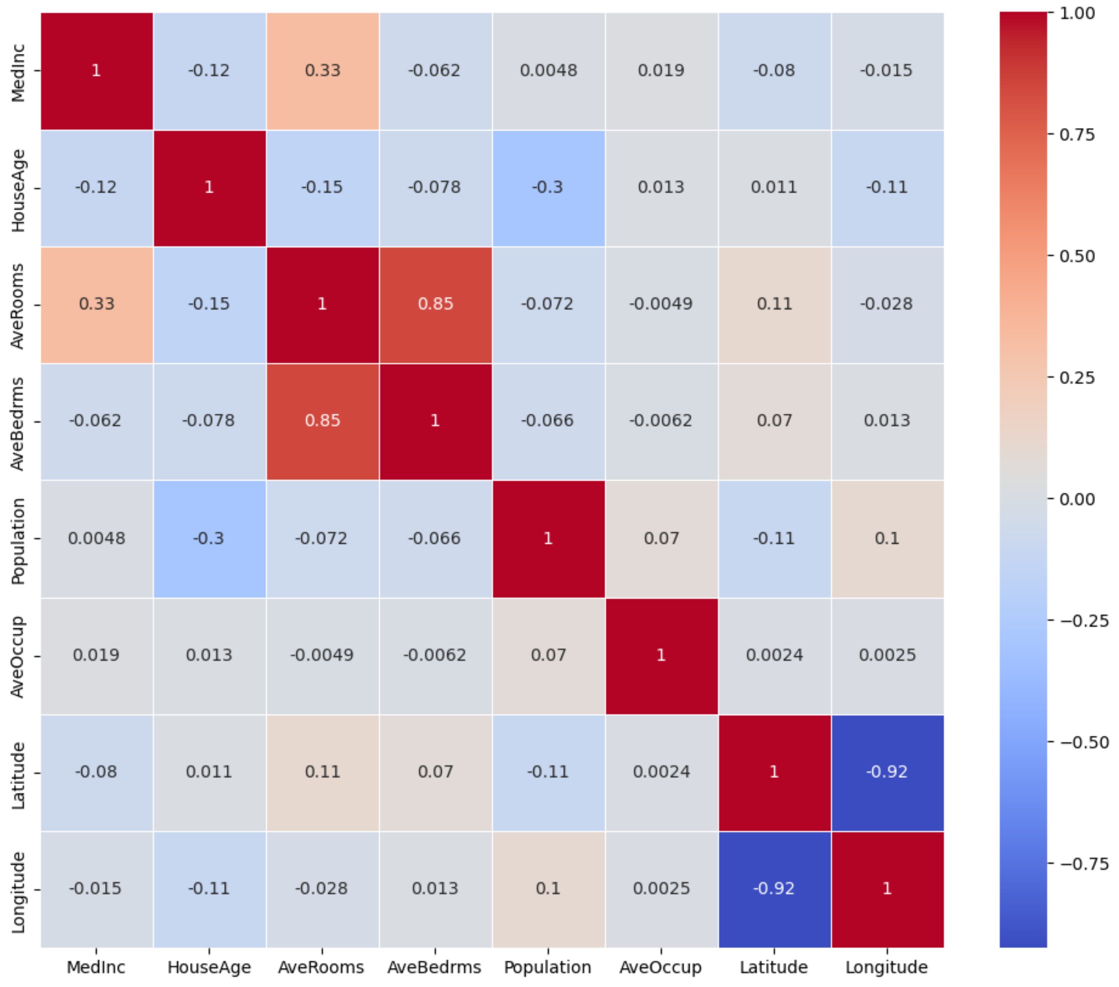


Diagonal Plots (Histograms/Density Plots): These plots appear along the diagonal of the grid. Each plot shows the distribution of a single variable. It's a way to visualize how the data points are spread out or concentrated across different values for that specific variable.

Lower Triangle Plots (Scatter Plots): These plots are located in the lower triangle of the grid (below the diagonal). Each scatter plot shows the relationship between two variables. Points on these plots represent individual data points, plotted according to their values for the two variables being compared. These plots are useful for visually identifying trends, correlations, or patterns between pairs of variables.

Upper Triangle Plots (Correlation Coefficients): These appear in the upper triangle of the grid (above the diagonal). In each of these plots, instead of a graph, you'll see a numerical value representing the Pearson correlation coefficient between the pair of variables. This coefficient is a measure of the linear correlation between the two variables, ranging from -1 to 1. A value close to 1 indicates a strong positive correlation, close to -1 indicates a strong negative correlation, and around 0 indicates no linear correlation.

[8]:
```python
# Create a correlation matrix
correlation_matrix = ch_df.corr()

# Set up the matplotlib figure
plt.figure(figsize=(12, 10))

# Create a heatmap using Seaborn
sns.heatmap(correlation_matrix, annot=True, cmap='coolwarm', linewidths=0.5)

# Show the plot
plt.show()
```



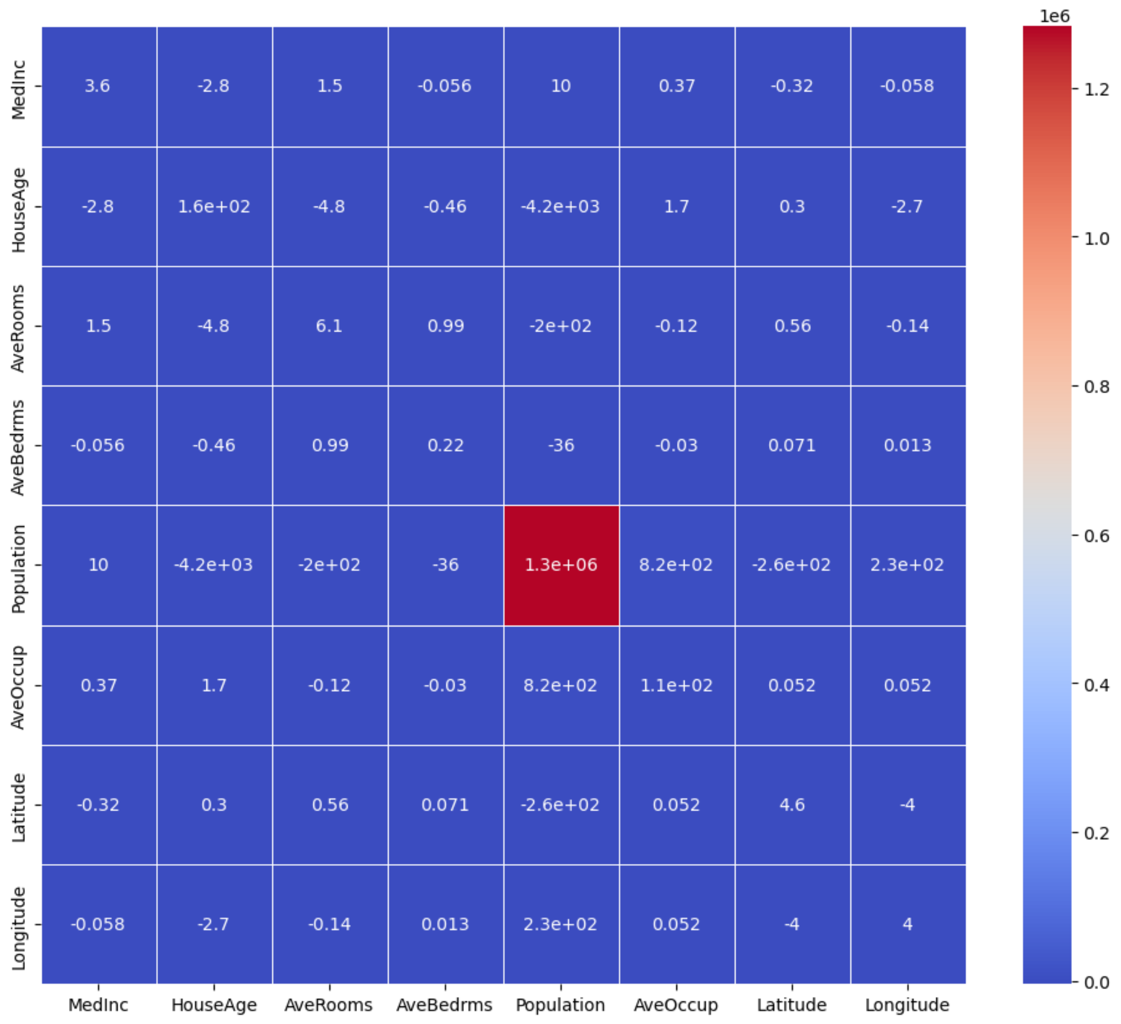
	MedInc	HouseAge	AveRooms	AveBedrms	Population	AveOccup	Latitude	Longitude
MedInc	1	-0.12	0.33	-0.062	0.0048	0.019	-0.08	-0.015
HouseAge	-0.12	1	-0.15	-0.078	-0.3	0.013	0.011	-0.11
AveRooms	0.33	-0.15	1	0.85	-0.072	-0.0049	0.11	-0.028
AveBedrms	-0.062	-0.078	0.85	1	-0.066	-0.0062	0.07	0.013
Population	0.0048	-0.3	-0.072	-0.066	1	0.07	-0.11	0.1
AveOccup	0.019	0.013	-0.0049	-0.0062	0.07	1	0.0024	0.0025
Latitude	-0.08	0.011	0.11	0.07	-0.11	0.0024	1	-0.92
Longitude	-0.015	-0.11	-0.028	0.013	0.1	0.0025	-0.92	1

By analyzing the plot we can see that longitude and latitude have a very high correlation (almost 1) which could impair the model.

So first we remove longitude from the dataset

Futhermore a more viable way to see correlation is by lokking at the VIF-value for every varibale to identify multcolinearity.

[9]:
```
cov_matrix = ch_df.cov()
plt.figure(figsize=(12, 10))

sns.heatmap(cov_matrix, annot=True, cmap='coolwarm', linewidths=0.5)

plt.show()
```



	MedInc	HouseAge	AveRooms	AveBedrms	Population	AveOccup	Latitude	Longitude
MedInc	3.6	-2.8	1.5	-0.056	10	0.37	-0.32	-0.058
HouseAge	-2.8	1.6e+02	-4.8	-0.46	-4.2e+03	1.7	0.3	-2.7
AveRooms	1.5	-4.8	6.1	0.99	-2e+02	-0.12	0.56	-0.14
AveBedrms	-0.056	-0.46	0.99	0.22	-36	-0.03	0.071	0.013
Population	10	-4.2e+03	-2e+02	-36	1.3e+06	8.2e+02	-2.6e+02	2.3e+02
AveOccup	0.37	1.7	-0.12	-0.03	8.2e+02	1.1e+02	0.052	0.052
Latitude	-0.32	0.3	0.56	0.071	-2.6e+02	0.052	4.6	-4
Longitude	-0.058	-2.7	-0.14	0.013	2.3e+02	0.052	-4	4

[10]:
```
ch_df_02 = ch_df.drop(ch_df.columns[7], axis=1)
ch_df_02.head()
```

```
[10]:    MedInc  HouseAge  AveRooms  AveBedrms  Population  AveOccup  Latitude
      0  8.3252      41.0  6.984127   1.023810       322.0  2.555556     37.88
      1  8.3014      21.0  6.238137   0.971880      2401.0  2.109842     37.86
      2  7.2574      52.0  8.288136   1.073446       496.0  2.802260     37.85
      3  5.6431      52.0  5.817352   1.073059       558.0  2.547945     37.85
      4  3.8462      52.0  6.281853   1.081081       565.0  2.181467     37.85
```

```
[11]: # Create a DataFrame to store the VIF values
      vif_data = pd.DataFrame()
      vif_data["Variable"] = ch_df_02.columns

      # Calculate VIF for each variable
      vif_data["VIF"] = [variance_inflation_factor(ch_df_02.values, i) for i in␣
       ↪range(ch_df_02.shape[1])]

      # Display the VIF values
      print(vif_data)
```

```
     Variable        VIF
0      MedInc   9.865861
1    HouseAge   6.880512
2    AveRooms  42.192223
3   AveBedrms  39.768396
4  Population   2.793169
5    AveOccup   1.094908
6    Latitude  22.498755
```

```
[12]: ch_df_03 = ch_df_02.drop(ch_df.columns[2], axis=1)
      ch_df_03.head()
```

```
[12]:    MedInc  HouseAge  AveBedrms  Population  AveOccup  Latitude
      0  8.3252      41.0   1.023810       322.0  2.555556     37.88
      1  8.3014      21.0   0.971880      2401.0  2.109842     37.86
      2  7.2574      52.0   1.073446       496.0  2.802260     37.85
      3  5.6431      52.0   1.073059       558.0  2.547945     37.85
      4  3.8462      52.0   1.081081       565.0  2.181467     37.85
```

```
[13]: # Create a DataFrame to store the VIF values
      vif_data_02 = pd.DataFrame()
      vif_data_02["Variable"] = ch_df_03.columns

      # Calculate VIF for each variable
      vif_data_02["VIF"] = [variance_inflation_factor(ch_df_03.values, i) for i in␣
       ↪range(ch_df_03.shape[1])]

      # Display the VIF values
      print(vif_data_02)
```

```
    Variable        VIF
0     MedInc   5.036342
1   HouseAge   6.660725
2   AveBedrms  6.430073
3  Population  2.752356
4   AveOccup   1.094801
5   Latitude  21.922515
```

[14]:
```
ch_df_04 = ch_df_03.drop(ch_df.columns[6], axis=1)
ch_df_04.head()
```

[14]:
```
   MedInc  HouseAge  AveBedrms  Population  AveOccup
0  8.3252      41.0   1.023810       322.0  2.555556
1  8.3014      21.0   0.971880      2401.0  2.109842
2  7.2574      52.0   1.073446       496.0  2.802260
3  5.6431      52.0   1.073059       558.0  2.547945
4  3.8462      52.0   1.081081       565.0  2.181467
```

[15]:
```
# Create a DataFrame to store the VIF values
vif_data_03 = pd.DataFrame()
vif_data_03["Variable"] = ch_df_04.columns

# Calculate VIF for each variable
vif_data_03["VIF"] = [variance_inflation_factor(ch_df_04.values, i) for i in␣
 ↪range(ch_df_04.shape[1])]

# Display the VIF values
print(vif_data_03)
```

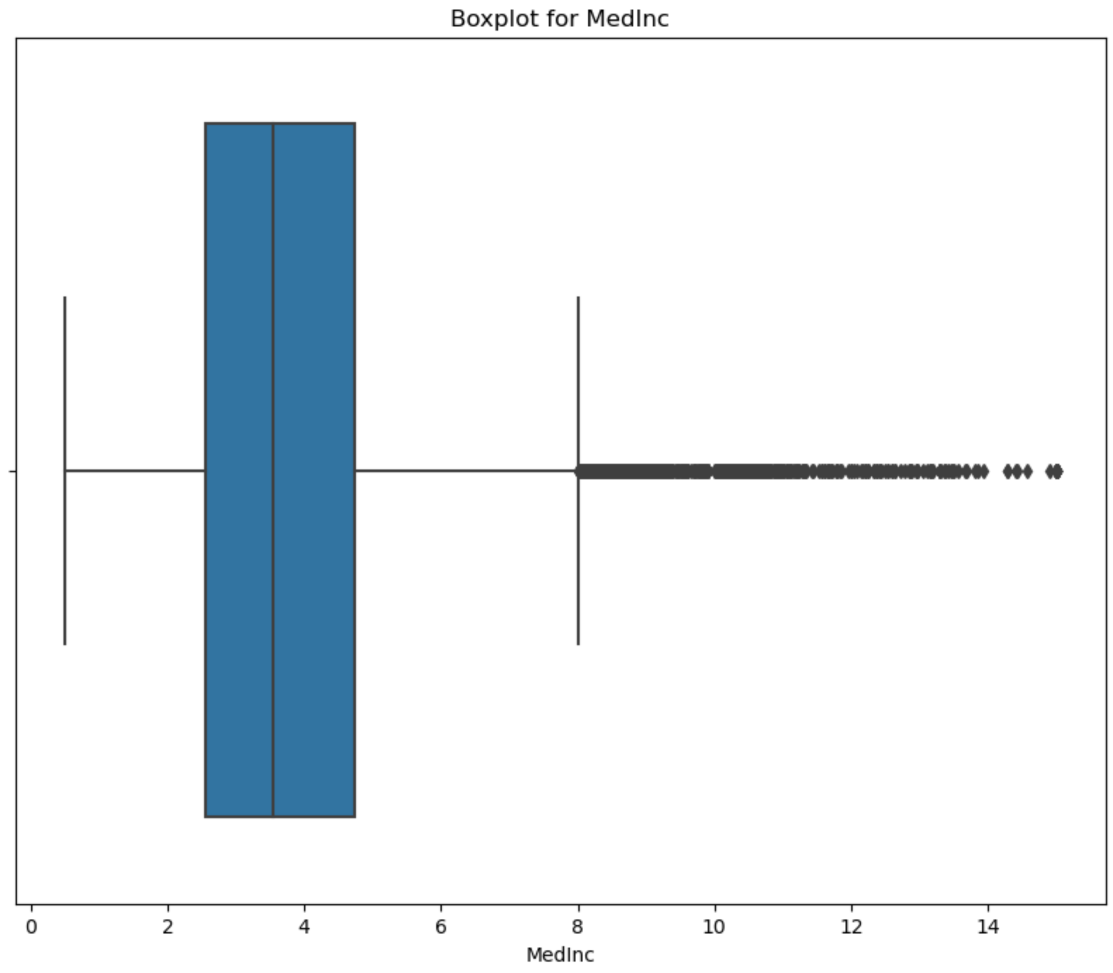
```
    Variable       VIF
0     MedInc  3.801038
1   HouseAge  3.769898
2   AveBedrms 4.263506
3  Population 2.222516
4   AveOccup  1.094644
```

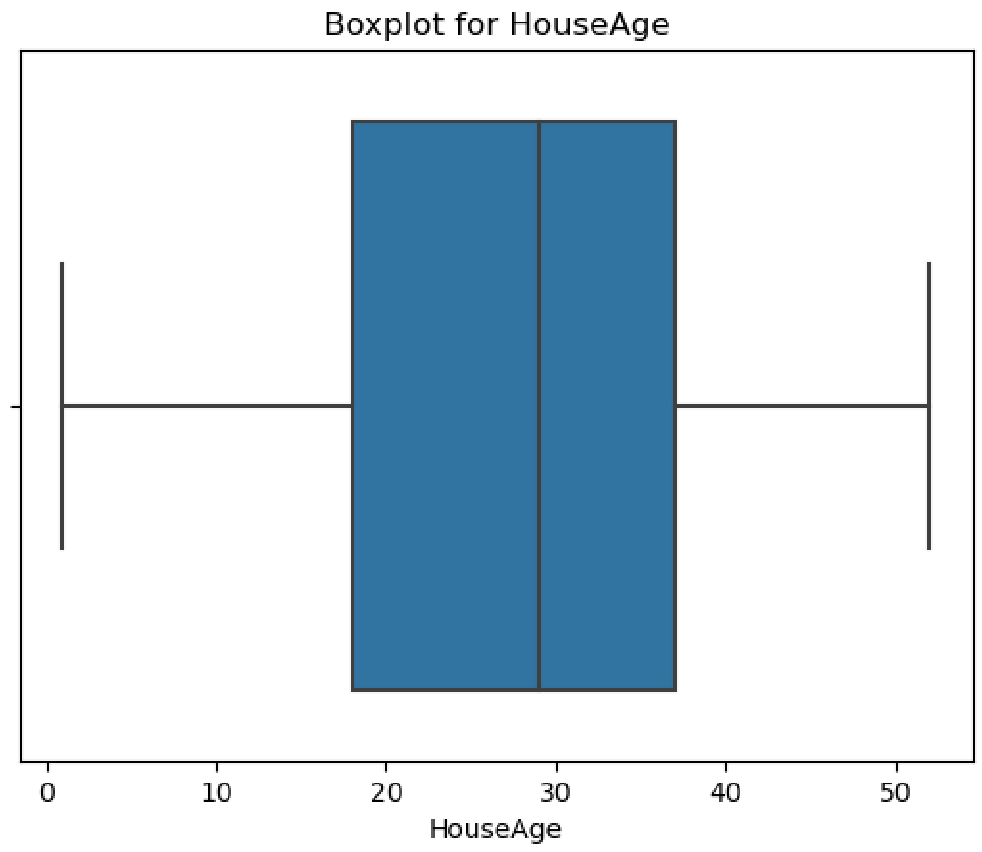
How to interpret VIF-Value:

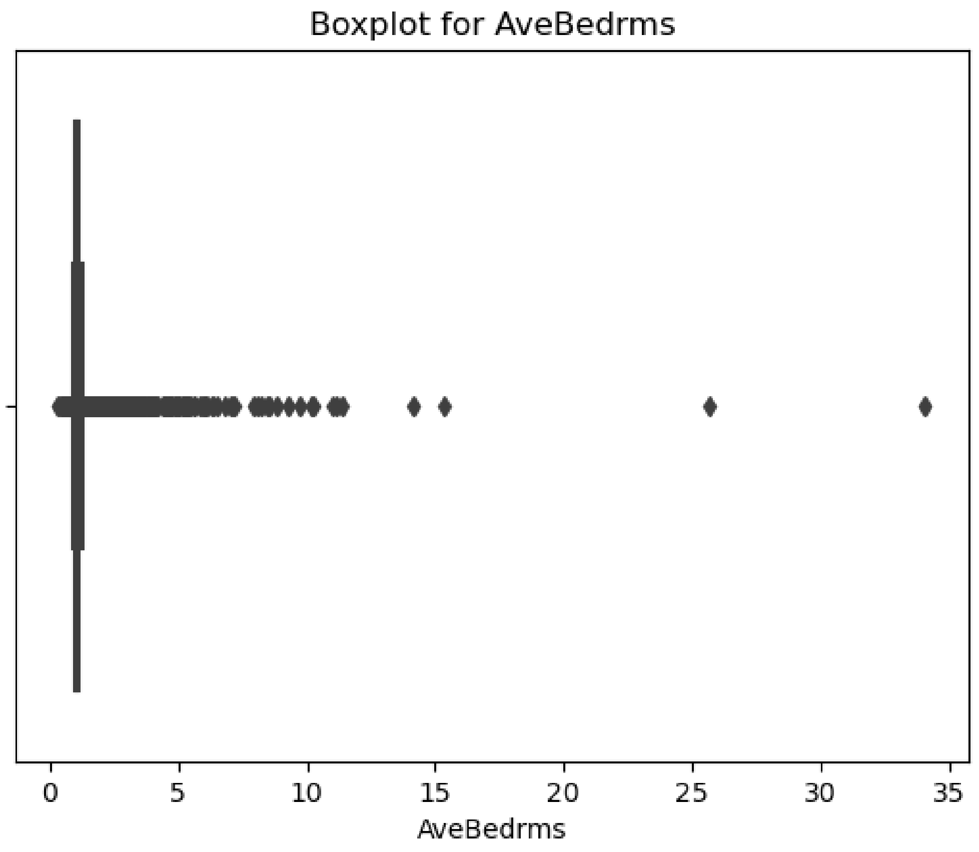
A value between 0-5 is acceptble A value between 5-10 should be investigated A value >10 should be removed

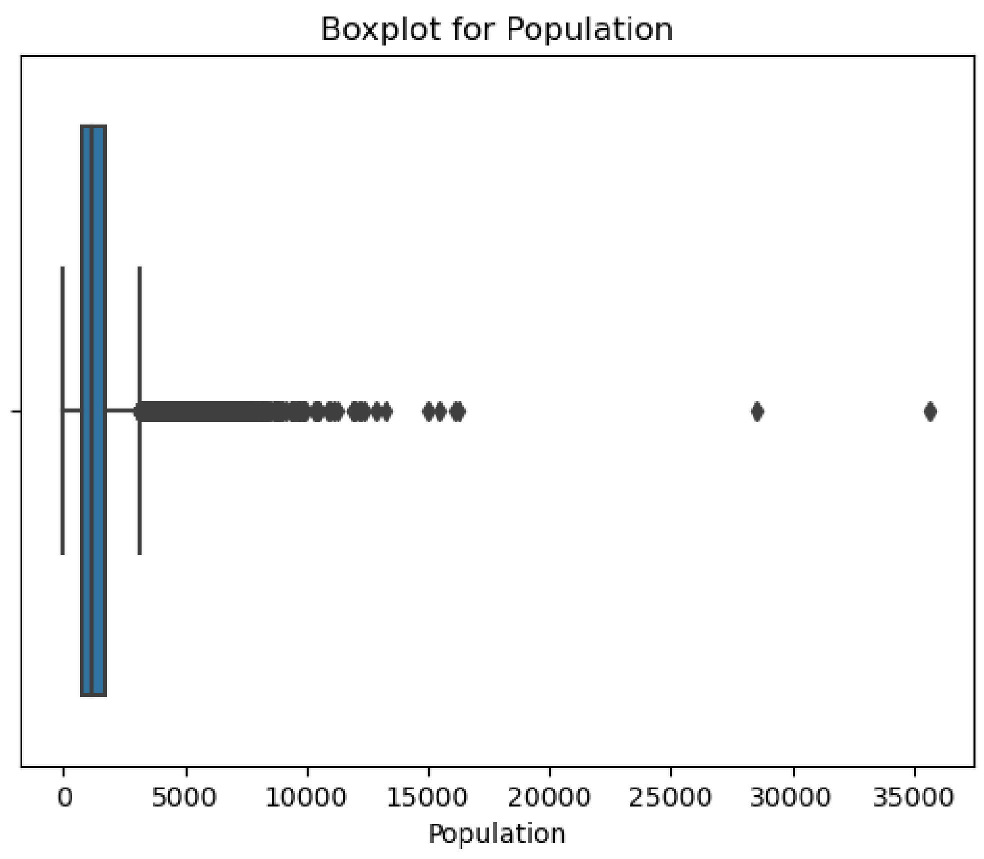
with our 5 varibales all of them has a VIF-value below 5

[16]:
```
# Create a boxplot for each variable in ch_df_03
plt.figure(figsize=(10, 8))  # Set the figure size

# Loop through each column in the DataFrame and create a boxplot
for column in ch_df_03.columns:
    sns.boxplot(x=ch_df_03[column], orient='h', fliersize=5)
    plt.title(f'Boxplot for {column}')
    plt.show()
```

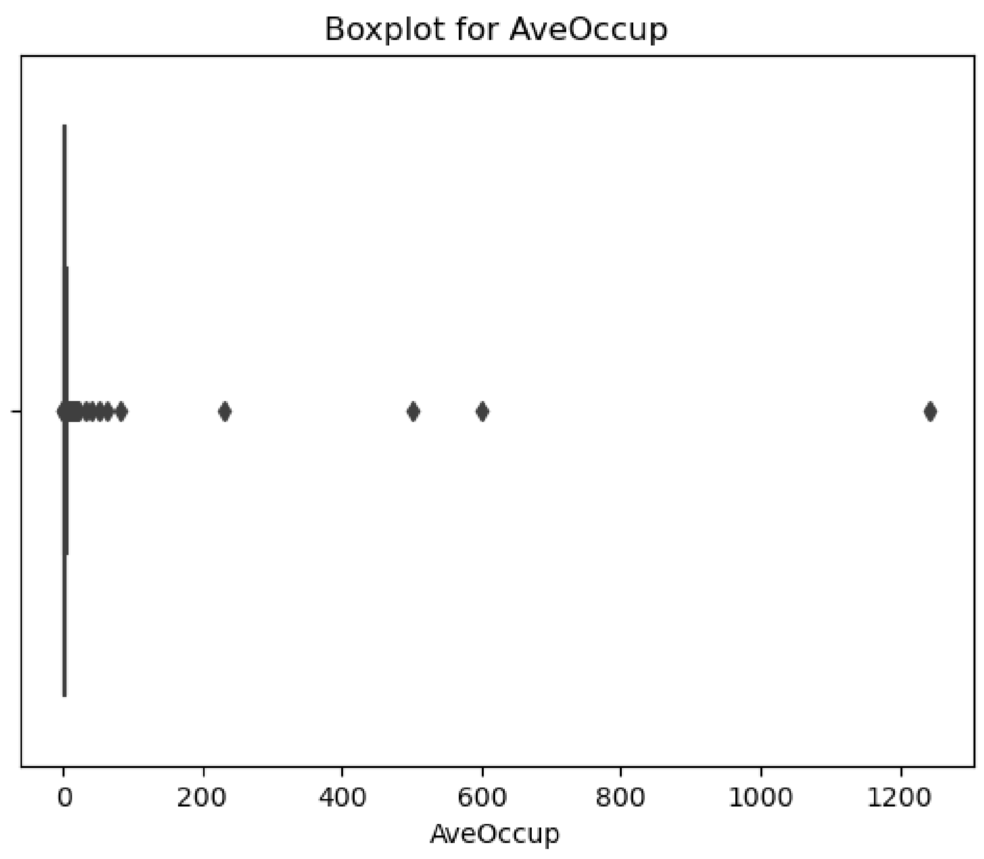


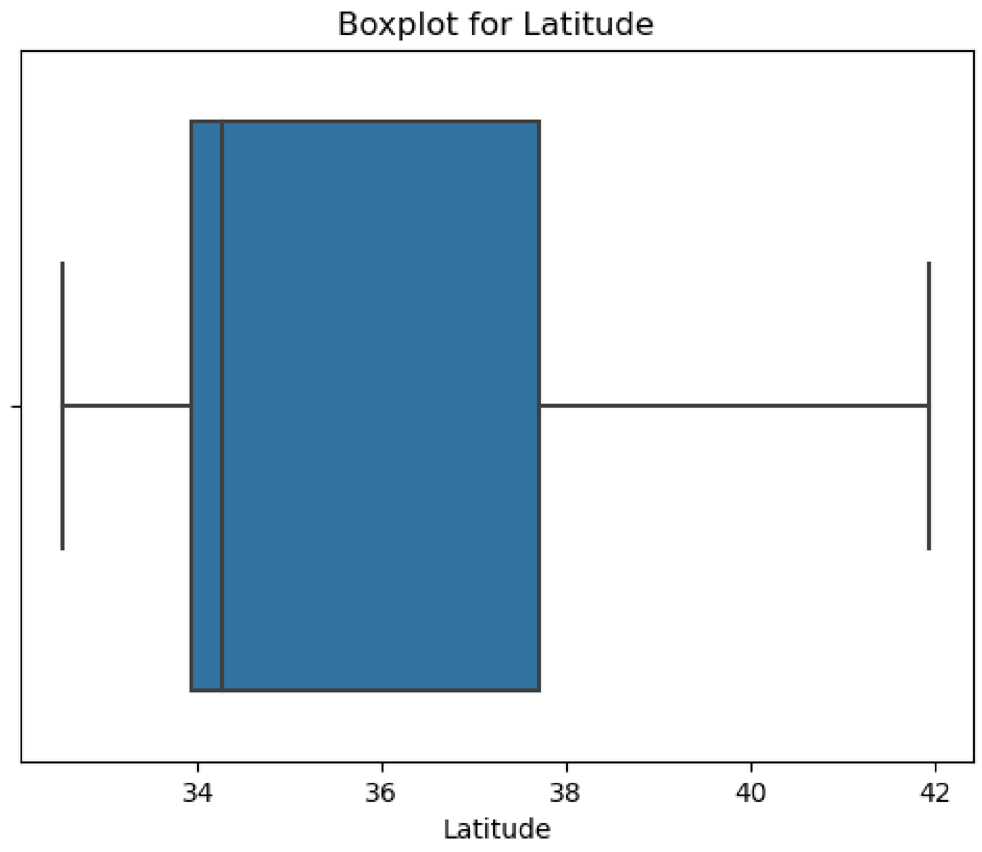

Boxplot for HouseAge
0
10
20
30
40
50
HouseAge


Boxplot for AveBedrms
0
5
10
15
20
25
30
35
AveBedrms

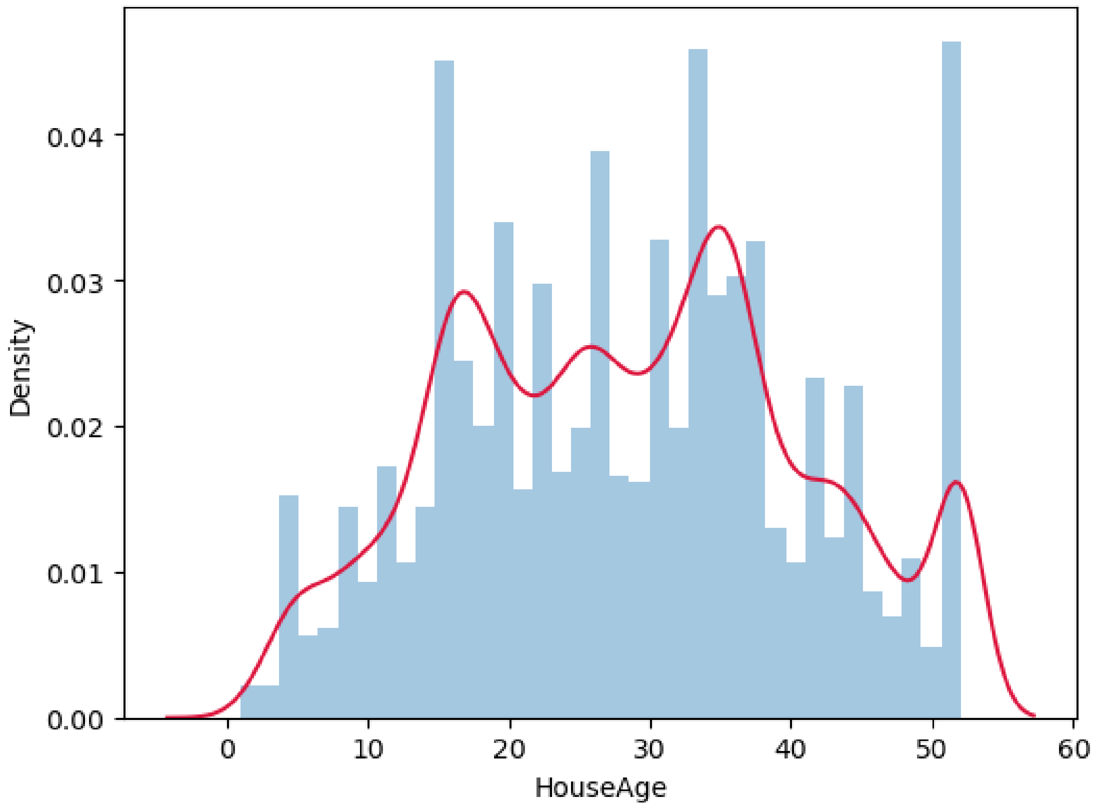


Boxplot for Population
0
5000
10000
15000
20000
25000
30000
35000
Population


Boxplot for AveOccup
0
200
400
600
800
1000
1200
AveOccup

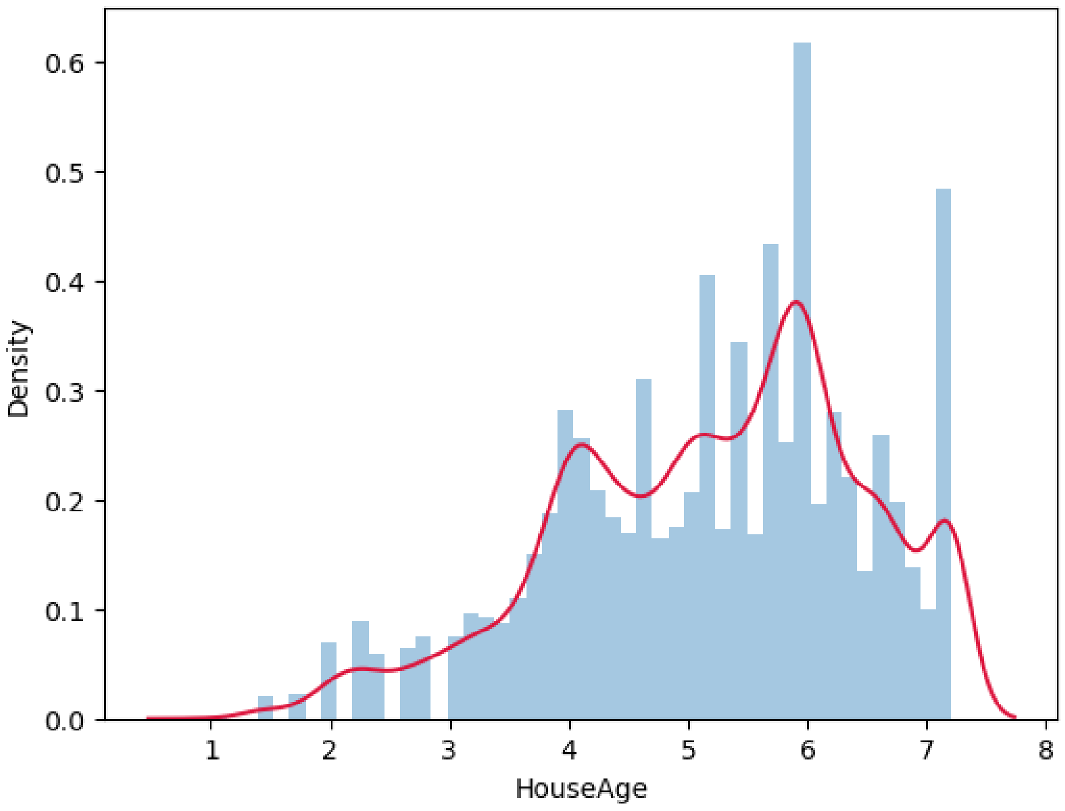


[17]:
```
sns.distplot(a=ch_df_04.HouseAge, kde=True)
plt.gca().lines[0].set_color('crimson')
plt.show()
```



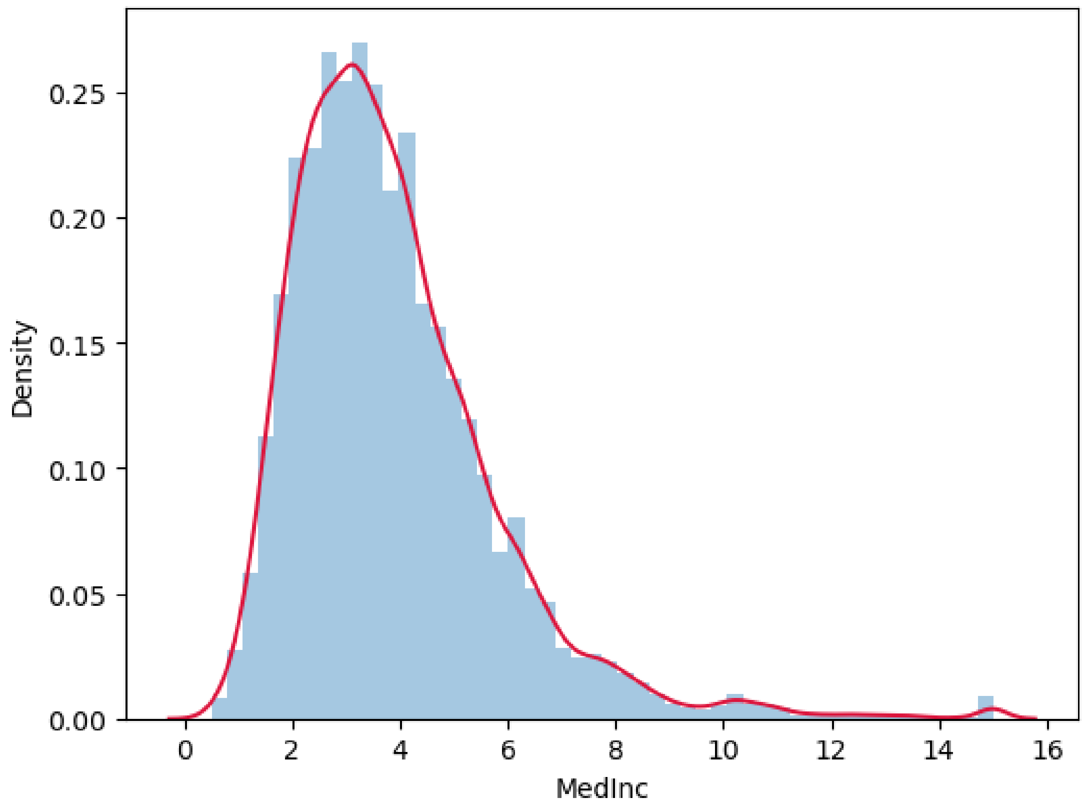
This is not normal distributed so some transformation is required.

[18]:
```
ch_df_04['HouseAge'] = np.sqrt((ch_df_04['HouseAge']))
sns.distplot(a=ch_df_04.HouseAge, kde=True)
plt.gca().lines[0].set_color('crimson')
plt.show()
```

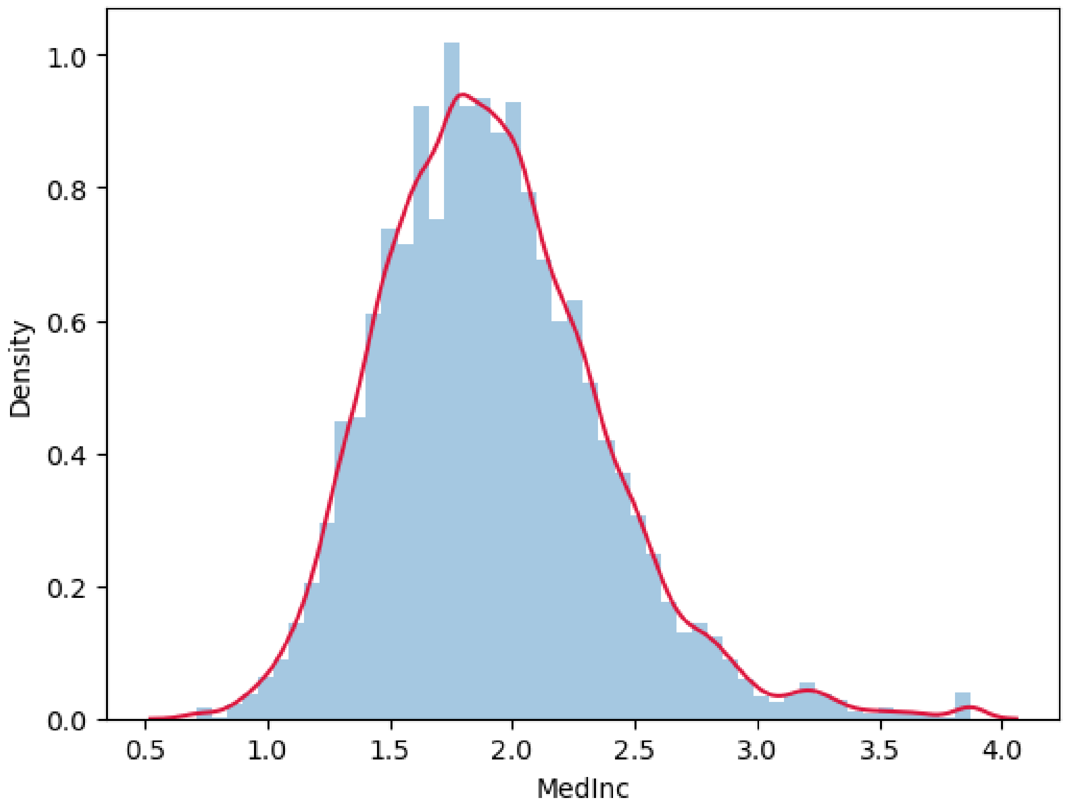


Good enough, i have tried different methods but this one were the best. Lets do the same for every verible we have left.

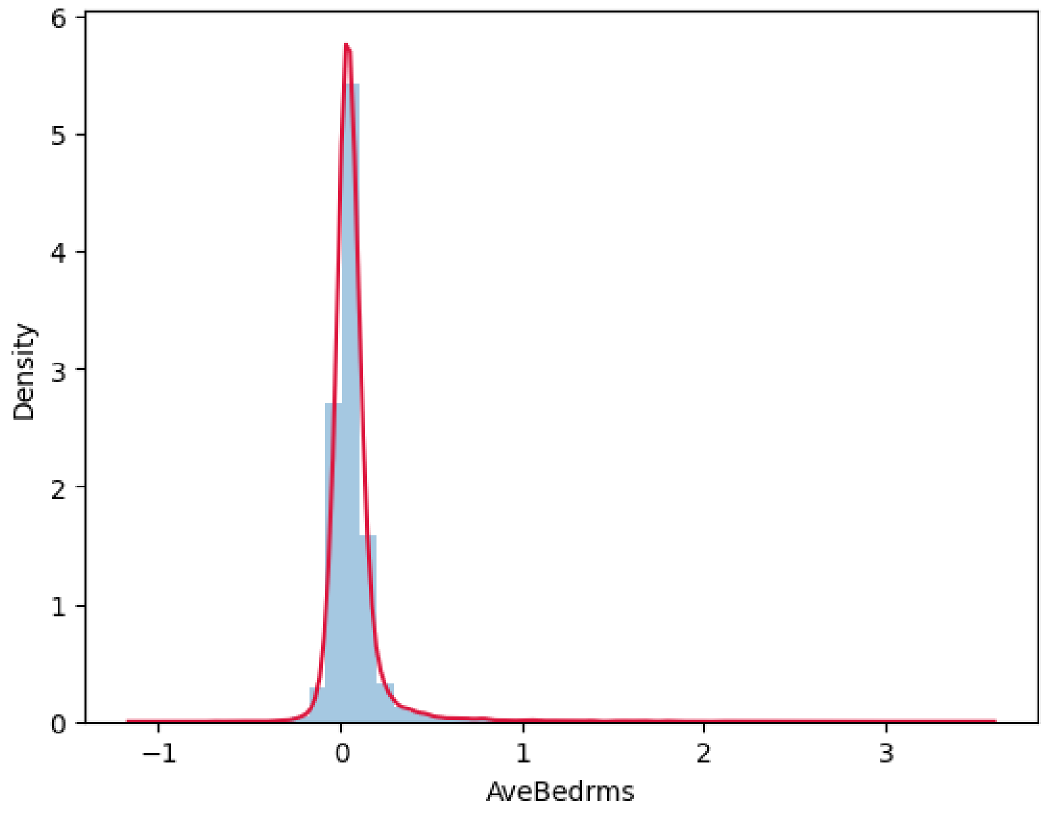
[19]:
```
sns.distplot(a=ch_df_04.MedInc, kde=True)
plt.gca().lines[0].set_color('crimson')
plt.show()
```



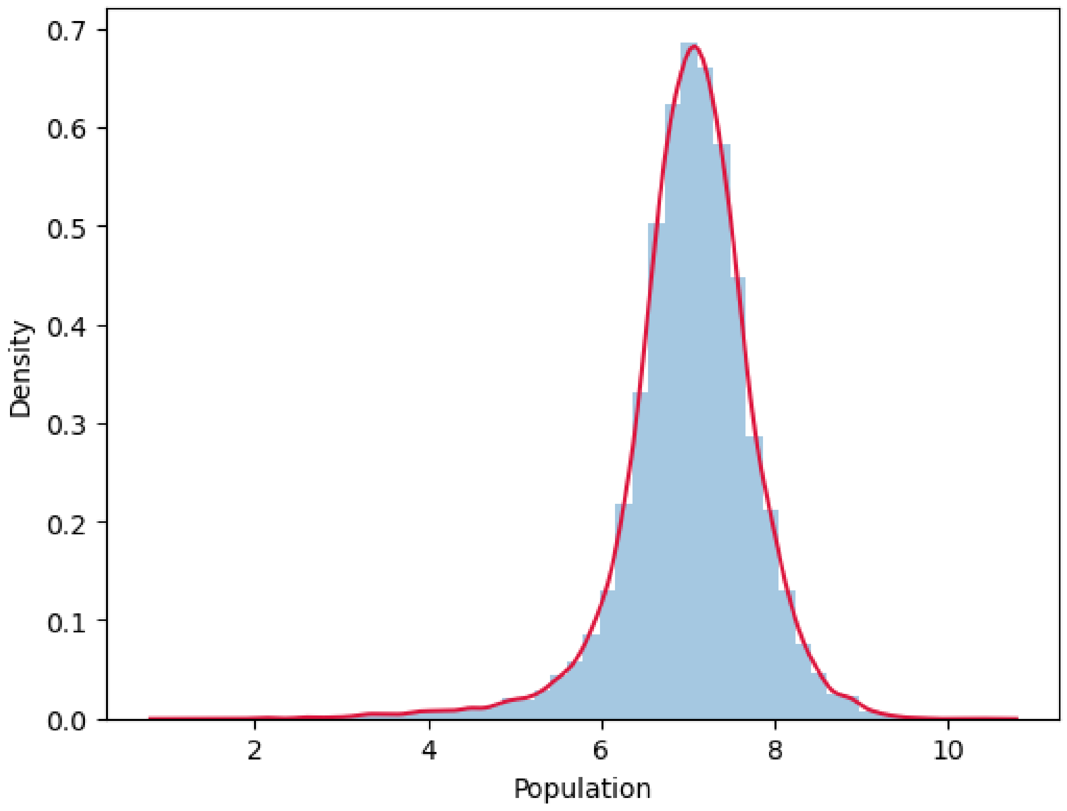
```
[20]: ch_df_04['MedInc'] = np.sqrt((ch_df_04['MedInc']))
      sns.distplot(a=ch_df_04.MedInc, kde=True)
      plt.gca().lines[0].set_color('crimson')
      plt.show()
```



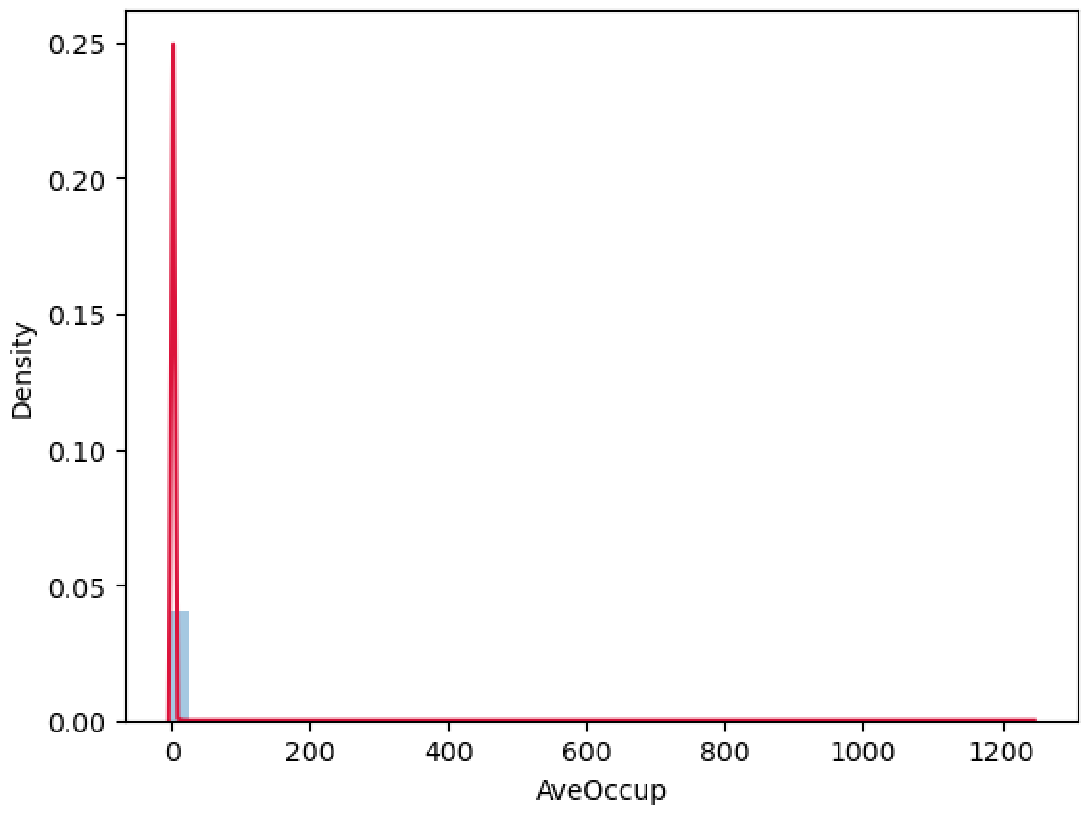
[21]:
```
ch_df_04['AveBedrms'] = np.log((ch_df_04['AveBedrms']))

sns.distplot(a=ch_df_04.AveBedrms, kde=True)
plt.gca().lines[0].set_color('crimson')
plt.show()
```



```
[22]: ch_df_04['Population'] = np.log((ch_df_04['Population']))

      sns.distplot(a=ch_df_04.Population, kde=True)
      plt.gca().lines[0].set_color('crimson')
      plt.show()
```



```
[23]: sns.distplot(a=ch_df_04.AveOccup, kde=True)
      plt.gca().lines[0].set_color('crimson')
      plt.show()
```



After all the transormations are done lets see if there is any missing values in the dataset we need to fix.

[24]:
```
# Check for missing values in ch_df_03
missing_values = ch_df_04.isnull().sum()

# Display the columns with missing values (if any) and the count of missing␣
  ↪values
print(missing_values[missing_values > 0])
```

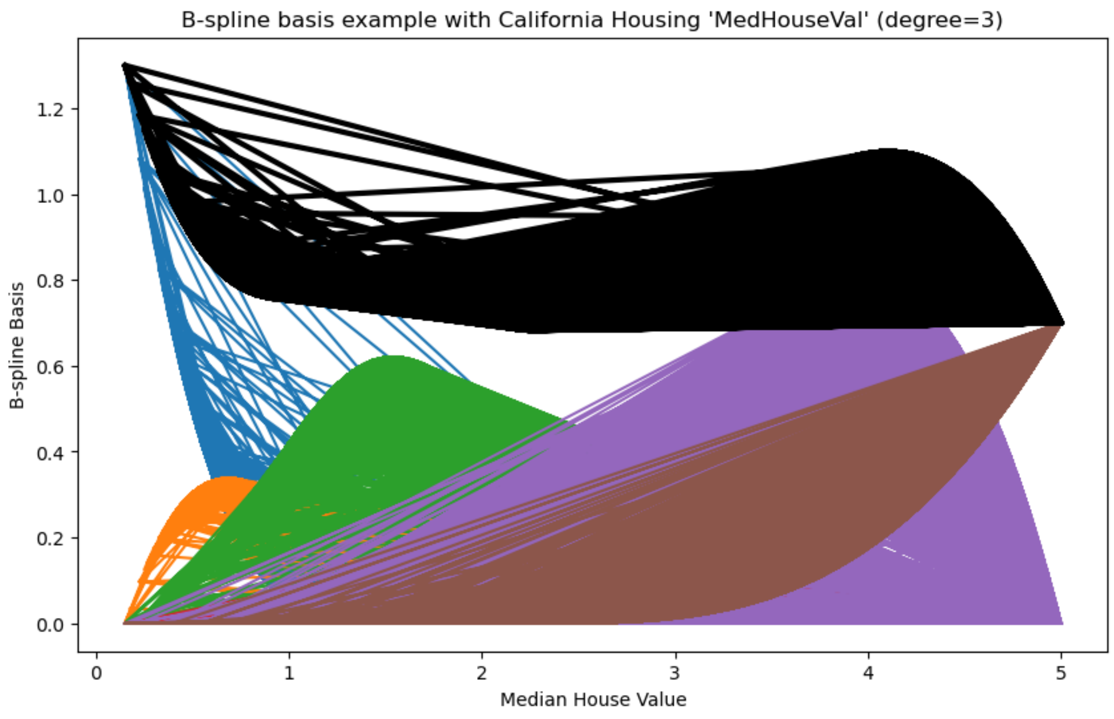
```
Series([], dtype: int64)
```

The output above shows that there is no missing value in the dataset.

Each of these features likely operates on a different scale and unit. For instance, population median income could be the actuall value, age of the house in years, average bedrooms as a count. These vastly different scales can impact the performance of many regression models. However with our transformation these should not be an issue for our dataset

[25]:
```
# Select a variable for B-spline basis
x = med_house_val_df

# Make sure x is sorted (important for plotting)
```

```
x_sorted = np.sort(x)

# Create B-spline basis matrix
y = dmatrix("bs(x_sorted, df=6, degree=3, include_intercept=True) - 1",␣
 ↪{"x_sorted": x_sorted})

# Define some coefficients
b = np.array([1.3, 0.6, 0.9, 0.4, 1.6, 0.7])

# Plot B-spline basis functions (colored curves) each multiplied by its␣
 ↪coefficient
plt.figure(figsize=(10, 6))
plt.plot(x_sorted, y*b)

# Plot the spline itself (sum of the basis functions, thick black curve)
plt.plot(x_sorted, np.dot(y, b), color='k', linewidth=3)

plt.title("B-spline basis example with California Housing 'MedHouseVal'␣
 ↪(degree=3)")
plt.xlabel("Median House Value")
plt.ylabel("B-spline Basis")
plt.show()
```



The plot shows the B-spline basis on the y-axis against the Median House Value on the x-axis.

Each colored curve represents a B-spline basis function, and the black curve is the sum of these basis functions (i.e., the actual B-spline curve fitted to the data).

This kind of plot is used in statistical modeling to show the components of a B-spline fit. It can be used in regression models where you want to fit a non-linear relationship between a predictor (in this case, Median House Value) and a response variable. Each basis function contributes to the final fitted curve, allowing for a flexible fit that can model non-linear trends in data.

[26]:
```
formula = "cr(MedInc+HouseAge+AveBedrms+Population+AveOccup, knots = [100,160])"
transformed_x = dmatrix(formula, data=ch_df_04, return_type='dataframe')

display(transformed_x)



model = LinearRegression()
model.fit(transformed_x,med_house_val_df)

MedHouseVal_pred = model.predict(transformed_x)

mean_squared_error(med_house_val_df, MedHouseVal_pred)
```

```
       Intercept  \
0            1.0
1            1.0
2            1.0
3            1.0
4            1.0
…          …
20635        1.0
20636        1.0
20637        1.0
20638        1.0
20639        1.0

       cr(MedInc + HouseAge + AveBedrms + Population + AveOccup, knots=[100,␣
 ↪160])[0]  \
0                                               0.838672
1                                               0.842982
2                                               0.820234
3                                               0.826481
4                                               0.836934
…                                                  …
20635                                           0.865778
20636                                           0.873513
20637                                           0.878114
20638                                           0.882008
20639                                           0.867523
```

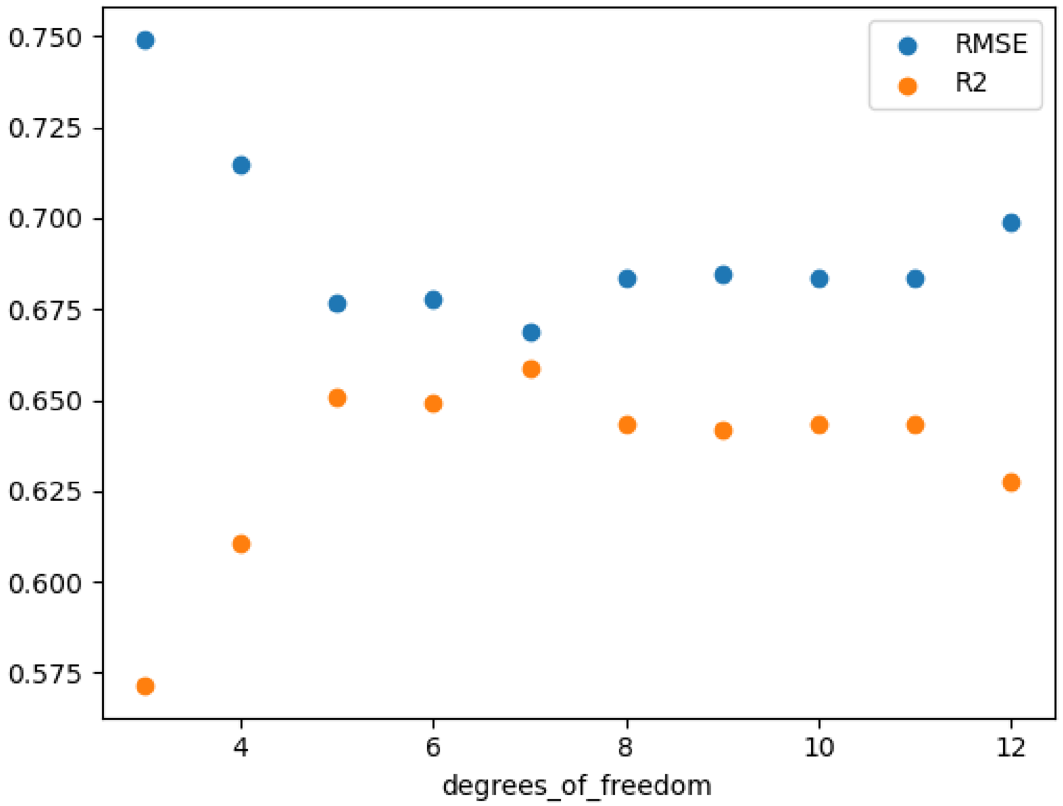
```
       cr(MedInc + HouseAge + AveBedrms + Population + AveOccup, knots=[100,␣
 ↪160])[1]  \
0                                                 0.221476
1                                                 0.215596
2                                                 0.246592
3                                                 0.238089
4                                                 0.223846
…                                                    …
20635                                             0.184449
20636                                             0.173862
20637                                             0.167562
20638                                             0.162229
20639                                             0.182061

       cr(MedInc + HouseAge + AveBedrms + Population + AveOccup, knots=[100,␣
 ↪160])[2]  \
0                                                -0.060230
1                                                -0.058658
2                                                -0.066917
3                                                -0.064659
4                                                -0.060863
…                                                    …
20635                                            -0.050296
20636                                            -0.047441
20637                                            -0.045739
20638                                            -0.044297
20639                                            -0.049652

       cr(MedInc + HouseAge + AveBedrms + Population + AveOccup, knots=[100,␣
 ↪160])[3]
0                                                 0.000082
1                                                 0.000080
2                                                 0.000092
3                                                 0.000088
4                                                 0.000083
…                                                    …
20635                                             0.000069
20636                                             0.000065
20637                                             0.000063
20638                                             0.000061
20639                                             0.000068

[20640 rows x 5 columns]
```

```
[26]: 1.3160881150454604
```

[27]:
```
scaler = StandardScaler()
scaled_features = scaler.fit_transform(ch_df)
scaled_features_df = pd.DataFrame(scaled_features, columns=ch_df.columns)
```

[28]:
```
X_train, X_test, y_train, y_test = train_test_split(scaled_features_df,␣
  ↪med_house_val_df, test_size=0.2, random_state=42)
feature_names = scaled_features_df.columns
```

[29]:
```
# Experimenting with different degrees of freedom for the spline model
degrees_of_freedom = np.arange(3,13)
rmse =[]
r2=[]
for df in degrees_of_freedom:


    formula = ' + '.join([f"cr({ch_df}, df="+str(df)+")" for ch_df in␣
  ↪feature_names])

    transformed_X_train = dmatrix(formula, data=X_train,␣
  ↪return_type='dataframe')
    transformed_X_test = dmatrix(formula, data=X_test, return_type='dataframe')

     # Fit the model
    model = LinearRegression()
    model.fit(transformed_X_train, y_train)

    # Predict on the test set
    y_pred = model.predict(transformed_X_test)

    # Evaluate the model
    rmse.append(np.sqrt(mean_squared_error(y_test, y_pred)))
    r2.append(r2_score(y_test, y_pred))


plt.scatter(degrees_of_freedom, rmse)
plt.scatter(degrees_of_freedom, r2)
plt.xlabel('degrees_of_freedom')
plt.legend(['RMSE' , 'R2'])
plt.show()
```



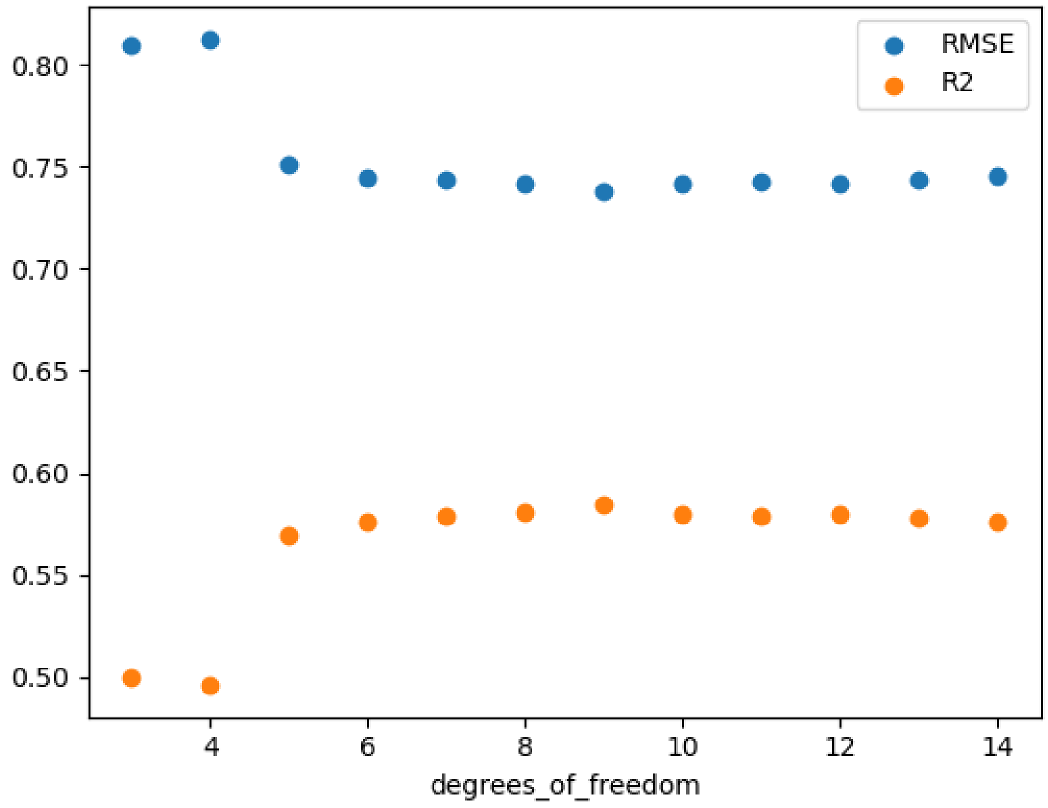
lets try with my own created data frame, it will probaly be worst but i wont just take your code.

[30]:
```
scaler_my = StandardScaler()
scaled_features_my = scaler_my.fit_transform(ch_df_04)
scaled_features_df_my = pd.DataFrame(ch_df_04, columns=ch_df_04.columns)
```

[31]:
```
X_train_my, X_test_my, y_train_my, y_test_my =␣
 ↪train_test_split(scaled_features_df_my, med_house_val_df, test_size=0.2,␣
 ↪random_state=42)
feature_names_my = scaled_features_df_my.columns
```

[32]:
```
degrees_of_freedom_my = np.arange(3,15)
rmse_my =[]
r2_my=[]
for df in degrees_of_freedom_my:


    formula = ' + '.join([f"cr({ch_df_04}, df="+str(df)+")" for ch_df_04 in␣
 ↪feature_names_my])

```

```
    transformed_X_train_my = dmatrix(formula, data=X_train_my,␣
↪return_type='dataframe')
    transformed_X_test_my = dmatrix(formula, data=X_test_my,␣
↪return_type='dataframe')

     # Fit the model
    model_my = LinearRegression()
    model_my.fit(transformed_X_train_my, y_train_my)

    # Predict on the test set
    y_pred_my = model_my.predict(transformed_X_test_my)

    # Evaluate the model
    rmse_my.append(np.sqrt(mean_squared_error(y_test_my, y_pred_my)))
    r2_my.append(r2_score(y_test_my, y_pred_my))


plt.scatter(degrees_of_freedom_my, rmse_my)
plt.scatter(degrees_of_freedom_my, r2_my)
plt.xlabel('degrees_of_freedom')
plt.legend(['RMSE' , 'R2'])
plt.show()
```



My feture values preformes best with 9 degree of freedoms. But its worst then the code that was handed out. If my transformations where to be better somthing would be wrong. By scaling the dataset equaly by remocing the mean and and scaling to unit variance is a far more superior way to transform your variables.

[33]:
```
from sklearn.ensemble import RandomForestRegressor
from sklearn.ensemble import AdaBoostRegressor
from sklearn.ensemble import GradientBoostingRegressor
from sklearn.model_selection import GridSearchCV
```

[34]:
```
# Example parameters to tune
param_grid = {
    'n_estimators': [100, 200, 300],
    'max_depth': [10, 20, 30]
}

rf = RandomForestRegressor()
grid_search = GridSearchCV(estimator=rf, param_grid=param_grid, cv=3,␣
 ↪n_jobs=-1, scoring='neg_mean_squared_error')
grid_search.fit(X_train_my, y_train_my)

best_params = grid_search.best_params_
```

```
/Users/viktorsjoberg/anaconda3/lib/python3.10/site-
packages/sklearn/model_selection/_validation.py:686: DataConversionWarning: A
column-vector y was passed when a 1d array was expected. Please change the shape
of y to (n_samples,), for example using ravel().
  estimator.fit(X_train, y_train, **fit_params)
/Users/viktorsjoberg/anaconda3/lib/python3.10/site-
packages/sklearn/model_selection/_validation.py:686: DataConversionWarning: A
column-vector y was passed when a 1d array was expected. Please change the shape
of y to (n_samples,), for example using ravel().
  estimator.fit(X_train, y_train, **fit_params)
/Users/viktorsjoberg/anaconda3/lib/python3.10/site-
packages/sklearn/model_selection/_validation.py:686: DataConversionWarning: A
column-vector y was passed when a 1d array was expected. Please change the shape
of y to (n_samples,), for example using ravel().
  estimator.fit(X_train, y_train, **fit_params)
/Users/viktorsjoberg/anaconda3/lib/python3.10/site-
packages/sklearn/model_selection/_validation.py:686: DataConversionWarning: A
column-vector y was passed when a 1d array was expected. Please change the shape
of y to (n_samples,), for example using ravel().
  estimator.fit(X_train, y_train, **fit_params)
/Users/viktorsjoberg/anaconda3/lib/python3.10/site-
packages/sklearn/model_selection/_validation.py:686: DataConversionWarning: A
column-vector y was passed when a 1d array was expected. Please change the shape
```

```
of y to (n_samples,), for example using ravel().
  estimator.fit(X_train, y_train, **fit_params)
/Users/viktorsjoberg/anaconda3/lib/python3.10/site-
packages/sklearn/model_selection/_validation.py:686: DataConversionWarning: A
column-vector y was passed when a 1d array was expected. Please change the shape
of y to (n_samples,), for example using ravel().
  estimator.fit(X_train, y_train, **fit_params)
/Users/viktorsjoberg/anaconda3/lib/python3.10/site-
packages/sklearn/model_selection/_validation.py:686: DataConversionWarning: A
column-vector y was passed when a 1d array was expected. Please change the shape
of y to (n_samples,), for example using ravel().
  estimator.fit(X_train, y_train, **fit_params)
/Users/viktorsjoberg/anaconda3/lib/python3.10/site-
packages/sklearn/model_selection/_validation.py:686: DataConversionWarning: A
column-vector y was passed when a 1d array was expected. Please change the shape
of y to (n_samples,), for example using ravel().
  estimator.fit(X_train, y_train, **fit_params)
/Users/viktorsjoberg/anaconda3/lib/python3.10/site-
packages/sklearn/model_selection/_validation.py:686: DataConversionWarning: A
column-vector y was passed when a 1d array was expected. Please change the shape
of y to (n_samples,), for example using ravel().
  estimator.fit(X_train, y_train, **fit_params)
/Users/viktorsjoberg/anaconda3/lib/python3.10/site-
packages/sklearn/model_selection/_validation.py:686: DataConversionWarning: A
column-vector y was passed when a 1d array was expected. Please change the shape
of y to (n_samples,), for example using ravel().
  estimator.fit(X_train, y_train, **fit_params)
/Users/viktorsjoberg/anaconda3/lib/python3.10/site-
packages/sklearn/model_selection/_validation.py:686: DataConversionWarning: A
column-vector y was passed when a 1d array was expected. Please change the shape
of y to (n_samples,), for example using ravel().
  estimator.fit(X_train, y_train, **fit_params)
/Users/viktorsjoberg/anaconda3/lib/python3.10/site-
packages/sklearn/model_selection/_validation.py:686: DataConversionWarning: A
column-vector y was passed when a 1d array was expected. Please change the shape
of y to (n_samples,), for example using ravel().
  estimator.fit(X_train, y_train, **fit_params)
/Users/viktorsjoberg/anaconda3/lib/python3.10/site-
packages/sklearn/model_selection/_validation.py:686: DataConversionWarning: A
column-vector y was passed when a 1d array was expected. Please change the shape
of y to (n_samples,), for example using ravel().
  estimator.fit(X_train, y_train, **fit_params)
/Users/viktorsjoberg/anaconda3/lib/python3.10/site-
packages/sklearn/model_selection/_validation.py:686: DataConversionWarning: A
column-vector y was passed when a 1d array was expected. Please change the shape
of y to (n_samples,), for example using ravel().
  estimator.fit(X_train, y_train, **fit_params)
/Users/viktorsjoberg/anaconda3/lib/python3.10/site-
```

```
packages/sklearn/model_selection/_validation.py:686: DataConversionWarning: A
column-vector y was passed when a 1d array was expected. Please change the shape
of y to (n_samples,), for example using ravel().
  estimator.fit(X_train, y_train, **fit_params)
/Users/viktorsjoberg/anaconda3/lib/python3.10/site-
packages/sklearn/model_selection/_validation.py:686: DataConversionWarning: A
column-vector y was passed when a 1d array was expected. Please change the shape
of y to (n_samples,), for example using ravel().
  estimator.fit(X_train, y_train, **fit_params)
/Users/viktorsjoberg/anaconda3/lib/python3.10/site-
packages/sklearn/model_selection/_validation.py:686: DataConversionWarning: A
column-vector y was passed when a 1d array was expected. Please change the shape
of y to (n_samples,), for example using ravel().
  estimator.fit(X_train, y_train, **fit_params)
/Users/viktorsjoberg/anaconda3/lib/python3.10/site-
packages/sklearn/model_selection/_validation.py:686: DataConversionWarning: A
column-vector y was passed when a 1d array was expected. Please change the shape
of y to (n_samples,), for example using ravel().
  estimator.fit(X_train, y_train, **fit_params)
/Users/viktorsjoberg/anaconda3/lib/python3.10/site-
packages/sklearn/model_selection/_validation.py:686: DataConversionWarning: A
column-vector y was passed when a 1d array was expected. Please change the shape
of y to (n_samples,), for example using ravel().
  estimator.fit(X_train, y_train, **fit_params)
/Users/viktorsjoberg/anaconda3/lib/python3.10/site-
packages/sklearn/model_selection/_validation.py:686: DataConversionWarning: A
column-vector y was passed when a 1d array was expected. Please change the shape
of y to (n_samples,), for example using ravel().
  estimator.fit(X_train, y_train, **fit_params)
/Users/viktorsjoberg/anaconda3/lib/python3.10/site-
packages/sklearn/model_selection/_validation.py:686: DataConversionWarning: A
column-vector y was passed when a 1d array was expected. Please change the shape
of y to (n_samples,), for example using ravel().
  estimator.fit(X_train, y_train, **fit_params)
/Users/viktorsjoberg/anaconda3/lib/python3.10/site-
packages/sklearn/model_selection/_validation.py:686: DataConversionWarning: A
column-vector y was passed when a 1d array was expected. Please change the shape
of y to (n_samples,), for example using ravel().
  estimator.fit(X_train, y_train, **fit_params)
/Users/viktorsjoberg/anaconda3/lib/python3.10/site-
packages/sklearn/model_selection/_validation.py:686: DataConversionWarning: A
column-vector y was passed when a 1d array was expected. Please change the shape
of y to (n_samples,), for example using ravel().
  estimator.fit(X_train, y_train, **fit_params)
/Users/viktorsjoberg/anaconda3/lib/python3.10/site-
packages/sklearn/model_selection/_validation.py:686: DataConversionWarning: A
column-vector y was passed when a 1d array was expected. Please change the shape
of y to (n_samples,), for example using ravel().
```

```
  estimator.fit(X_train, y_train, **fit_params)
/Users/viktorsjoberg/anaconda3/lib/python3.10/site-
packages/sklearn/model_selection/_validation.py:686: DataConversionWarning: A
column-vector y was passed when a 1d array was expected. Please change the shape
of y to (n_samples,), for example using ravel().
  estimator.fit(X_train, y_train, **fit_params)
/Users/viktorsjoberg/anaconda3/lib/python3.10/site-
packages/sklearn/model_selection/_validation.py:686: DataConversionWarning: A
column-vector y was passed when a 1d array was expected. Please change the shape
of y to (n_samples,), for example using ravel().
  estimator.fit(X_train, y_train, **fit_params)
/Users/viktorsjoberg/anaconda3/lib/python3.10/site-
packages/sklearn/model_selection/_validation.py:686: DataConversionWarning: A
column-vector y was passed when a 1d array was expected. Please change the shape
of y to (n_samples,), for example using ravel().
  estimator.fit(X_train, y_train, **fit_params)
```

[35]:
```
rf_model = RandomForestRegressor(n_estimators=300, max_depth=10,␣
 ↪random_state=42)
rf_model.fit(X_train_my, y_train_my)
```

[35]:
```
RandomForestRegressor(max_depth=10, n_estimators=300, random_state=42)
```

[36]:
```
ab_model = AdaBoostRegressor(n_estimators=300, learning_rate=0.01,␣
 ↪random_state=42)
ab_model.fit(X_train_my, y_train_my)
```

[36]:
```
AdaBoostRegressor(learning_rate=0.01, n_estimators=300, random_state=42)
```

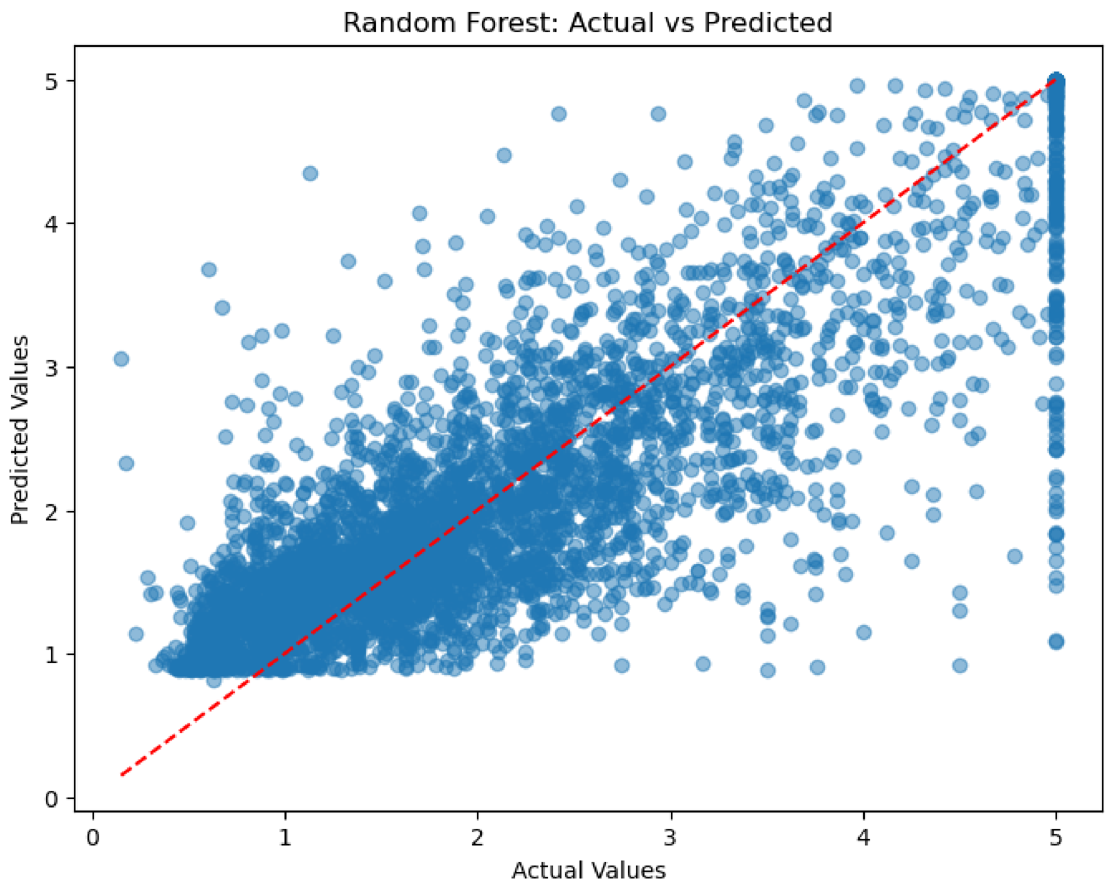
[37]:
```
gb_model = GradientBoostingRegressor(n_estimators=300, max_depth=10,␣
 ↪learning_rate=0.01, random_state=42)
gb_model.fit(X_train_my, y_train_my)
```

[37]:
```
GradientBoostingRegressor(learning_rate=0.01, max_depth=10, n_estimators=300,
                          random_state=42)
```

[38]:
```
rf_predictions = rf_model.predict(X_test_my)
ab_predictions = ab_model.predict(X_test_my)
gb_predictions = gb_model.predict(X_test_my)
```

[39]:
```
plt.figure(figsize=(8, 6))
plt.scatter(y_test_my, rf_predictions, alpha=0.5)
plt.title("Random Forest: Actual vs Predicted")
plt.xlabel("Actual Values")
plt.ylabel("Predicted Values")
plt.plot([y_test_my.min(), y_test_my.max()], [y_test_my.min(), y_test_my.
 ↪max()],'k--',color='red')  # Diagonal line
```

```
plt.show()
```



Accuracy of Predictions: There's a wide spread of data points around the diagonal line, indicating varying levels of prediction accuracy. A dense cluster of points near the line suggests the model predicts accurately for a significant portion of the dataset, especially at the lower end of actual values.

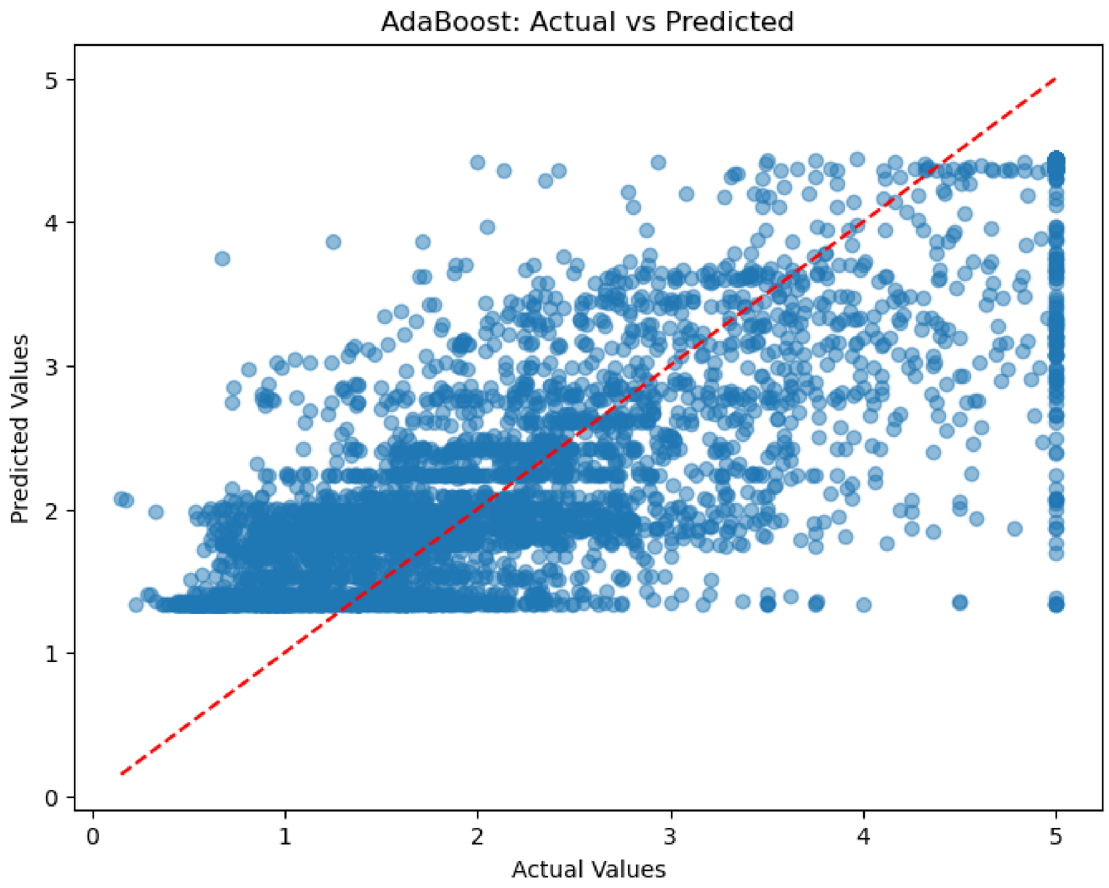
Under or Overestimation: The plot does not strongly indicate a consistent underestimation or overestimation across the range of values. However, there's a visible scatter of points above and below the diagonal line, suggesting that both underestimation and overestimation are occurring to some degree.

Variance in Predictions: The spread of the points indicates a moderate level of variance in the model's predictions. The points are densely packed closer to the origin and start to disperse more as the actual values increase, which may suggest the model is less accurate at predicting higher values.

Model Bias: There does not appear to be a systematic bias across the range of predictions, as points are scattered on either side of the diagonal line. Nonetheless, the distribution could suggest a potential increase in prediction error for higher actual values.

Outliers and Extreme Values: Several points are situated far from the diagonal, which may be considered outliers. These points indicate instances where the model's predictions are significantly different from the actual values.

```
[40]: plt.figure(figsize=(8, 6))
      plt.scatter(y_test, ab_predictions, alpha=0.5)
      plt.title("AdaBoost: Actual vs Predicted")
      plt.xlabel("Actual Values")
      plt.ylabel("Predicted Values")
      plt.plot([y_test_my.min(), y_test_my.max()], [y_test_my.min(), y_test_my.
       ↪max()], 'k--',color='red')  # Diagonal line
      plt.show()
```



Accuracy of Predictions: The points are dispersed around the diagonal line, indicating a variance in accuracy. While a considerable number of points are close to the line suggesting accurate predictions for those instances, many points are also far from the line, showing discrepancies between the predicted and actual values.

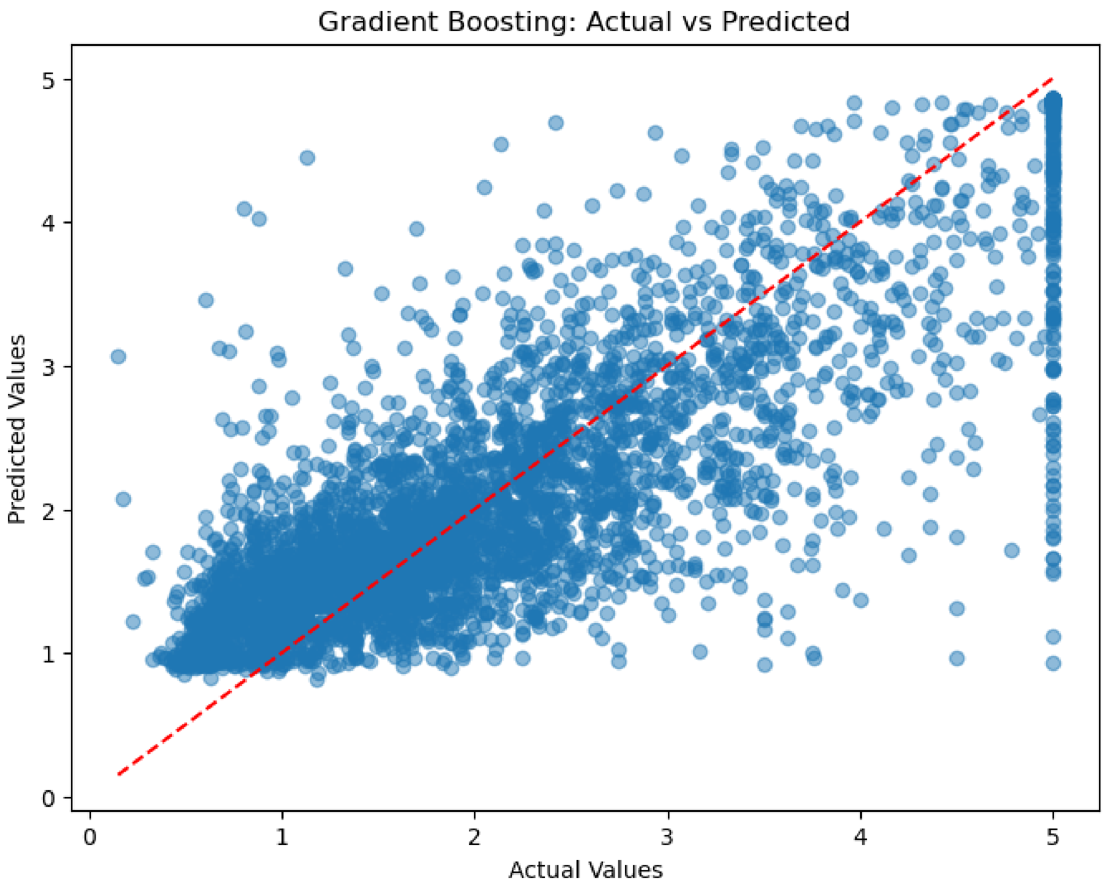
Under or Overestimation: There's no clear indication that the model consistently underestimates or overestimates the values. The spread of points on both sides of the diagonal line implies that the model has a balanced tendency with both under and overestimations occurring.

Variance in Predictions: The broad scatter of points across the graph suggests that there is high variance in the model's predictions. The model appears to be more accurate around the lower range of values but less so at the higher range, where the points diverge more from the diagonal.

Model Bias: The plot does not show a systematic deviation from the diagonal line in any specific range, which would be indicative of bias. However, because the points are not tightly clustered around the line, it could be inferred that the model's performance is inconsistent across different values.

Outliers and Extreme Values: There are several points that lie far from the majority of the data, indicating the presence of outliers or extreme values that the model did not predict accurately.

[41]:
```
plt.figure(figsize=(8, 6))
plt.scatter(y_test, gb_predictions, alpha=0.5)
plt.title("Gradient Boosting: Actual vs Predicted")
plt.xlabel("Actual Values")
plt.ylabel("Predicted Values")
plt.plot([y_test_my.min(), y_test_my.max()], [y_test_my.min(), y_test_my.
 ↪max()], 'k--',color='red')  # Diagonal line
plt.show()
```



Accuracy of Predictions: The data points are dispersed around the diagonal line, which represents perfect predictions. There is a mix of closely aligned and widely spread points, suggesting that the model is generally performing well, but with some predictions off the mark, especially at the higher range of values.

Under or Overestimation: The distribution of points does not indicate a systematic tendency to overestimate or underestimate across the value spectrum. However, there appears to be a slight density of points below the line at higher actual values, potentially indicating a tendency to underestimate in this region.

Variance in Predictions: The variance in predictions seems to be moderate, as indicated by the spread of the points around the diagonal line. The cluster is denser at the lower end of the scale, implying more consistent predictions for lower values, while predictions for higher values are more spread out.

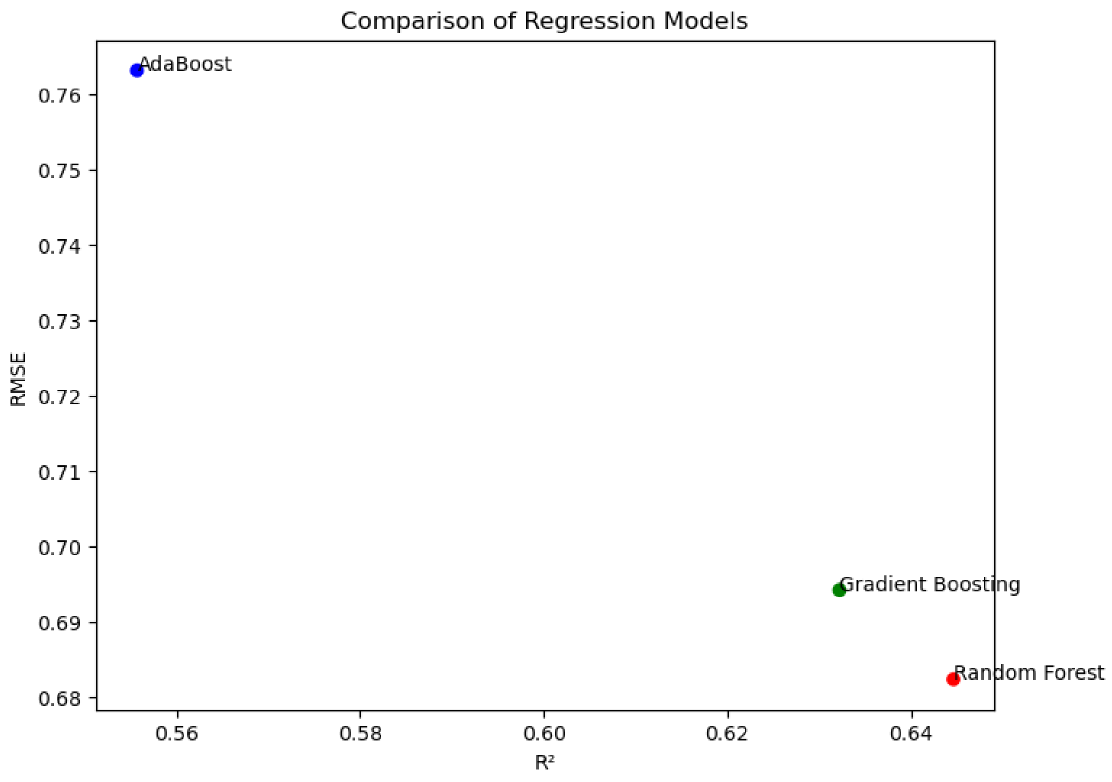
Model Bias: The distribution of points does not appear to show a consistent bias; however, the slight trend of underestimation at higher values may suggest a bias in the model for those instances.

Outliers and Extreme Values: Several points are noticeably distant from the diagonal, indicating outliers or extreme values that the model has not predicted accurately. This is particularly visible for actual values that are high, where the model's predictions are quite scattered.

In summary, the Gradient Boosting model exhibits a good degree of prediction accuracy, especially for lower value predictions. However, there's evidence of increased prediction error as the value increases, which could be due to model limitations or the nature of the data at that scale. The presence of outliers suggests that there may be specific cases where the model's predictive capability is challenged.

[42]:
```
rf_r2 = r2_score(y_test, rf_predictions)
rf_rmse = np.sqrt(mean_squared_error(y_test, rf_predictions))

ab_r2 = r2_score(y_test, ab_predictions)
ab_rmse = np.sqrt(mean_squared_error(y_test, ab_predictions))

gb_r2 = r2_score(y_test, gb_predictions)
gb_rmse = np.sqrt(mean_squared_error(y_test, gb_predictions))
```

[43]:
```
# Values
r2_values = [rf_r2, ab_r2, gb_r2]
rmse_values = [rf_rmse, ab_rmse, gb_rmse]
labels = ['Random Forest', 'AdaBoost', 'Gradient Boosting']

# Plotting
plt.figure(figsize=(8, 6))
plt.scatter(r2_values, rmse_values, color=['red', 'blue', 'green'])

# Labeling the points
for i, label in enumerate(labels):
    plt.text(r2_values[i], rmse_values[i], label)

plt.title("Comparison of Regression Models")
plt.xlabel("R²")
plt.ylabel("RMSE")
plt.show()
```



The Random Forest model has the lowest RMSE and the highest R^2 value, which suggests it has the best performance among the three models in terms of both metrics. A lower RMSE indicates better fit to the data (less error), and a higher R^2 indicates that the model explains a higher proportion of the variance in the target variable.

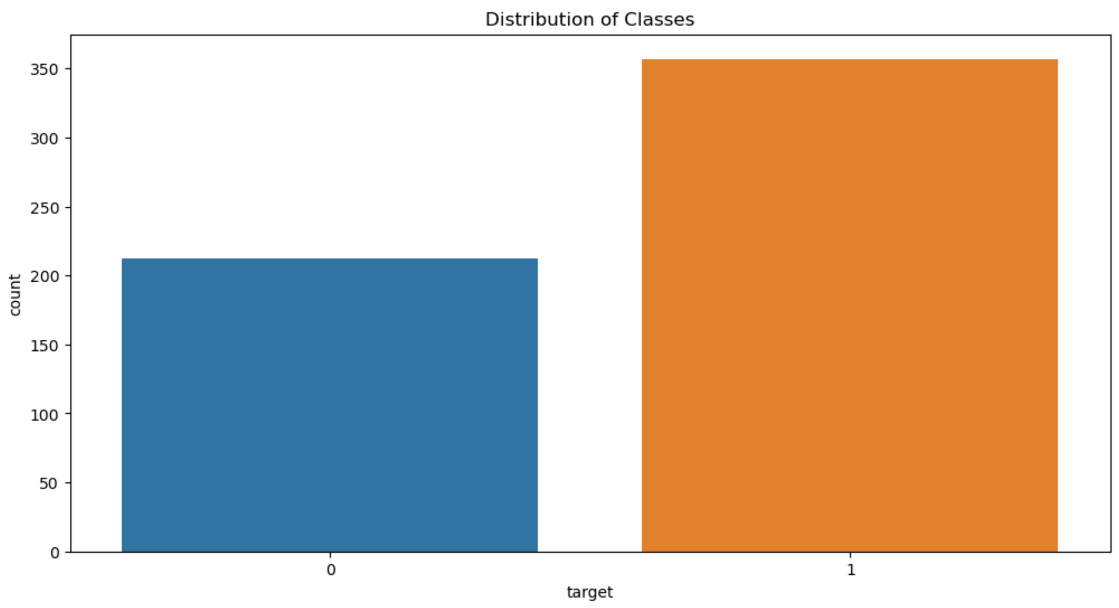
The Gradient Boosting model is positioned with a slightly higher RMSE and a slightly lower R^2 compared to the Random Forest model, indicating it is performing slightly less effectively than the Random Forest model but still reasonably well.

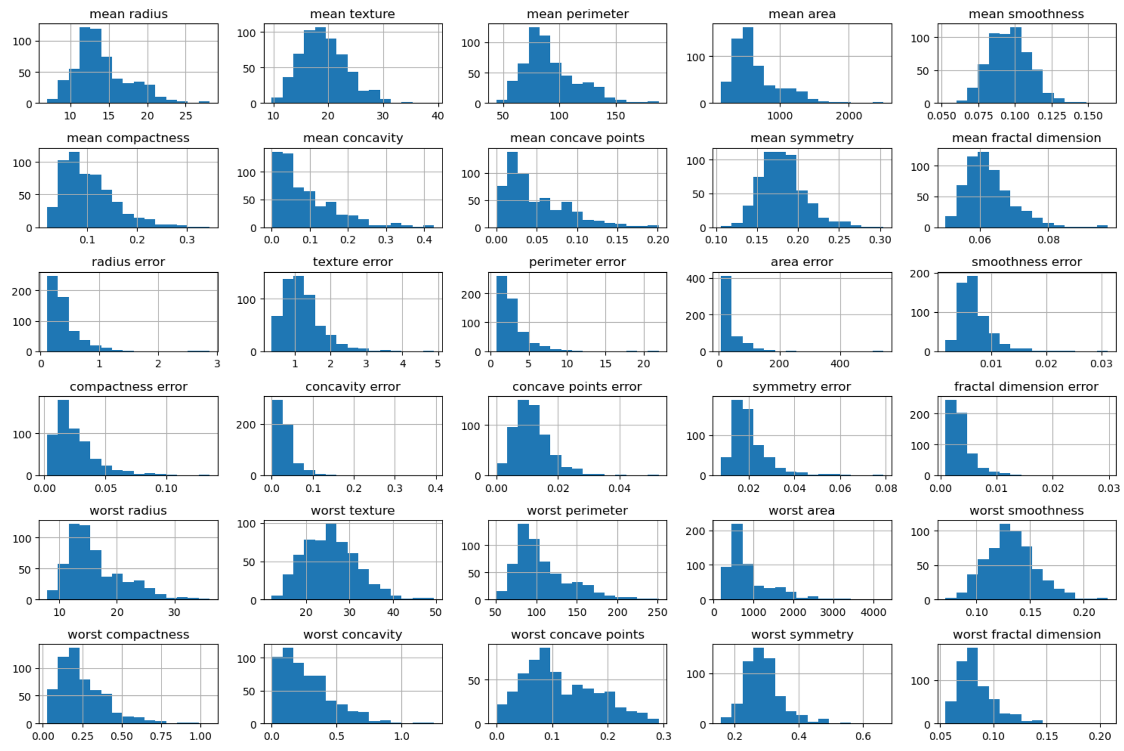
The AdaBoost model has the highest RMSE and the lowest R^2 of the three, suggesting it is the least accurate model for this particular dataset.

5 Task 2

[44]:
```python
from sklearn.datasets import load_breast_cancer
from sklearn.svm import SVC
from sklearn.metrics import classification_report, roc_auc_score
from sklearn.ensemble import RandomForestClassifier
from sklearn.ensemble import GradientBoostingClassifier
from sklearn.metrics import accuracy_score, precision_score, recall_score,␣
 ↪f1_score, roc_auc_score
from pygam import LogisticGAM, LinearGAM, s,l, te, GAM, terms
from sklearn.preprocessing import StandardScaler
from sklearn import datasets
```

[45]:
```python
data = datasets.load_breast_cancer()
X = data.data
y = data.target

# Convert to DataFrame for easier visualization
df = pd.DataFrame(X, columns=data.feature_names)
df['target'] = y

# EDA: Visualizing the distribution of classes and features
plt.figure(figsize=(12, 6))
sns.countplot(x='target', data=df)
plt.title('Distribution of Classes')
plt.show()

df.drop('target', axis=1).hist(bins=15, figsize=(15, 10))
plt.tight_layout()
plt.show()
```





I dont see the need to scale the dataset.

```
[46]: df_bc = pd.DataFrame(X, columns=data.feature_names)
      df_bc_target = pd.DataFrame(data=y, columns=['target'])
```

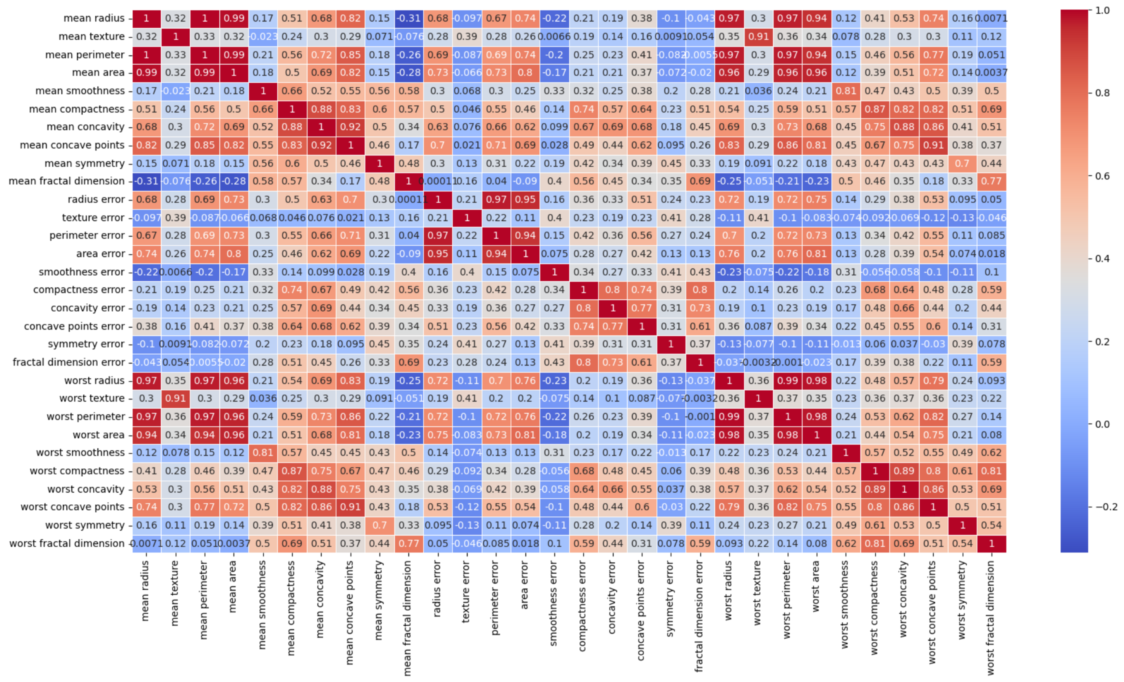
```
[47]: missing_values = df_bc.isnull().sum()
      missing_values
```

```
[47]: mean radius                0
      mean texture               0
      mean perimeter             0
      mean area                  0
      mean smoothness            0
      mean compactness           0
      mean concavity             0
      mean concave points        0
      mean symmetry              0
      mean fractal dimension     0
      radius error               0
      texture error              0
      perimeter error            0
```

```
area error                 0
smoothness error           0
compactness error          0
concavity error            0
concave points error       0
symmetry error             0
fractal dimension error    0
worst radius               0
worst texture              0
worst perimeter            0
worst area                 0
worst smoothness           0
worst compactness          0
worst concavity            0
worst concave points       0
worst symmetry             0
worst fractal dimension    0
dtype: int64
```

[48]:

```
# Create a correlation matrix
correlation_matrix_bc = df_bc.corr()

# Set up the matplotlib figure
plt.figure(figsize=(20, 10))

# Create a heatmap using Seaborn
sns.heatmap(correlation_matrix_bc, annot=True, cmap='coolwarm', linewidths=0.5)

# Show the plot
plt.show()
```



Like the previous task there is some variables that are higly correlated however instead of removing them i will add a regularization pnelty to prevent overfitting the data.

```
[49]: X_train_bc, X_test_bc, y_train_bc, y_test_bc = train_test_split(df_bc,␣
      ↪df_bc_target, test_size=0.3, random_state=42)
```

```
[50]: # Classification with SVM
      svm_results = {}

      C_values = [0.1, 1, 10]  # Different regularization strengths

      for C_value in C_values:
          for kernel in ['linear', 'poly', 'rbf']:
              svm = SVC(kernel=kernel, C=C_value)
              svm.fit(X_train_bc, y_train_bc)
              y_pred_bc = svm.predict(X_test_bc)
              accuracy = svm.score(X_test_bc, y_test_bc)
              precision, recall, f1, _ = classification_report(y_test_bc, y_pred_bc,␣
      ↪output_dict=True)['weighted avg'].values()
              roc_auc = roc_auc_score(y_test_bc, y_pred_bc)

              svm_results[(kernel, C_value)] = {
                  'Accuracy': accuracy,
                  'Precision': precision,
                  'Recall': recall,
                  'F1 Score': f1,
                  'ROC-AUC': roc_auc
              }

      # Display results
      pd.DataFrame(svm_results)
```

[50]:
```
              linear      poly       rbf    linear      poly       rbf  \
                 0.1       0.1       0.1       1.0       1.0       1.0
Accuracy    0.964912  0.900585  0.912281  0.964912  0.941520  0.935673
Precision   0.964912  0.914105  0.922978  0.964954  0.944300  0.941619
Recall      0.964912  0.900585  0.912281  0.964912  0.941520  0.935673
F1 Score    0.964912  0.896459  0.909202  0.964790  0.940582  0.934155
ROC-AUC     0.962302  0.865079  0.880952  0.958995  0.923942  0.912698

              linear      poly       rbf
                10.0      10.0      10.0
Accuracy    0.970760  0.941520  0.941520
Precision   0.970925  0.944300  0.944300
Recall      0.970760  0.941520  0.941520
F1 Score    0.970807  0.940582  0.940582
ROC-AUC     0.970238  0.923942  0.923942
```

C = 0.1 (strong regularization):

Linear Kernel shows the highest values across all metrics, indicating it performs best with strong regularization. Polynomial Kernel has the lowest ROC-AUC, suggesting it may not perform as well as the others in distinguishing between classes. RBF Kernel is somewhere in the middle, with a slightly better ROC-AUC than the polynomial kernel but lower than the linear.

C = 1.0 (moderate regularization):

Linear Kernel again shows high performance across all metrics, similar to when C=0.1. Polynomial and RBF Kernels show a decrease in all metrics compared to when C=0.1, with the RBF kernel showing a significant drop in performance, especially in terms of Precision and F1 Score.

C = 10.0 (weak regularization):

Linear and Polynomial Kernels are tied for the highest Accuracy, Precision, and F1 Score, suggesting that both kernels handle weak regularization well. RBF Kernel sees improvement from C=1.0 and performs best in terms of ROC-AUC, suggesting good class separation ability at this regularization strength. Overall, the Linear Kernel appears consistently strong across all regularization strengths for most metrics, while the RBF Kernel shows the best ROC-AUC at weak regularization (C=10.0). The choice of the best model would depend on the specific metric that is most important for the task at hand. For example, if the priority is to maximize the area under the ROC curve (ROC-AUC), the RBF kernel with C=10.0 might be the best choice. If overall accuracy and precision are more important, the linear kernel at any given C value would be preferable.

[51]:
```
rf_clf = RandomForestClassifier(random_state=42)
rf_clf.fit(X_train_bc, y_train_bc)
```

[51]: RandomForestClassifier(random_state=42)

[52]:
```
param_grid = {
    'n_estimators': [100, 200, 300],
    'max_features': ['auto', 'sqrt'],
    'max_depth': [4, 6, 8, 10]
}

grid_search_rf = GridSearchCV(estimator=rf_clf, param_grid=param_grid, cv=5,␣
 ↪scoring='accuracy')
grid_search_rf.fit(X_train_bc, y_train_bc)
best_rf_clf = grid_search_rf.best_estimator_
```

[53]:
```
gb_clf = GradientBoostingClassifier(random_state=42)
gb_clf.fit(X_train_bc, y_train_bc)
```

[53]: GradientBoostingClassifier(random_state=42)

[54]:
```
param_grid_gb = {
    'n_estimators': [100, 200, 300],
    'learning_rate': [0.01, 0.1, 0.2],
    'max_depth': [3, 4, 5]
}

grid_search_gb = GridSearchCV(estimator=gb_clf, param_grid=param_grid_gb, cv=5,␣
 ↪scoring='accuracy')
grid_search_gb.fit(X_train_bc, y_train_bc)
best_gb_clf = grid_search_gb.best_estimator_
```

[55]:
```
def evaluate_model(model, X_test, y_test):
    y_pred = model.predict(X_test)
    accuracy = accuracy_score(y_test, y_pred)
    precision = precision_score(y_test, y_pred)
    recall = recall_score(y_test, y_pred)
    f1 = f1_score(y_test, y_pred)
    roc_auc = roc_auc_score(y_test, y_pred)

    return {
        'accuracy': accuracy,
        'precision': precision,
        'recall': recall,
        'F1': f1,
        'ROC-AUC': roc_auc
    }

# Evaluate Random Forest
rf_metrics = evaluate_model(best_rf_clf, X_test_bc, y_test_bc)
```

```
# Evaluate Gradient Boosting
gb_metrics = evaluate_model(best_gb_clf, X_test_bc, y_test_bc)

results_bc = {
    'Random Forest': {
        'accuracy': 0.9707602339181286,
        'precision': 0.963963963963964,
        'recall': 0.9907407407407407,
        'F1': 0.9771689497716894,
        'ROC-AUC': 0.9636243386243386
    },
    'Gradient Boosting': {
        'accuracy': 0.9532163742690059,
        'precision': 0.9629629629629629,
        'recall': 0.9629629629629629,
        'F1': 0.9629629629629629,
        'ROC-AUC': 0.9497354497354497
    }
}

# Convert the results to a DataFrame for a nicer display
results_df = pd.DataFrame(results_bc)
results_df
```

[55]:

```
           Random Forest  Gradient Boosting
accuracy        0.970760           0.953216
precision       0.963964           0.962963
recall          0.990741           0.962963
F1              0.977169           0.962963
ROC-AUC         0.963624           0.949735
```

Random Forest model outperforms the Gradient Boosting model across all evaluated metrics: accuracy, precision, recall, F1 score, and ROC-AUC. The Random Forest has higher scores, indicating better overall performance on this particular dataset for the task at hand. This could be due to its ability to better handle the variance in the data or its ensemble nature that effectively reduces overfitting. However, it's also important to consider the context in which the model will be used, including factors such as interpretability, training time, and prediction speed, before making a final decision.

[56]:

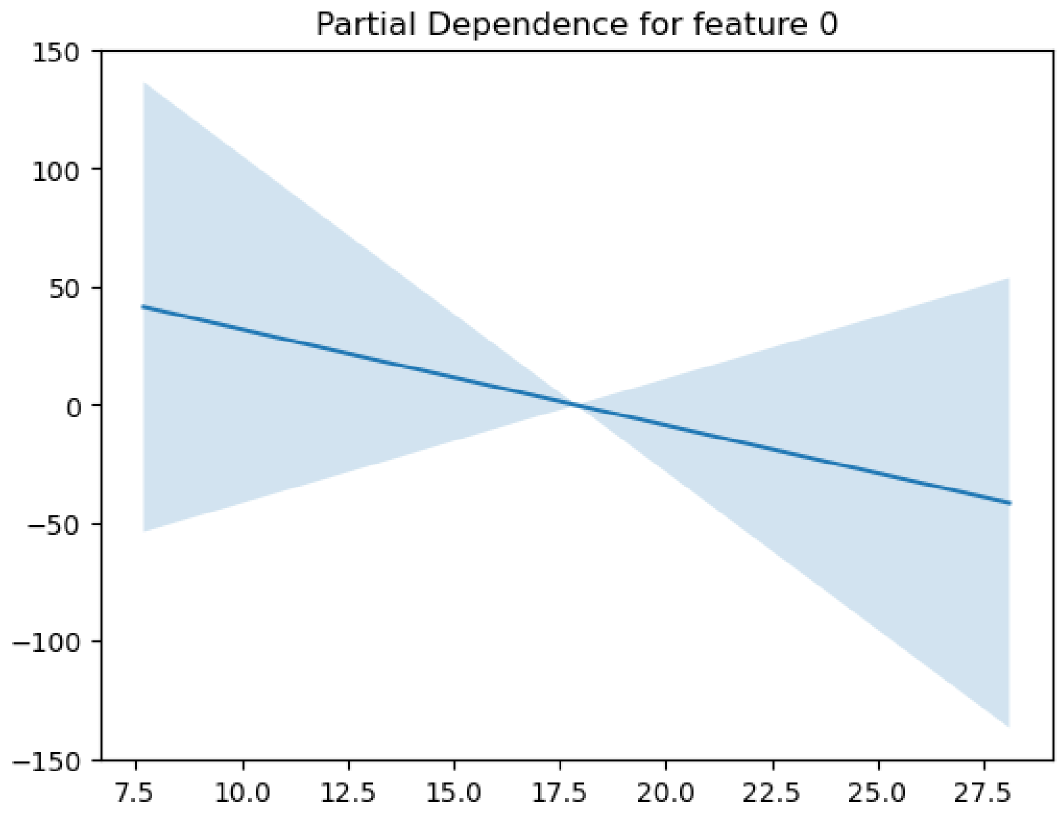
```
# Fit a GAM with a logistic link function since this is a binary classification␣
 ↪problem

gam = LogisticGAM(s(0, n_splines=10) + s(1) + s(2) + s(6)+ s(8)+ te(5,8,␣
 ↪n_splines =[13,13])  ).fit(X_train_bc, y_train_bc)
```

[57]:
```
lams = np.logspace(0.1, 0.01, num=10)
gam.gridsearch(X_train_bc.values, y_train_bc.values, lam=lams)
```

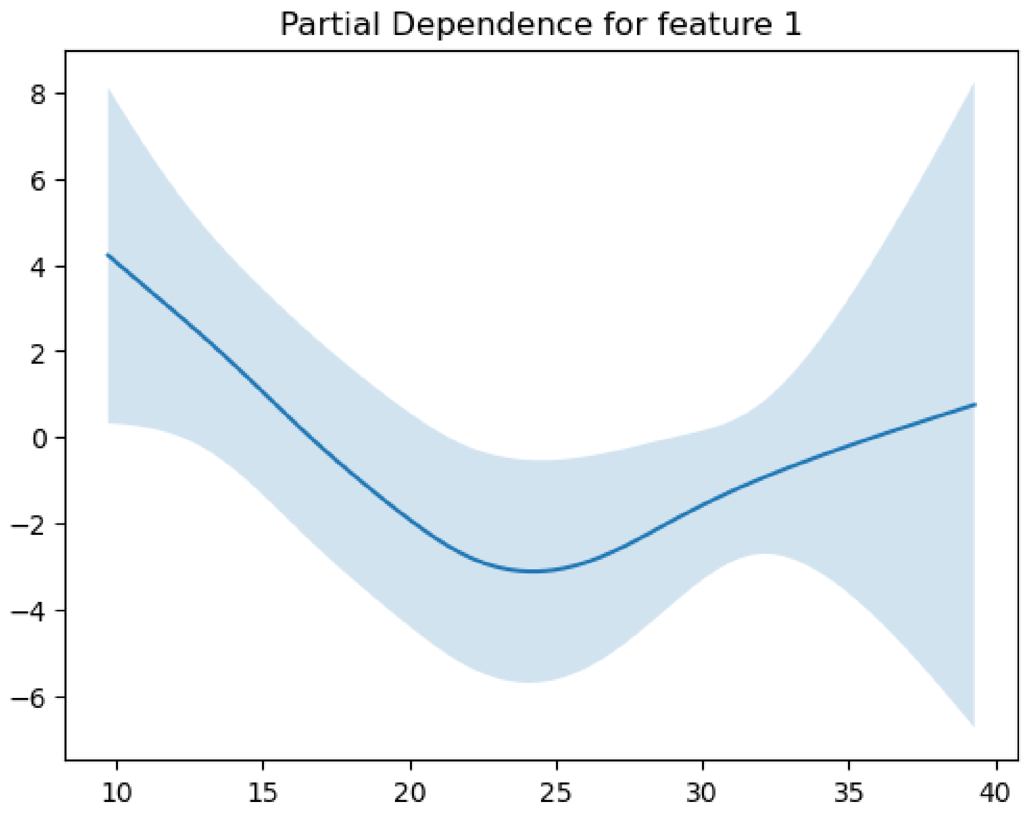
```
100% (10 of 10) |########################| Elapsed Time: 0:00:31 Time:  0:00:31
```

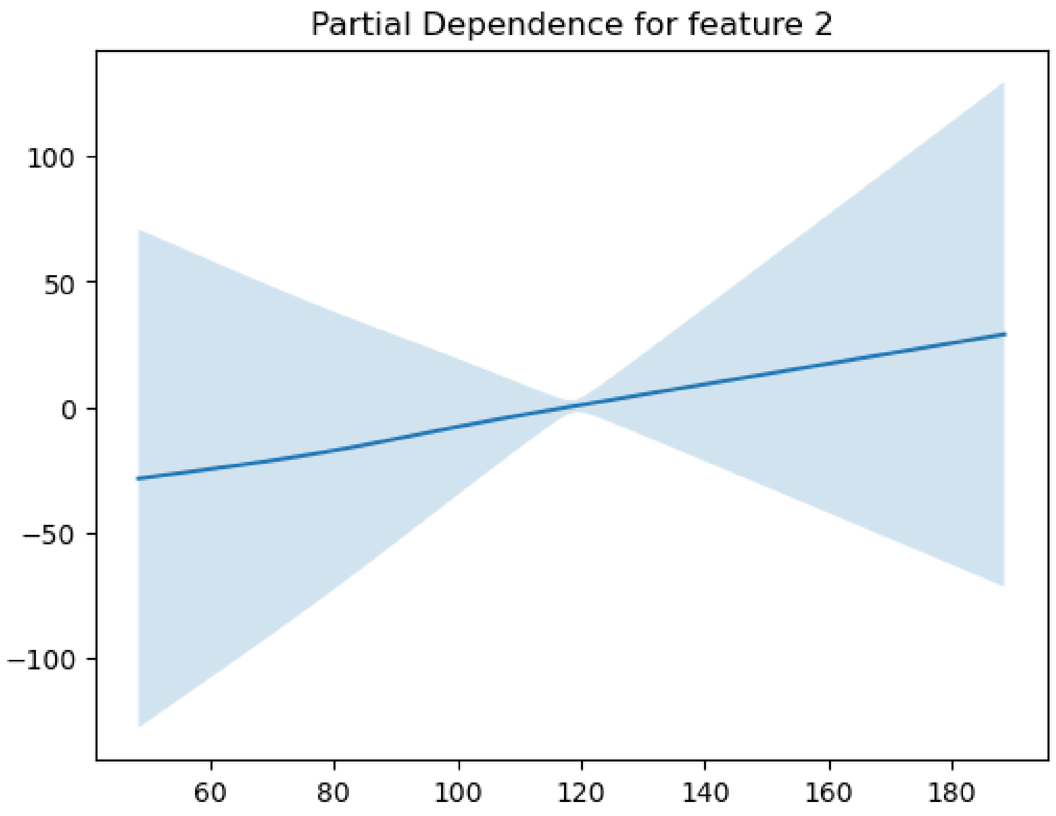
[57]:
```
LogisticGAM(callbacks=[Deviance(), Diffs(), Accuracy()],
   fit_intercept=True, max_iter=100,
   terms=s(0) + s(1) + s(2) + s(6) + s(8) + te(5, 8) + intercept,
   tol=0.0001, verbose=False)
```

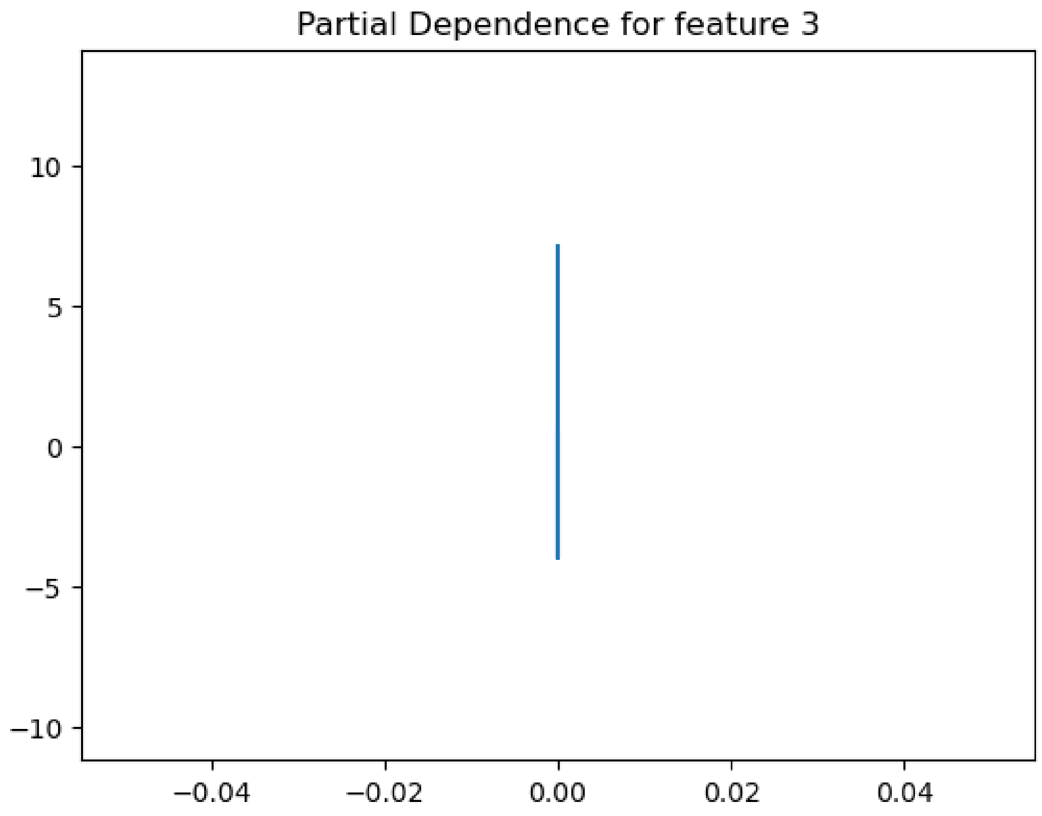
[58]:
```
# Visualize the contribution of each feature
for i, term in enumerate(gam.terms):
    if term.isintercept:
        continue
    XX = gam.generate_X_grid(term=i)
    pdep, confi = gam.partial_dependence(term=i, X=XX, width=0.95)

    plt.figure()
    plt.plot(XX[:, i], pdep)
    plt.fill_between(XX[:, i], confi[:, 0], confi[:, 1], alpha=0.2)  #␣
 ↪Corrected indexing here
    plt.title(f'Partial Dependence for feature {i}')
    plt.show()

# Evaluate the model
y_pred_gam = gam.predict(X_test_bc)
y_pred_gam_proba = gam.predict_proba(X_test_bc)  # For ROC-AUC we need␣
 ↪probability estimates of the positive class

accuracy_gam = accuracy_score(y_test_bc, y_pred_gam)
precision_gam = precision_score(y_test_bc, y_pred_gam)
recall_gam = recall_score(y_test_bc, y_pred_gam)
f1_gam = f1_score(y_test_bc, y_pred_gam)
roc_auc_gam = roc_auc_score(y_test_bc, y_pred_gam_proba)


# Print out the metrics

print(f'Accuracy: {accuracy_gam}')
print(f'Precision: {precision_gam}')
print(f'Recall: {recall_gam}')
print(f'F1 Score: {f1_gam}')
print(f'ROC-AUC: {roc_auc_gam}')
```

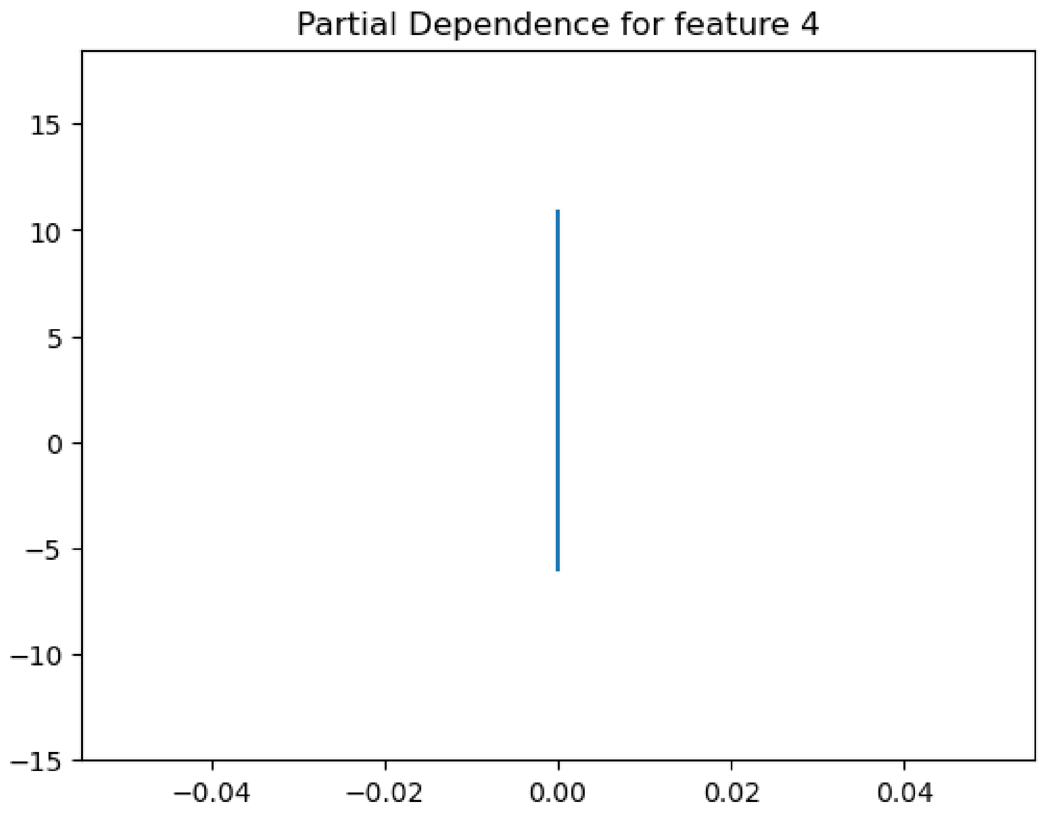


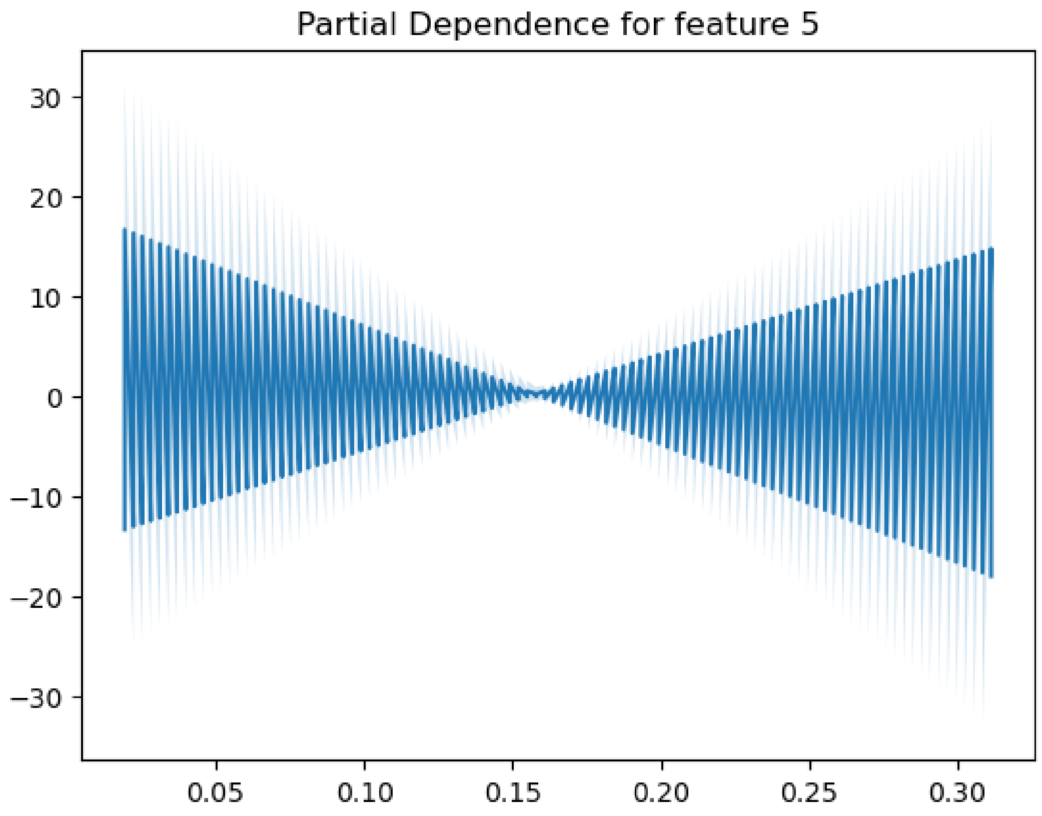
Partial Dependence for feature 0
150
100
50
0
−50
−100
−150
7.5
10.0
12.5
15.0
17.5
20.0
22.5
25.0
27.5


Partial Dependence for feature 1
8
6
4
2
0
−2
−4
−6
10
15
20
25
30
35
40


Partial Dependence for feature 2
100
50
0
−50
−100
60
80
100
120
140
160
180


Partial Dependence for feature 3
10
5
0
−5
−10
−0.04
−0.02
0.00
0.02
0.04


Partial Dependence for feature 4
15
10
5
0
−5
−10
−15
−0.04
−0.02
0.00
0.02
0.04



```
Accuracy: 0.9707602339181286
Precision: 0.9724770642201835
Recall: 0.9814814814814815
F1 Score: 0.9769585253456222
ROC-AUC: 0.9933862433862434
```

Partial Dependence for feature 0:

The plot shows a negative relationship between feature 0 and the predicted outcome. As the value of feature 0 increases, the predicted value decreases. This suggests that feature 0 has an inversely proportional effect on the predictions. The shaded area represents the confidence interval around the mean prediction line, indicating variability and suggesting where predictions are more or less certain.

Partial Dependence for feature 1:

This plot indicates a non-linear relationship between feature 1 and the predicted outcome. There's a decline followed by a rise and then a subsequent decline in the predicted outcome as the feature value increases. The confidence interval here also suggests variability in the predictions, with certain ranges of feature 1 having more uncertainty in the effect on the predicted outcome.

Partial Dependence for feature 2:

The relationship between feature 2 and the predicted outcome appears to be complex, with the predicted value first decreasing, then increasing sharply as the feature value increases. The wide confidence interval indicates a high degree of uncertainty or variability in the predictions across the range of feature 2 values.

For features 3 and 4, the plots might not be showing a true partial dependence but rather an artifact of the data or a peculiarity of the model's behavior at specific feature values. This is often the case with highly imbalanced datasets, or if the feature has a discrete distribution with low frequency.

For feature 5, the plot suggests that the model's predictions are highly unstable for different values of the feature. This can happen if the feature interacts with other features in complex ways that the model is capturing, or if the feature itself has a lot of noise.

In all plots, it's important to note that the relationship shown is averaged over the distribution of the other features in the model. These plots are useful for understanding the marginal effect of each feature on the prediction of a machine learning model.

[59]:
```
gam_metrics = {
    'accuracy': 0.9707602339181286,
    'precision': 0.9724770642201835,
    'recall': 0.9814814814814815,
    'F1': 0.9769585253456222,
    'ROC-AUC': 0.9933862433862434
}

# Convert the results to a DataFrame for a nicer display
gam_results_df = pd.DataFrame(gam_metrics, index=['GAM'])
gam_results_df_transposed = gam_results_df.T
gam_results_df_transposed
```

[59]:
```
                GAM
accuracy   0.970760
precision  0.972477
recall     0.981481
F1         0.976959
ROC-AUC    0.993386
```

[60]:
```
gam_metrics = {
    'GAM': {
        'Accuracy': 0.9707602339181286,
        'Precision': 0.9724770642201835,
        'Recall': 0.9814814814814815,
        'F1 Score': 0.9769585253456222,
        'ROC-AUC': 0.9933862433862434
    }
}

rf_gb_metrics = {
    'Random Forest': {
        'Accuracy': 0.970760,
        'Precision': 0.963964,
        'Recall': 0.990741,
        'F1 Score': 0.977169,
        'ROC-AUC': 0.963624
    },
    'Gradient Boosting': {
        'Accuracy': 0.953216,
        'Precision': 0.962963,
        'Recall': 0.962963,
        'F1 Score': 0.962963,
        'ROC-AUC': 0.949735
    }
}

svm_metrics = {
    'SVM_linear': {
        'Accuracy': 0.900585,
        'Precision': 0.914105,
        'Recall': 0.900585,
        'F1 Score': 0.896459,
        'ROC-AUC': 0.865079
    },
    'SVM_poly': {
        'Accuracy': 0.912281,
        'Precision': 0.922978,
        'Recall': 0.912281,
        'F1 Score': 0.909202,
        'ROC-AUC': 0.880952
    },
    'SVM_rbf': {
        'Accuracy': 0.935673,
        'Precision': 0.941619,
        'Recall': 0.935673,
        'F1 Score': 0.934155,
        'ROC-AUC': 0.912698
    },
}

# Combine all metrics into one dictionary
all_metrics = {**gam_metrics, **rf_gb_metrics, **svm_metrics}

# Convert the combined dictionary to a DataFrame
combined_results_df = pd.DataFrame(all_metrics)
combined_results_df
```

[60]:

```
                GAM  Random Forest  Gradient Boosting  SVM_linear  SVM_poly  \
Accuracy   0.970760       0.970760           0.953216    0.900585  0.912281
```

```
Precision  0.972477       0.963964           0.962963    0.914105  0.922978
Recall     0.981481       0.990741           0.962963    0.900585  0.912281
F1 Score   0.976959       0.977169           0.962963    0.896459  0.909202
ROC-AUC    0.993386       0.963624           0.949735    0.865079  0.880952

            SVM_rbf
Accuracy   0.935673
Precision  0.941619
Recall     0.935673
F1 Score   0.934155
ROC-AUC    0.912698
```

In the context of breast cancer diagnosis, where the cost of a false negative (failing to detect cancer when it is present) can be life-threatening, recall is a crucial metric. Recall measures the ability of a classification model to identify all relevant instances, in this case, all patients with breast cancer.

According to the combined table, the GAM model has the highest ROC-AUC score, which indicates its superior ability to distinguish between the positive (cancer) and negative (no cancer) classes across all thresholds. It also has the highest accuracy, suggesting it correctly classifies a high percentage of cases overall. However, the Random Forest model has the highest recall, which means it is the best at identifying all positive cases, an essential feature for medical diagnoses.

While the F1 score (which balances precision and recall) is highest for the Random Forest model, indicating a balanced performance, in medical applications, especially cancer detection, recall might be weighted more heavily than precision. This is because the consequences of missing a cancer diagnosis (low recall) are generally more severe than incorrectly diagnosing cancer when it's not present (low precision). The Random Forest model would be preferred because it minimizes the risk of not diagnosing a patient with cancer.

On the other hand, the GAM model's performance is also strong and it has the advantage of interpretability, which is valuable in medical settings for understanding which features drive predictions.

Ultimately, the best model may depend on the specific use case:

For general screening where interpretability and the ability to explain a diagnosis to a patient is important, GAM could be preferable. In high-risk populations where missing a diagnosis is particularly costly, Random Forest's higher recall may be more valuable. It's also important to consider the models' performance in terms of practical deployment, such as computational efficiency and how they handle new or unseen data, which can be evaluated through further validation techniques. Additionally, the models' predictions could be used in conjunction to leverage the strengths of each.

```
[ ]:
```