Assignment 1

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.

Dataset

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

Tasks

1. Data Exploration and Preprocessing

- Load the California Housing Prices dataset.
- Perform exploratory data analysis (EDA) to understand the dataset (https://towardsdatascience.com/exploratory-data-analysis-8fc1cb20fd15).
- 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. 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).

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).

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.

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.

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.

Dataset

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

Tasks

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.

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. 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.

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.

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.

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.

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.

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)

1. Regression

1. Data Exploration and Preprocessing

Load the California Housing Prices dataset.

In [2]:
```python
from sklearn.datasets import fetch_california_housing

california_housing = fetch_california_housing(as_frame=True)
```

```
In [3]: print(california_housing.DESCR)
```

```
.. _california_housing_dataset:

California Housing dataset
--------------------------

**Data Set Characteristics:**

    :Number of Instances: 20640

    :Number of Attributes: 8 numeric, predictive attributes and the target

    :Attribute Information:
        - MedInc        median income in block group
        - HouseAge      median house age in block group
        - AveRooms      average number of rooms per household
        - AveBedrms     average number of bedrooms per household
        - Population    block group population
        - AveOccup      average number of household members
        - Latitude      block group latitude
        - Longitude     block group longitude

    :Missing Attribute Values: None

This dataset was obtained from the StatLib repository.
https://www.dcc.fc.up.pt/~ltorgo/Regression/cal_housing.html

The target variable is the median house value for California districts,
expressed in hundreds of thousands of dollars ($100,000).

This dataset was derived from the 1990 U.S. census, using one row per censu
s
block group. A block group is the smallest geographical unit for which the
U.S.
Census Bureau publishes sample data (a block group typically has a populati
on
of 600 to 3,000 people).

A household is a group of people residing within a home. Since the average
number of rooms and bedrooms in this dataset are provided per household, th
ese
columns may take surprisingly large values for block groups with few househ
olds
and many empty houses, such as vacation resorts.

It can be downloaded/loaded using the
:func:`sklearn.datasets.fetch_california_housing` function.

.. topic:: References

    - Pace, R. Kelley and Ronald Barry, Sparse Spatial Autoregressions,
      Statistics and Probability Letters, 33 (1997) 291-297
```

```
In [4]: california_housing.frame.head()
```

Out[4]:

	MedInc	HouseAge	AveRooms	AveBedrms	Population	AveOccup	Latitude	Longi
0	8.3252	41.0	6.984127	1.023810	322.0	2.555556	37.88	-122.2
1	8.3014	21.0	6.238137	0.971880	2401.0	2.109842	37.86	-122.2
2	7.2574	52.0	8.288136	1.073446	496.0	2.802260	37.85	-122.2
3	5.6431	52.0	5.817352	1.073059	558.0	2.547945	37.85	-122.2
4	3.8462	52.0	6.281853	1.081081	565.0	2.181467	37.85	-122.2

2. Perform exploratory data analysis (EDA) to understand the dataset.

```
In [5]: california_housing.frame.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 20640 entries, 0 to 20639
Data columns (total 9 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
 8   MedHouseVal  20640 non-null  float64
dtypes: float64(9)
memory usage: 1.4 MB
```

The dataset has 20'640 samples and 8 features; All features are numerical features encoded as floating number. There are no missing values.

Let's see statistics of our data:

In [6]:
```
california_housing.frame.describe()
```

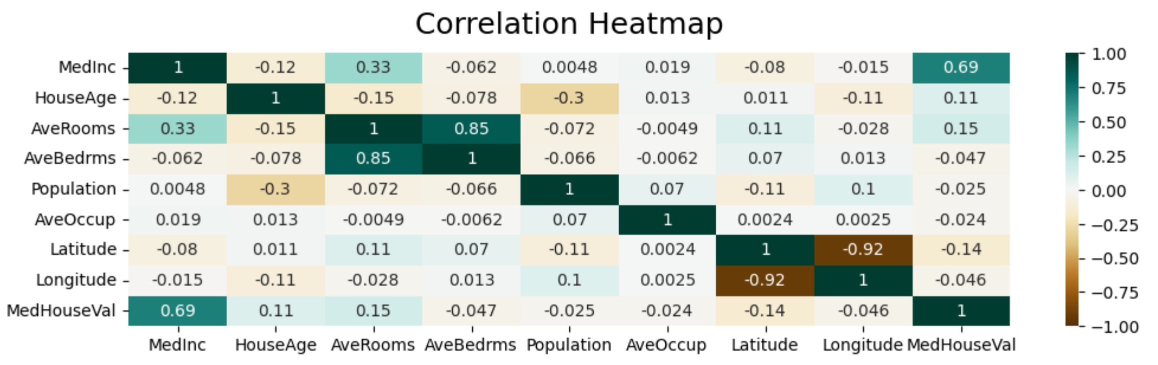
Out[6]:

	MedInc	HouseAge	AveRooms	AveBedrms	Population	AveO
count	20640.000000	20640.000000	20640.000000	20640.000000	20640.000000	20640
mean	3.870671	28.639486	5.429000	1.096675	1425.476744	3.0706
std	1.899822	12.585558	2.474173	0.473911	1132.462122	10.386
min	0.499900	1.000000	0.846154	0.333333	3.000000	0.692
25%	2.563400	18.000000	4.440716	1.006079	787.000000	2.429
50%	3.534800	29.000000	5.229129	1.048780	1166.000000	2.818
75%	4.743250	37.000000	6.052381	1.099526	1725.000000	3.282
max	15.000100	52.000000	141.909091	34.066667	35682.000000	1243.3

Some features have extreme values (outliers): Number of bedrooms, rooms, population and occupation.

Now let's create a Correlation Heat map to see any relationships between variables.

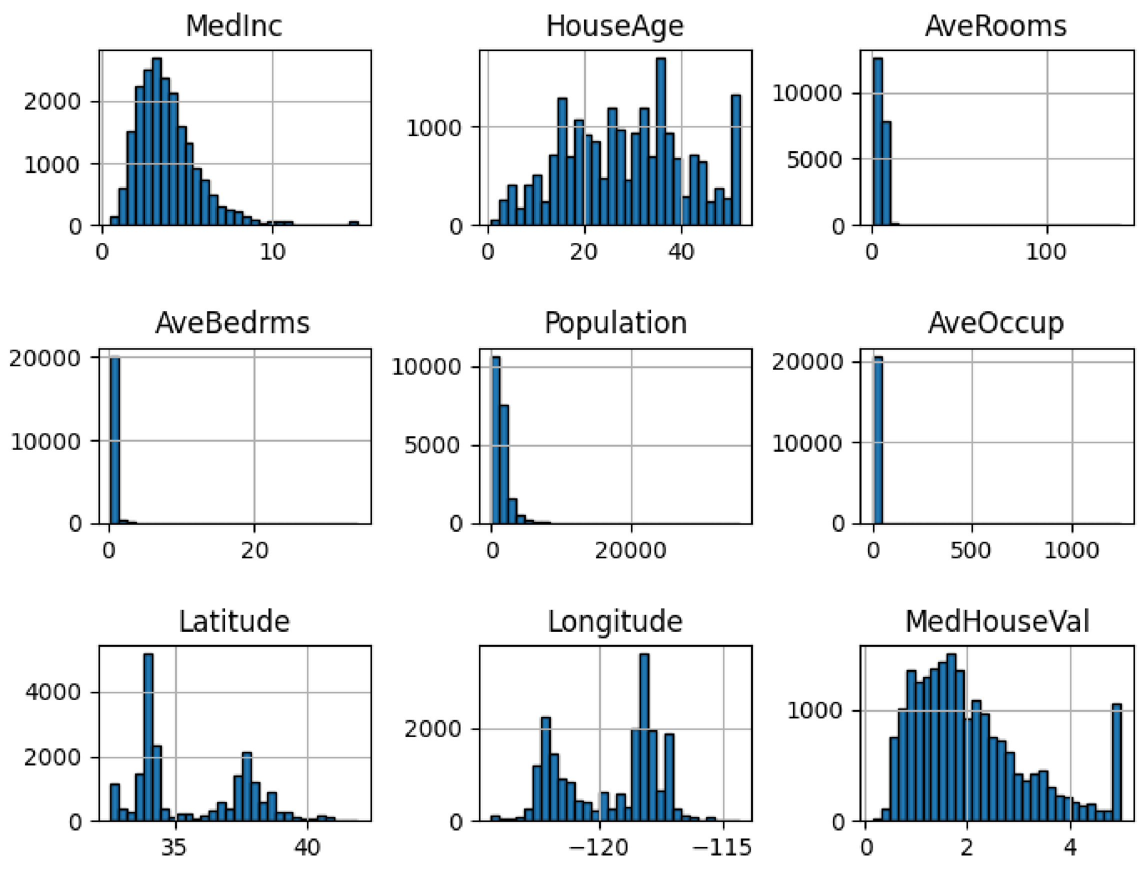
In [12]:
```
import pandas as pd
import numpy as np
import seaborn as sns
import matplotlib.pyplot as plt
plt.figure(figsize=(12, 3))
heatmap = sns.heatmap(california_housing.frame.corr(), vmin=-1, vmax=1, ann
ot=True, cmap='BrBG')
heatmap.set_title('Correlation Heatmap', fontdict={'fontsize':18}, pad=12);
plt.savefig('heatmap.png', dpi=300, bbox_inches='tight')
```



The features are not very correlated with each other. The Median income has the highest correlation with a target variable meaning that the median income is a useful feature to help at predicting the median house values.

Let's create a histogram

In [9]:
```
import matplotlib.pyplot as plt

california_housing.frame.hist(figsize=(8, 6), bins=30, edgecolor="black")
plt.subplots_adjust(hspace=0.7, wspace=0.4)
```



The median income is a distribution with a long tail. It means that the salary of people is more or less normally distributed but there is some people getting a high salary.

House age has the distribution is more or less uniform.

The target distribution has a long tail as well. Plus there are high-valued houses.

Let's normalize/standardize the features.

In [10]:
```
from sklearn.preprocessing import StandardScaler
# transform data
scaler = StandardScaler()
#https://towardsdatascience.com/patsy-build-powerful-features-with-arbitrar
y-python-code-bb4bb98db67a
```

In [11]:
```
df = pd.DataFrame(california_housing.data, columns=california_housing.featu
re_names)
df_scaled = scaler.fit_transform(df)
```

In [12]:
```
df_scaled.shape
```

Out[12]:
```
(20640, 8)
```

In [13]:
```
df['target'] = california_housing.target
```

In [14]:
```
target_scaled = scaler.fit_transform(df['target'].values.reshape(-1, 1))
```

In [15]:
```
target_scaled.shape
```

Out[15]:
```
(20640, 1)
```

2. Regression with Splines

Fit a spline model to the data. I use the patsy library in Python for creating spline features.

Since income group, region, and year may play a big part in determining the life expectancy, we will use Patsy to transform these 3 features into a matrix.

In [16]:
```
import patsy
from patsy import dmatrix
import statsmodels.api as sm
from sklearn.model_selection import train_test_split
from sklearn.metrics import mean_squared_error, r2_score
from math import sqrt

# Split the data into features (X) and target variable (y)
X = df_scaled
y = target_scaled
X_d = pd.DataFrame(X, columns=california_housing.feature_names)
y_d = pd.DataFrame(y, columns=['MedHouseVal'])

train_df = pd.concat([X_d, y_d], axis=1)
train_df.head(3)
```

Out[16]:

	MedInc	HouseAge	AveRooms	AveBedrms	Population	AveOccup	Latitude	Lor
0	2.344766	0.982143	0.628559	-0.153758	-0.974429	-0.049597	1.052548	-1.3
1	2.332238	-0.607019	0.327041	-0.263336	0.861439	-0.092512	1.043185	-1.3
2	1.782699	1.856182	1.155620	-0.049016	-0.820777	-0.025843	1.038503	-1.3

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).

In [17]:
```
from patsy import dmatrices, cr
import statsmodels.api as sm
from sklearn.model_selection import train_test_split
from sklearn.metrics import mean_squared_error, r2_score
from math import sqrt

# Split the data into features (X) and target variable (y)
formula = "MedHouseVal ~ cr(MedInc, df=10) + HouseAge + AveBedrms + Populat
ion + AveRooms + AveOccup + Latitude + Longitude"
y, X = dmatrices(formula, data=train_df, return_type='dataframe')

# Split the data into training and testing sets
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, ra
ndom_state=42)
```

In [18]:
```
# Experiment with different degrees of freedom
for df in range(3, 15):  # Trying degrees of freedom from 3 to 14
    # Update the formula with the current degrees of freedom
    formula = f"MedHouseVal ~ cr(MedInc, df={df}) + cr(HouseAge, df={df}) +
AveBedrms + Population + AveRooms + cr(AveOccup, df={df}) + cr(Latitude, df
={df}) + cr(Longitude, df={df})"
    y, X = dmatrices(formula, data=train_df, return_type='dataframe')

    # Split the data 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)

    # Fit the model
    model = sm.OLS(y_train, X_train)
    results = model.fit()

    # Make predictions
    y_pred = results.predict(X_test)

    # Evaluate the model
    rmse = sqrt(mean_squared_error(y_test, y_pred))
    r2 = r2_score(y_test, y_pred)

    print(f"Degrees of Freedom: {df}")
    print(f"RMSE: {rmse:.4f}")
    print(f"R²: {r2:.4f}")
    print("----------------------")
```

```
Degrees of Freedom: 3
RMSE: 0.6404
R²: 0.5833
-----------------------
Degrees of Freedom: 4
RMSE: 0.5876
R²: 0.6491
-----------------------
Degrees of Freedom: 5
RMSE: 0.5738
R²: 0.6655
-----------------------
Degrees of Freedom: 6
RMSE: 0.5586
R²: 0.6830
-----------------------
Degrees of Freedom: 7
RMSE: 0.5494
R²: 0.6932
-----------------------
Degrees of Freedom: 8
RMSE: 0.5517
R²: 0.6908
-----------------------
Degrees of Freedom: 9
RMSE: 0.5424
R²: 0.7010
-----------------------
Degrees of Freedom: 10
RMSE: 0.5427
R²: 0.7007
-----------------------
Degrees of Freedom: 11
RMSE: 0.5432
R²: 0.7002
-----------------------
Degrees of Freedom: 12
RMSE: 0.5376
R²: 0.7063
-----------------------
Degrees of Freedom: 13
RMSE: 0.5381
R²: 0.7058
-----------------------
Degrees of Freedom: 14
RMSE: 0.5318
R²: 0.7126
-----------------------
```

Polynomial features can capture non-linear relationships, and I am using them on the Median Income feature. That makes sense, especially in the context of predicting housing prices, as there might be non-linear patterns in how median income influences home values. The higher the degrees of freedom, the "wigglier" the spline gets because the number of knots is increased. In our case the complexity is increased to 9 degrees of freedom and RMSE is lowest on a Test data showing that we are still not overfitted our data. Then as we increase df our error is starting growing back.

In our case, with Regression with Splines and a degree of freedom of 9, I am allowing for a more flexible, non-linear relationship.

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, R²).

Random Forest

In [19]:
```
from sklearn.ensemble import RandomForestRegressor, AdaBoostRegressor, Grad
ientBoostingRegressor
from sklearn.model_selection import GridSearchCV, train_test_split
from sklearn.metrics import mean_squared_error, r2_score
from math import sqrt
import pandas as pd
```

In [20]:
```
# Split the data into features (X) and target variable (y)
X = train_df.drop('MedHouseVal', axis=1)
y = train_df['MedHouseVal']

# Split the data into training and testing sets
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, ra
ndom_state=42)
```

In [21]:
```
#y_train
```

In [22]:
```
from sklearn.model_selection import GridSearchCV
from sklearn.ensemble import RandomForestRegressor
from sklearn.metrics import make_scorer

parameters = {
    'n_estimators': [100, 200, 300],
    'max_depth': [6,10,12],
    ##'min_samples_split': [2, 5, 10],
    #'ccp_alpha': [0, 0.0001, 0.001]
}
regr = RandomForestRegressor(random_state=42)

clf = GridSearchCV(regr, parameters, cv=5)
clf.fit(X_train, y_train)
```

Out[22]:
```
▸           GridSearchCV
▸ estimator: RandomForestRegressor
      ▸ RandomForestRegressor
```

In [23]:
```
# Get the best Random Forest model
best_rf_model = clf.best_estimator_

# Evaluate the best model
y_pred_rf = best_rf_model.predict(X_test)
rmse_rf = sqrt(mean_squared_error(y_test, y_pred_rf))
r2_rf = r2_score(y_test, y_pred_rf)

print("Best Random Forest Model:")
print(f"RMSE: {rmse_rf:.4f}")
print(f"R²: {r2_rf:.4f}")
```

```
Best Random Forest Model:
RMSE: 0.4525
R²: 0.7919
```

In [24]:
```
best_rf_model
```

Out[24]:
```
▾                     RandomForestRegressor
RandomForestRegressor(max_depth=12, n_estimators=300, random_state=42)
```

Ada Boost

In [25]:
```
# AdaBoost
ab_model = AdaBoostRegressor(random_state=42)
ab_params = {'n_estimators': [100, 150, 200],
             'learning_rate': [0.001, 0.01, 0.1]}
ab_grid = GridSearchCV(ab_model, ab_params, cv=5, scoring='neg_mean_squared
_error')
ab_grid.fit(X_train, y_train)

# Best AdaBoost model
best_ab_model = ab_grid.best_estimator_
```

In [26]:
```
best_ab_model
```

Out[26]:
```
▾                     AdaBoostRegressor
AdaBoostRegressor(learning_rate=0.01, n_estimators=200, random_state=42)
```

In [27]:
```
# Evaluate the best model
y_pred_ab = best_ab_model.predict(X_test)
rmse_ab = sqrt(mean_squared_error(y_test, y_pred_ab))
r2_ab = r2_score(y_test, y_pred_ab)

print("Best Random Forest Model:")
print(f"RMSE: {rmse_ab:.4f}")
print(f"R²: {r2_ab:.4f}")
```

```
Best Random Forest Model:
RMSE: 0.6600
R²: 0.5574
```

Gradient Boosting

In [28]:
```
# GradientBoosting
gb_model = GradientBoostingRegressor(random_state=42)
gb_params = {'n_estimators': [100, 150, 200],
             'learning_rate': [0.001, 0.01, 0.1],
             'max_depth': [5, 7, 10, 12]}
gb_grid = GridSearchCV(gb_model, gb_params, cv=5, scoring='neg_mean_squared
_error')
gb_grid.fit(X_train, y_train)

# Best GradientBoosting model
best_gb_model = gb_grid.best_estimator_
```

In [29]:
```
best_gb_model
```

Out[29]:
```
▾                         GradientBoostingRegressor

GradientBoostingRegressor(max_depth=7, n_estimators=200, random_state=42)
```

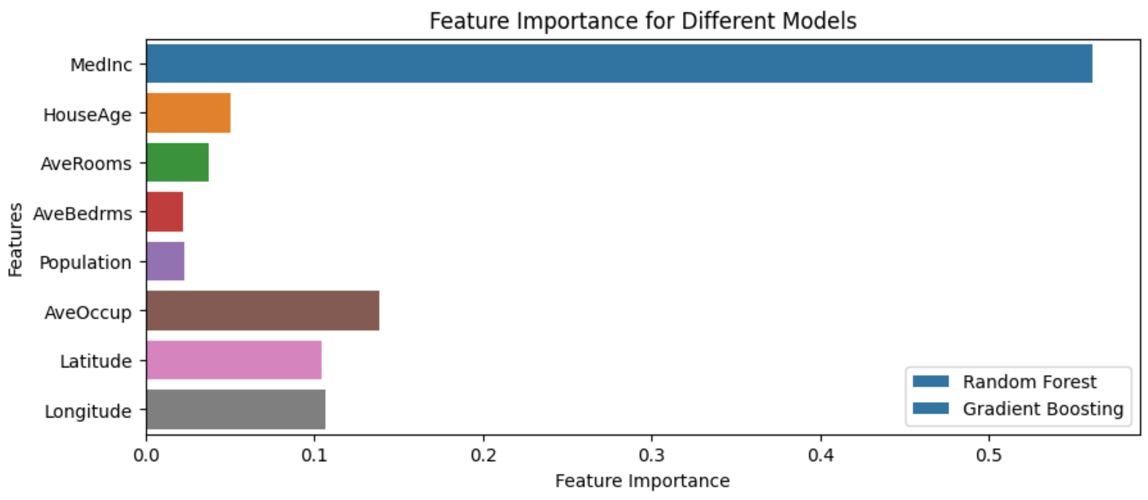
In [30]:
```
# Evaluate the best model
y_pred_gb = best_gb_model.predict(X_test)
rmse_gb = sqrt(mean_squared_error(y_test, y_pred_gb))
r2_gb = r2_score(y_test, y_pred_gb)

print("Best Random Forest Model:")
print(f"RMSE: {rmse_gb:.4f}")
print(f"R²: {r2_gb:.4f}")
```

```
Best Random Forest Model:
RMSE: 0.3946
R²: 0.8418
```

Feature Importance

In [31]:
```
# Get feature importances
feature_importances_gb = best_gb_model.feature_importances_
feature_importances_rf = best_rf_model.feature_importances_
feature_importances_ab = best_ab_model.feature_importances_
```

In [40]:
```python
import matplotlib.pyplot as plt
import seaborn as sns

# Assuming you have feature importances for each model
models = ['Random Forest', 'Gradient Boosting']
feature_importances = [feature_importances_rf, feature_importances_gb]

plt.figure(figsize=(10, 4))
for model, importance in zip(models, feature_importances):
    sns.barplot(x=importance, y=X_train.columns, orient='h', label=model)

plt.xlabel('Feature Importance')
plt.ylabel('Features')
plt.title('Feature Importance for Different Models')
plt.legend()
plt.show()
```



In [33]:
```python
feature_importances_rf = best_rf_model.feature_importances_
feature_importances_ab = best_ab_model.feature_importances_
feature_importances_gb
```

Out[33]:
```
array([0.53618607, 0.0472675 , 0.03439989, 0.01897591, 0.01979048,
       0.13234506, 0.10407207, 0.10696303])
```

In [34]:
```
# Evaluate models
models = {'Random Forest': best_rf_model, 'AdaBoost': best_ab_model, 'Gradi
entBoosting': best_gb_model}

for name, model in models.items():
    y_pred = model.predict(X_test)
    rmse = sqrt(mean_squared_error(y_test, y_pred))
    r2 = r2_score(y_test, y_pred)

    print(f"Model: {name}")
    print(f"RMSE: {rmse:.4f}")
    print(f"R²: {r2:.4f}")
    print("-----------------------")
```

```
Model: Random Forest
RMSE: 0.4525
R²: 0.7919
-----------------------
Model: AdaBoost
RMSE: 0.6600
R²: 0.5574
-----------------------
Model: GradientBoosting
RMSE: 0.3946
R²: 0.8418
-----------------------
```

4. Model Comparison

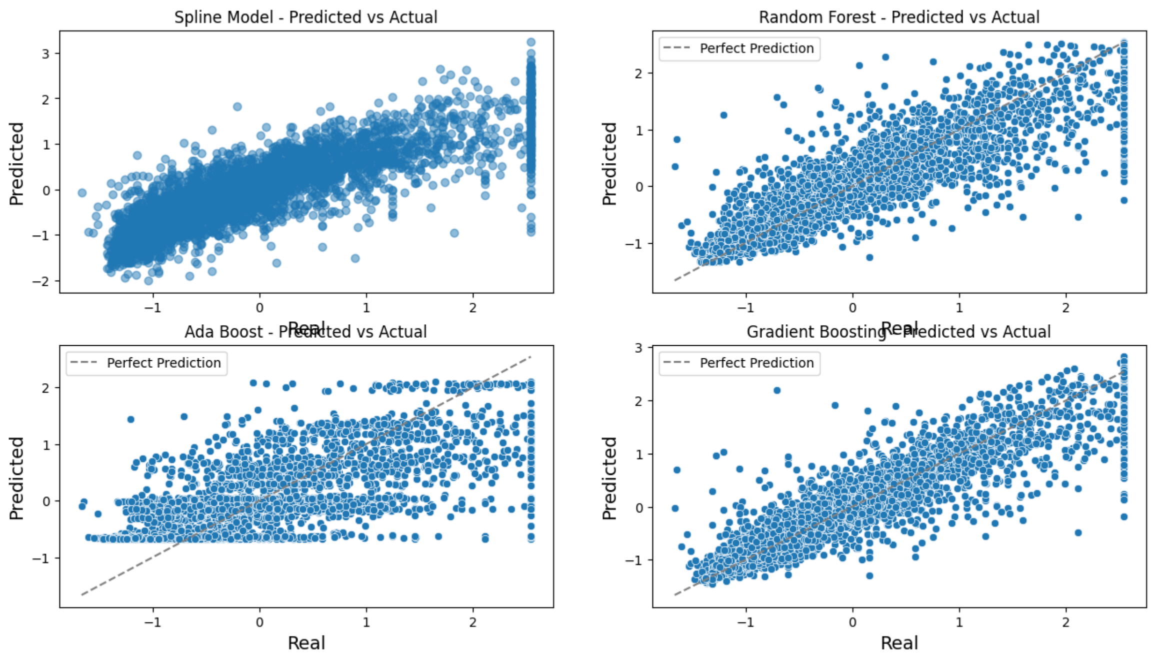
Compare the performance of the spline models with the advanced decision tree regression models. Use visualizations to compare the predicted vs actual values for each model. Discuss the bias-variance tradeoff for each model based on your results.

Let's create a graph showing Actual hosing prices vs Predicted for all models

In [35]:
```
import matplotlib.pyplot as plt
import seaborn as sns

#SPLINE MODEL
# Split the data into features (X) and target variable (y)
formula = "MedHouseVal ~ cr(MedInc, df=9) + HouseAge + AveBedrms + Populati
on + AveRooms + cr(AveOccup, df=9) + cr(Latitude, df=9) + cr(Longitude, df=
9)"
y, X = dmatrices(formula, data=train_df, return_type='dataframe')
# Split the data into training and testing sets
X_trains, X_tests, y_trains, y_tests = train_test_split(X, y, test_size=0.
2, random_state=0)
# Fit the model
model = sm.OLS(y_trains, X_trains)
results = model.fit()
# Make predictions
y_preds = results.predict(X_tests)
```

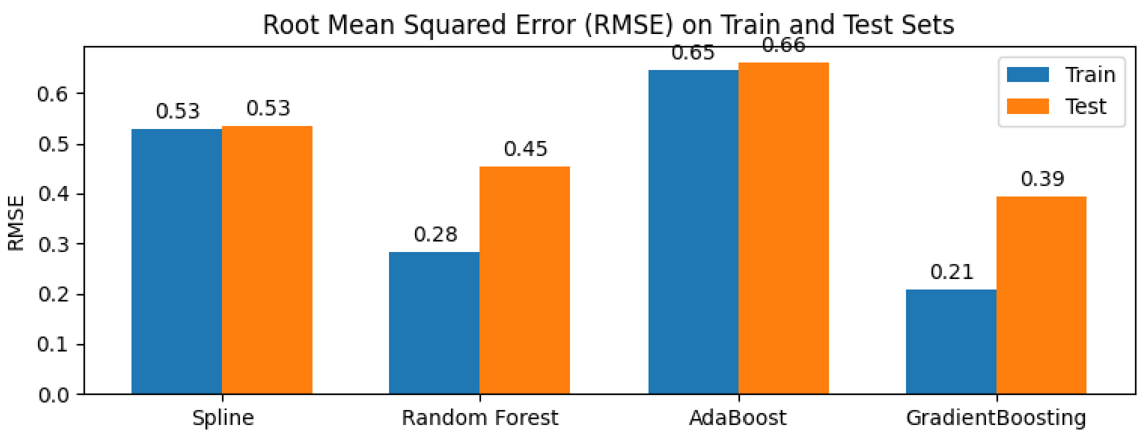
In [36]:
```
# Visualize predicted vs actual values for SPLINE model

plt.figure(figsize=(15, 8))

plt.subplot(2, 2, 1)
#plt.plot(np.linspace(min(y_tests), max(y_tests), 100), np.linspace(min(y_t
ests), max(y_tests), 100), '--', color='gray', label='Perfect Prediction')
plt.scatter(y_tests,y_preds, alpha = 0.5)
#sns.scatterplot(x = y_tests, y = y_preds)
plt.title('Spline Model - Predicted vs Actual')
plt.xlabel("Real", fontsize=14)
plt.ylabel("Predicted", fontsize=14)

#RANDOMFOREST
# Split the data into features (X) and target variable (y)
X = train_df.drop('MedHouseVal', axis=1)
y = train_df['MedHouseVal']
# Split the data into training and testing sets
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, ra
ndom_state=42)

y_pred_rf = best_rf_model.predict(X_test)
y_pred_ab = best_ab_model.predict(X_test)
y_pred_gb = best_gb_model.predict(X_test)

plt.subplot(2, 2, 2)
plt.plot(np.linspace(min(y_test), max(y_test), 100), np.linspace(min(y_tes
t), max(y_test), 100), '--', color='gray', label='Perfect Prediction')
sns.scatterplot(x = y_test, y = y_pred_rf)
plt.title('Random Forest - Predicted vs Actual')
plt.xlabel("Real", fontsize=14)
plt.ylabel("Predicted", fontsize=14)

plt.subplot(2, 2, 3)
plt.plot(np.linspace(min(y_test), max(y_test), 100), np.linspace(min(y_tes
t), max(y_test), 100), '--', color='gray', label='Perfect Prediction')
sns.scatterplot(x = y_test, y = y_pred_ab)
plt.title('Ada Boost - Predicted vs Actual')
plt.xlabel("Real", fontsize=14)
plt.ylabel("Predicted", fontsize=14)

plt.subplot(2, 2, 4)
plt.plot(np.linspace(min(y_test), max(y_test), 100), np.linspace(min(y_tes
t), max(y_test), 100), '--', color='gray', label='Perfect Prediction')
sns.scatterplot(x = y_test, y = y_pred_gb)
plt.title('Gradient Boosting - Predicted vs Actual')
plt.xlabel("Real", fontsize=14)
plt.ylabel("Predicted", fontsize=14)
```

```
Out[36]:  Text(0, 0.5, 'Predicted')
```



Let's calculate RMSE for Test and Train dataset for all models

In [43]:
```
from sklearn.metrics import mean_squared_error

# Assuming you have predictions for spline model (y_pred_spline),
# Random Forest (y_pred_rf), AdaBoost (y_pred_ab), and Gradient Boosting (y
_pred_gb)

# Calculate RMSE for each model on both training and test sets
rmse_spline_train = np.sqrt(mean_squared_error(y_trains, results.predict(X_
trains)))
rmse_spline_test = np.sqrt(mean_squared_error(y_tests, y_preds))

rmse_rf_train = np.sqrt(mean_squared_error(y_train, best_rf_model.predict(X
_train)))
rmse_rf_test = np.sqrt(mean_squared_error(y_test, y_pred_rf))

rmse_ab_train = np.sqrt(mean_squared_error(y_train, best_ab_model.predict(X
_train)))
rmse_ab_test = np.sqrt(mean_squared_error(y_test, y_pred_ab))

rmse_gb_train = np.sqrt(mean_squared_error(y_train, best_gb_model.predict(X
_train)))
rmse_gb_test = np.sqrt(mean_squared_error(y_test, y_pred_gb))

# Create bar plots for RMSE on both training and test sets
models = ['Spline', 'Random Forest', 'AdaBoost', 'GradientBoosting']
rmse_train_values = [rmse_spline_train, rmse_rf_train, rmse_ab_train, rmse_
gb_train]
rmse_test_values = [rmse_spline_test, rmse_rf_test, rmse_ab_test, rmse_gb_t
est]

width = 0.35  # the width of the bars

fig, ax = plt.subplots(figsize=(9, 3))
rects1 = ax.bar(np.arange(len(models)) - width/2, rmse_train_values, width,
label='Train')
rects2 = ax.bar(np.arange(len(models)) + width/2, rmse_test_values, width,
label='Test')

# Add labels, title, and legend
ax.set_ylabel('RMSE')
ax.set_title('Root Mean Squared Error (RMSE) on Train and Test Sets')
ax.set_xticks(np.arange(len(models)))
ax.set_xticklabels(models)
ax.legend()

# Display the RMSE values on top of the bars
for rects in [rects1, rects2]:
    for rect in rects:
        height = rect.get_height()
        ax.annotate(f'{height:.2f}',
                    xy=(rect.get_x() + rect.get_width() / 2, height),
                    xytext=(0, 3),  # 3 points vertical offset
                    textcoords="offset points",
                    ha='center', va='bottom')

plt.show()
```



The best is a Gradient boosting model which has the lowest RMSE compared with other models. On the second place is a Random Forest model.

Spline Model: Bias: The spline model may have bias if the chosen degree of freedom is too low, leading to oversimplification. Variance: As the degree of freedom increases, the spline model becomes more flexible, potentially leading to higher variance.

Random Forest Model: Bias: Random Forests can capture complex relationships in the data, leading to low bias. Variance: Random Forests tend to have low variance due to the ensemble nature, aggregating predictions from multiple trees.

AdaBoost Model: Bias: AdaBoost focuses on improving the performance of weak learners, reducing bias. Variance: AdaBoost can be sensitive to noisy data, potentially increasing variance.

Gradient Boosting Model: Bias: Gradient Boosting builds trees sequentially, reducing bias over iterations. Variance: Gradient Boosting can have moderate to high variance, especially if the model is too complex.

In Summary: the spline model's bias-variance tradeoff depends on the chosen degree of freedom, while advanced decision tree models offer a tradeoff between capturing complex patterns and controlling variance. Random Forest tends to have a balanced bias-variance tradeoff, AdaBoost focuses on reducing bias, and Gradient Boosting aims to balance bias and variance over iterations.

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.

Regression with Splines (Degrees of Freedom: 9):

RMSE: 0.5424

R²: 0.7010

This model seems to have decent performance, with a moderately low RMSE and a reasonably high R². *Median income* feature was used to make a polynominal with a high degree as there might be non-linear patterns in how median income influences home values.It appears to perform better than AdaBoost but is slightly behind Random Forest and GradientBoosting.

Random Forest:

RMSE: 0.4525

R²: 0.7919

Random Forest is doing well, with the lowest RMSE and a high R². It's a robust model known for handling complex relationships in data. The best hyperparameters max_depth=12, n_estimators=300 seem to be working well.

AdaBoost:

RMSE: 0.6600

R²: 0.5574

AdaBoost is the weakest performer in terms of both RMSE and R² in this comparison. Best hyperparameters: learning_rate=0.01, n_estimators=200.

GradientBoosting:

RMSE: 0.3946

R²: 0.8418

GradientBoosting shines here with the lowest RMSE and the highest R², indicating strong predictive performance. Best hyperparameters max_depth=7, n_estimators=200.

Feature importance

The most important features are:

```
- MedInc        median income in block group
- AveOccup      average number of household members
- Latitude      block group latitude
- Longitude     block group longitude
```

2. Classification

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

1. Data Exploration and Preprocessing

Load the Breast Canser Wiscinsin (Diagnostic) dataset.

In [1]:
```
from sklearn.datasets import load_breast_cancer
```

In [2]:
```python
cancer = load_breast_cancer()
print(cancer.DESCR) # Print the data set description
```

```
.. _breast_cancer_dataset:

Breast cancer wisconsin (diagnostic) dataset
--------------------------------------------

**Data Set Characteristics:**

    :Number of Instances: 569

    :Number of Attributes: 30 numeric, predictive attributes and the class

    :Attribute Information:
        - radius (mean of distances from center to points on the perimeter)
        - texture (standard deviation of gray-scale values)
        - perimeter
        - area
        - smoothness (local variation in radius lengths)
        - compactness (perimeter^2 / area - 1.0)
        - concavity (severity of concave portions of the contour)
        - concave points (number of concave portions of the contour)
        - symmetry
        - fractal dimension ("coastline approximation" - 1)

        The mean, standard error, and "worst" or largest (mean of the three
        worst/largest values) of these features were computed for each imag
e,
        resulting in 30 features.  For instance, field 0 is Mean Radius, fi
eld
        10 is Radius SE, field 20 is Worst Radius.

        - class:
                - WDBC-Malignant
                - WDBC-Benign

    :Summary Statistics:

    ===================================== ====== ======
                                           Min    Max
    ===================================== ====== ======
    radius (mean):                        6.981  28.11
    texture (mean):                       9.71   39.28
    perimeter (mean):                     43.79  188.5
    area (mean):                          143.5  2501.0
    smoothness (mean):                    0.053  0.163
    compactness (mean):                   0.019  0.345
    concavity (mean):                     0.0    0.427
    concave points (mean):                0.0    0.201
    symmetry (mean):                      0.106  0.304
    fractal dimension (mean):             0.05   0.097
    radius (standard error):              0.112  2.873
    texture (standard error):             0.36   4.885
    perimeter (standard error):           0.757  21.98
    area (standard error):                6.802  542.2
    smoothness (standard error):          0.002  0.031
    compactness (standard error):         0.002  0.135
    concavity (standard error):           0.0    0.396
    concave points (standard error):      0.0    0.053
    symmetry (standard error):            0.008  0.079
    fractal dimension (standard error):   0.001  0.03
    radius (worst):                       7.93   36.04
    texture (worst):                      12.02  49.54
    perimeter (worst):                    50.41  251.2
    area (worst):                         185.2  4254.0
    smoothness (worst):                   0.071  0.223
    compactness (worst):                  0.027  1.058
    concavity (worst):                    0.0    1.252
    concave points (worst):               0.0    0.291
    symmetry (worst):                     0.156  0.664
    fractal dimension (worst):            0.055  0.208
    ===================================== ====== ======

    :Missing Attribute Values: None

    :Class Distribution: 212 - Malignant, 357 - Benign

    :Creator:  Dr. William H. Wolberg, W. Nick Street, Olvi L. Mangasarian

    :Donor: Nick Street

    :Date: November, 1995

This is a copy of UCI ML Breast Cancer Wisconsin (Diagnostic) datasets.
https://goo.gl/U2Uwz2

Features are computed from a digitized image of a fine needle
aspirate (FNA) of a breast mass.  They describe
characteristics of the cell nuclei present in the image.

Separating plane described above was obtained using
Multisurface Method-Tree (MSM-T) [K. P. Bennett, "Decision Tree
Construction Via Linear Programming." Proceedings of the 4th
Midwest Artificial Intelligence and Cognitive Science Society,
pp. 97-101, 1992], a classification method which uses linear
programming to construct a decision tree.  Relevant features
were selected using an exhaustive search in the space of 1-4
features and 1-3 separating planes.

The actual linear program used to obtain the separating plane
in the 3-dimensional space is that described in:
[K. P. Bennett and O. L. Mangasarian: "Robust Linear
Programming Discrimination of Two Linearly Inseparable Sets",
Optimization Methods and Software 1, 1992, 23-34].

This database is also available through the UW CS ftp server:

ftp ftp.cs.wisc.edu
cd math-prog/cpo-dataset/machine-learn/WDBC/

.. topic:: References

   - W.N. Street, W.H. Wolberg and O.L. Mangasarian. Nuclear feature extrac
tion
     for breast tumor diagnosis. IS&T/SPIE 1993 International Symposium on
     Electronic Imaging: Science and Technology, volume 1905, pages 861-87
0,
     San Jose, CA, 1993.
   - O.L. Mangasarian, W.N. Street and W.H. Wolberg. Breast cancer diagnosi
s and
     prognosis via linear programming. Operations Research, 43(4), pages 57
0-577,
     July-August 1995.
   - W.H. Wolberg, W.N. Street, and O.L. Mangasarian. Machine learning tech
```

```
niques
     to diagnose breast cancer from fine-needle aspirates. Cancer Letters 7
7 (1994)
     163-171.
```

Number of Instances: 569

Number of Attributes: 30 numeric, predictive attributes and the class.

class: WDBC-Malignant, WDBC-Benign

Missing Attribute Values: None

Class Distribution: 212 - Malignant, 357 - Benign

In [3]:
```
import numpy as np
import pandas as pd

dff=pd.DataFrame(cancer.data,columns =[cancer.feature_names])
dff['target']=pd.Series(data=cancer.target,index=dff.index)
```

In [4]:
```
dff.head(2)
```

Out[4]:

	mean radius	mean texture	mean perimeter	mean area	mean smoothness	mean compactness	mean concavity	mean concave points
0	17.99	10.38	122.8	1001.0	0.11840	0.27760	0.3001	0.14710
1	20.57	17.77	132.9	1326.0	0.08474	0.07864	0.0869	0.07017

2 rows × 31 columns

In [5]:
```
dff.describe()
```

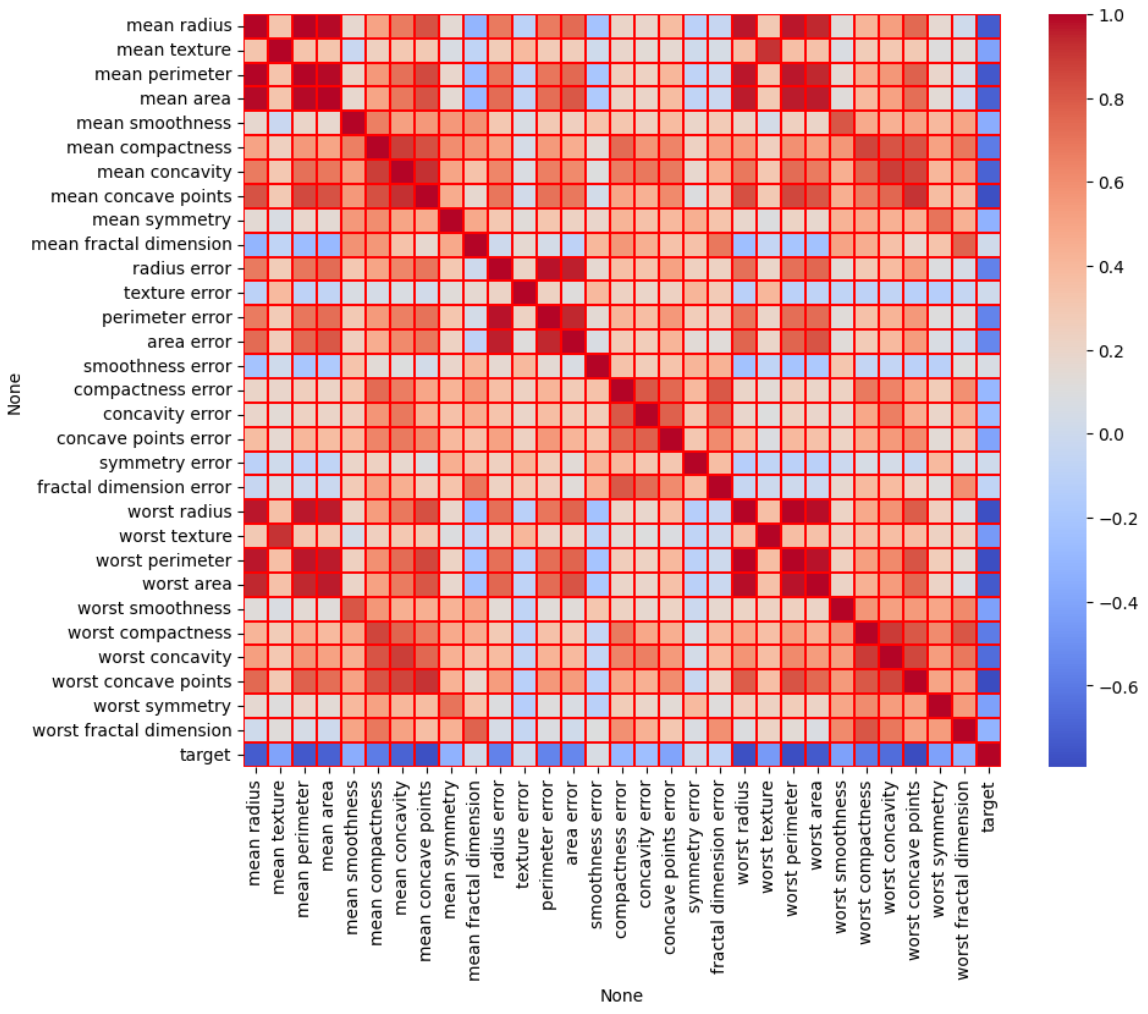
Out[5]:

	mean radius	mean texture	mean perimeter	mean area	mean smoothness	mean compactness
count	569.000000	569.000000	569.000000	569.000000	569.000000	569.000000
mean	14.127292	19.289649	91.969033	654.889104	0.096360	0.104341
std	3.524049	4.301036	24.298981	351.914129	0.014064	0.052813
min	6.981000	9.710000	43.790000	143.500000	0.052630	0.019380
25%	11.700000	16.170000	75.170000	420.300000	0.086370	0.064920
50%	13.370000	18.840000	86.240000	551.100000	0.095870	0.092630
75%	15.780000	21.800000	104.100000	782.700000	0.105300	0.130400
max	28.110000	39.280000	188.500000	2501.000000	0.163400	0.345400

8 rows × 31 columns

Target classes are more ol less balanced. There are few feature have outliers.

In [37]:
```
import pandas as pd
import pandas as pd
import seaborn as sns
import matplotlib.pyplot as plt

f,ax = plt.subplots(figsize=(10, 8))
sns.heatmap(dff.corr(), annot= False, linewidths= 0.3, linecolor= "red", fm
t= ".0%", ax= ax, cmap = 'coolwarm')
plt.show()
```



There is a High correlation between features that can lead to multicollinearity, which can be problematic for certain statistical models and may not provide additional information.

Here are some considerations:

Redundancy:

If two features are highly correlated, it means they are providing similar information. Keeping both features might not add much value and could lead to redundancy in your model.

Computational Efficiency:

High correlation can lead to computational inefficiencies, especially in algorithms that involve matrix inversion. Removing one of the correlated features can speed up computations.

Model Interpretability:

In some cases, having highly correlated features might make it harder to interpret the model. Removing one of the features can simplify the interpretation.

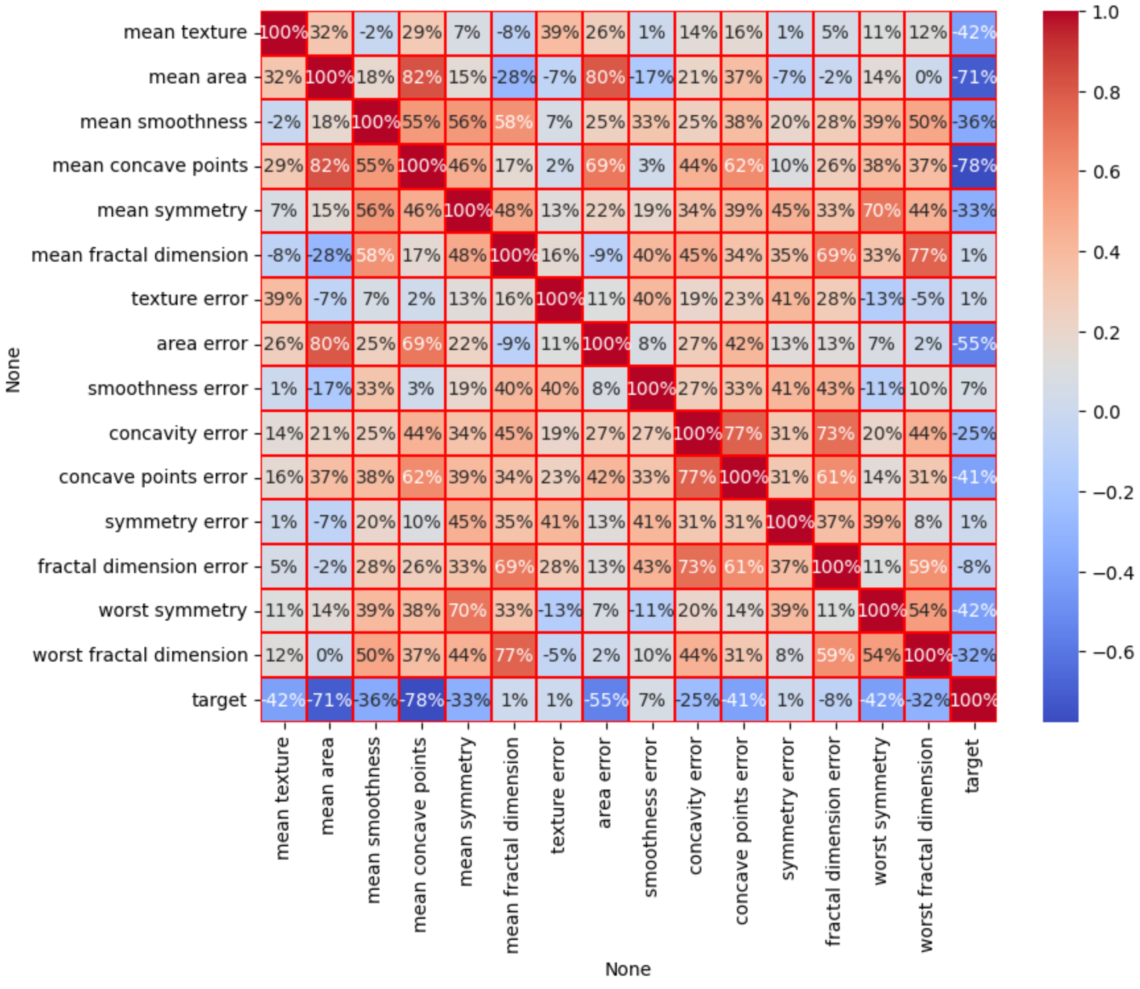
In our case we will drop features where correlation is more than 80%. Therefore the following features will be droped: worst radius, worst perimeter, mean perimeter, mean radius, radius error, perimeter error, worst concave points, worst concavity, worst smoothness, compactness error, worst texture, worst area, worst compactness.

In [7]:
```
df1 = dff.drop(['worst radius','worst texture', 'mean concavity', 'mean com
pactness', 'worst compactness', 'worst area','worst perimeter','mean perime
ter', 'mean radius', 'radius error', 'perimeter error', 'worst concave poin
ts', 'worst concavity', 'worst smoothness', 'compactness error'], axis=1)
```

```
<ipython-input-7-3e127ce2012c>:1: PerformanceWarning: dropping on a non-lex
sorted multi-index without a level parameter may impact performance.
  df1 = dff.drop(['worst radius','worst texture', 'mean concavity', 'mean c
ompactness', 'worst compactness', 'worst area','worst perimeter','mean peri
meter', 'mean radius', 'radius error', 'perimeter error', 'worst concave po
ints', 'worst concavity', 'worst smoothness', 'compactness error'], axis=1)
```

Here what we ended up with:

In [36]:
```
f,ax = plt.subplots(figsize=(9, 7))
sns.heatmap(df1.corr(), annot= True, linewidths= 0.3, linecolor= "red", fmt
= ".0%", ax= ax, cmap = 'coolwarm')
plt.show()
```



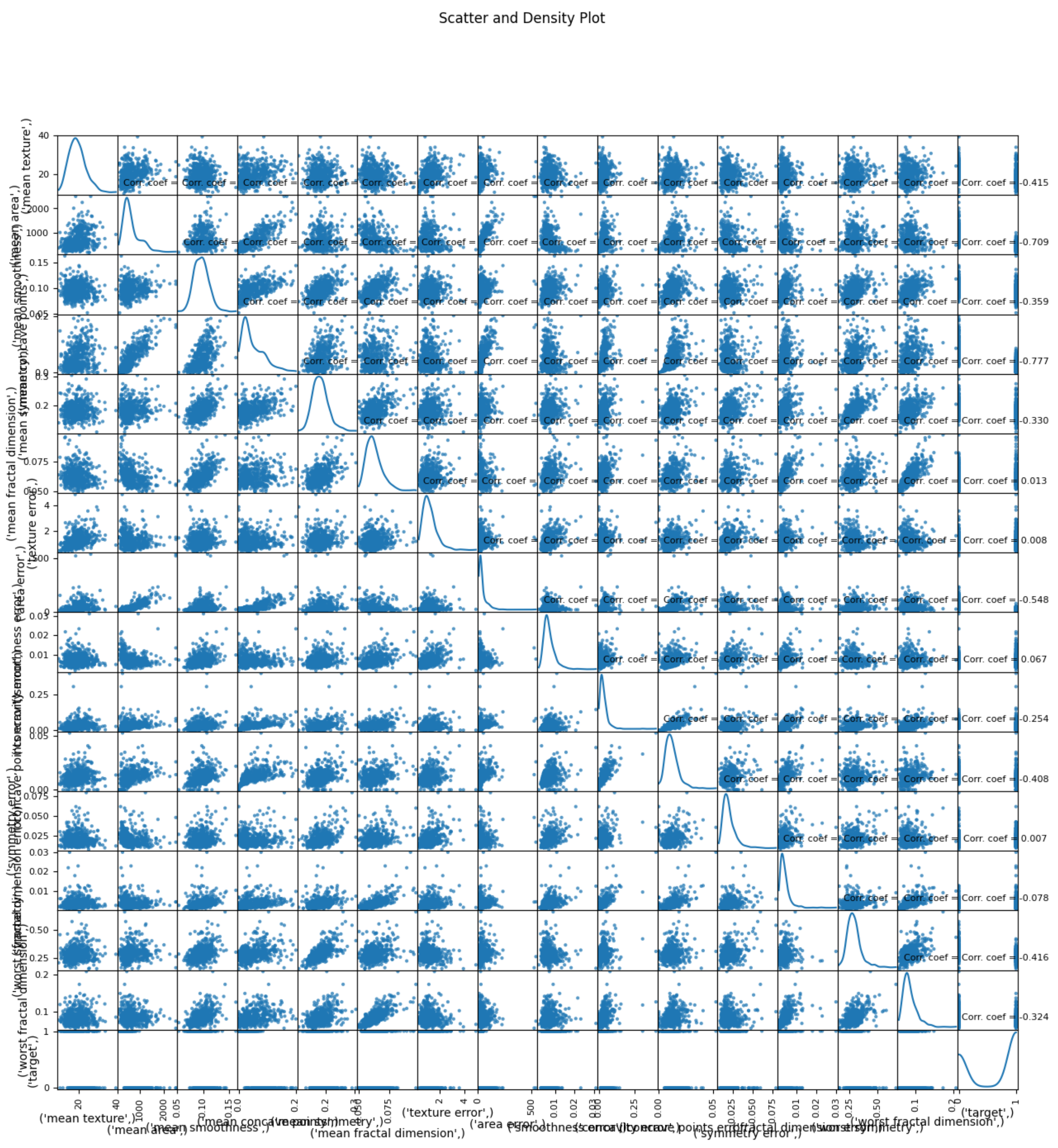
Let's create a Scatter- Density plot

In [9]:
```
#!pip install chart-studio
```

In [10]:
```
# Scatter and density plots
def plotScatterMatrix(df, plotSize, textSize):
    df = df.select_dtypes(include =[np.number]) # keep only numerical colum
ns
    # Remove rows and columns that would lead to df being singular
    df = df.dropna('columns')
    df = df[[col for col in df if df[col].nunique() > 1]] # keep columns wh
ere there are more than 1 unique values
    columnNames = list(df)
    if len(columnNames) > 40: # reduce the number of columns for matrix inv
ersion of kernel density plots
        columnNames = columnNames[:15]
    df = df[columnNames]
    ax = pd.plotting.scatter_matrix(df, alpha=0.75, figsize=[plotSize, plot
Size], diagonal='kde')
    corrs = df.corr().values
    for i, j in zip(*plt.np.triu_indices_from(ax, k = 1)):
        ax[i, j].annotate('Corr. coef = %.3f' % corrs[i, j], (0.8, 0.2), xy
coords='axes fraction', ha='center', va='center', size=textSize)
    plt.suptitle('Scatter and Density Plot')
    plt.show()
```

```
In [11]: plotScatterMatrix(df1, 15, 8)
```

```
<ipython-input-10-3766b111cba6>:5: FutureWarning: In a future version of pa
ndas all arguments of DataFrame.dropna will be keyword-only.
  df = df.dropna('columns')
```



Most of variables have a normal distribution or a "positively skewed normal distribution" (Right-skewed). Better to consider robust methods for handling outliers or use techniques specifically designed for skewed data. Non-parametric methods or those based on ranks can be particularly useful when dealing with skewed data.

In [12]:
```
df1.head(2)
```

Out[12]:

	mean texture	mean area	mean smoothness	mean concave points	mean symmetry	mean fractal dimension	texture error	area error	smoo error
0	10.38	1001.0	0.11840	0.14710	0.2419	0.07871	0.9053	153.40	0.006
1	17.77	1326.0	0.08474	0.07017	0.1812	0.05667	0.7339	74.08	0.005

Let's normalize/standardize the features.

In [13]:
```
from sklearn.preprocessing import StandardScaler
# transform data
scaler = StandardScaler()
# transform data
df11 = scaler.fit_transform(df1.drop('target',axis=1))
#data = scaler.fit_transform(california_housing.data)
#target = scaler.fit_transform(california_housing.target.values.reshape(-1,
1))
```

```
<ipython-input-13-cb7bf0fb2e6c>:5: PerformanceWarning: dropping on a non-le
xsorted multi-index without a level parameter may impact performance.
  df11 = scaler.fit_transform(df1.drop('target',axis=1))
```

In [14]:
```
df11 = pd.DataFrame(df11, columns=df1.drop('target',axis=1).columns)
```

```
<ipython-input-14-82851bfb1f18>:1: PerformanceWarning: dropping on a non-le
xsorted multi-index without a level parameter may impact performance.
  df11 = pd.DataFrame(df11, columns=df1.drop('target',axis=1).columns)
```

In [15]:
```
df11.head(2)
```

Out[15]:

	mean texture	mean area	mean smoothness	mean concave points	mean symmetry	mean fractal dimension	texture error	area error
0	-2.073335	0.984375	1.568466	2.532475	2.217515	2.255747	-0.565265	2.487
1	-0.353632	1.908708	-0.826962	0.548144	0.001392	-0.868652	-0.876244	0.742

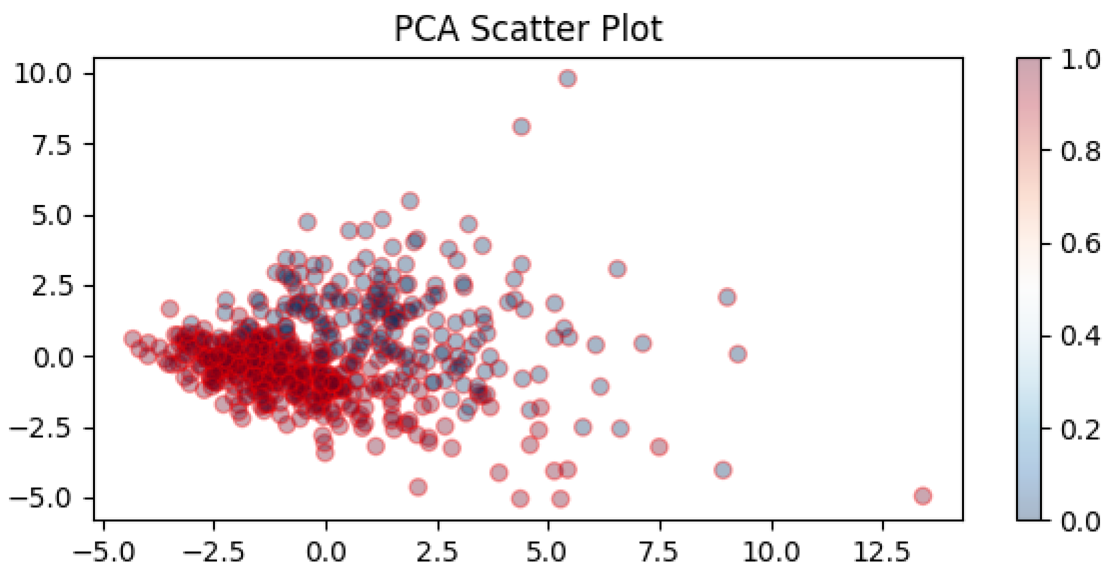
In [16]:
```
features_mean = ['mean texture','mean area', 'mean smoothness', 'mean conca
ve points', 'mean fractal dimension', 'texture error', 'area error', 'conca
vity error', 'concave points error', 'symmetry error', 'fractal dimension e
rror', 'worst symmetry', 'worst fractal dimension']
#df11 = pd.DataFrame(df11, columns=df1.columns)
data_drop = df11
#data_drop = data_drop[features_mean]
```

```
In [57]: cancer.feature_names
```

```
Out[57]: array(['mean radius', 'mean texture', 'mean perimeter', 'mean area',
                'mean smoothness', 'mean compactness', 'mean concavity',
                'mean concave points', 'mean symmetry', 'mean fractal dimension',
                'radius error', 'texture error', 'perimeter error', 'area error',
                'smoothness error', 'compactness error', 'concavity error',
                'concave points error', 'symmetry error',
                'fractal dimension error', 'worst radius', 'worst texture',
                'worst perimeter', 'worst area', 'worst smoothness',
                'worst compactness', 'worst concavity', 'worst concave points',
                'worst symmetry', 'worst fractal dimension'], dtype='<U23')
```

```
In [38]: from sklearn.decomposition import PCA

         X = data_drop.values
         pca = PCA(n_components=2) #Binary Classifier
         pca = pca.fit_transform(X)
         plt.figure(figsize = (7,3))
         plt.scatter(pca[:,0],pca[:,1], c = df1['target'], cmap = "RdBu_r", edgecolo
         r = "Red", alpha=0.35)
         plt.colorbar()
         plt.title('PCA Scatter Plot')
```

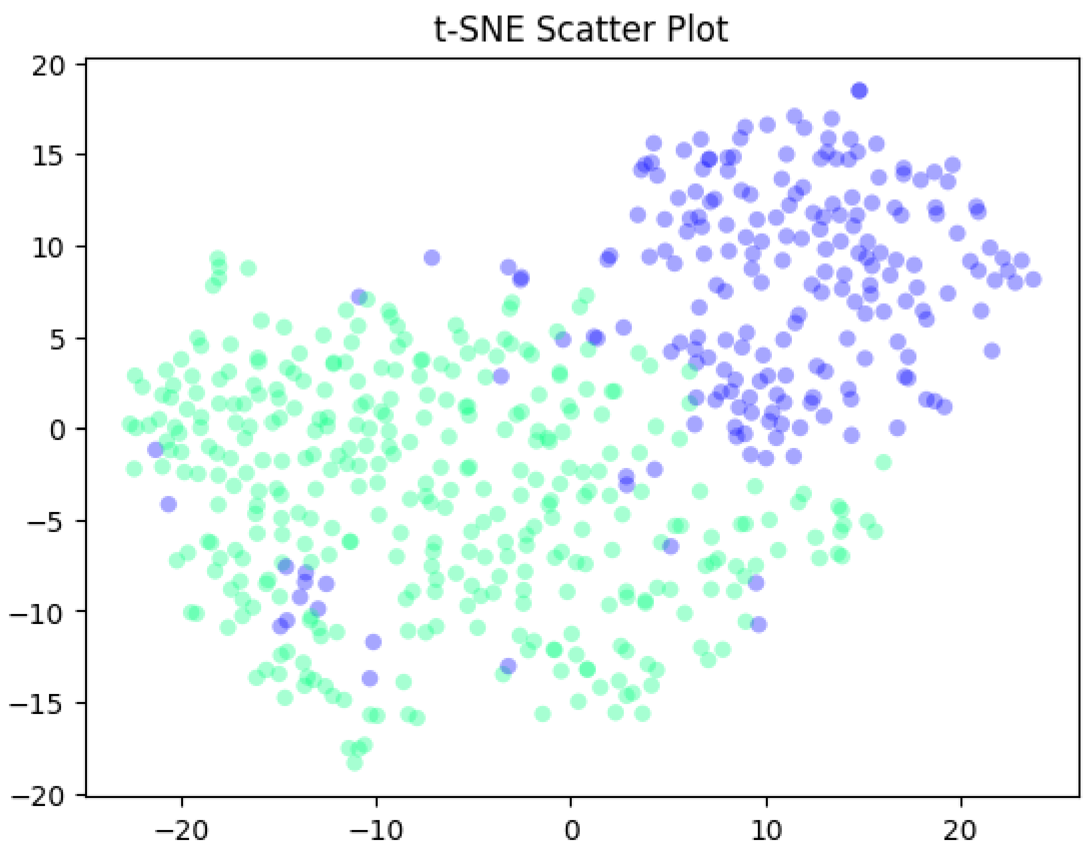
```
Out[38]: Text(0.5, 1.0, 'PCA Scatter Plot')
```



```
In [18]: # Principal component analysis and it's Scatter Plot
         from sklearn.manifold import TSNE
         tsne = TSNE(verbose=1, perplexity=40, n_iter= 4000)
         tsne = tsne.fit_transform(X)
         plt.scatter(tsne[:,0],tsne[:,1],  c = df1['target'], cmap = "winter", edgec
         olor = "None", alpha=0.35)
         plt.title('t-SNE Scatter Plot')
```

```
[t-SNE] Computing 121 nearest neighbors...
[t-SNE] Indexed 569 samples in 0.002s...
[t-SNE] Computed neighbors for 569 samples in 0.035s...
[t-SNE] Computed conditional probabilities for sample 569 / 569
[t-SNE] Mean sigma: 1.145660
[t-SNE] KL divergence after 250 iterations with early exaggeration: 59.1392
56
[t-SNE] KL divergence after 2350 iterations: 0.939384
```

```
Out[18]: Text(0.5, 1.0, 't-SNE Scatter Plot')
```



With Principal component analysis, We can see a strong split on two target classes even using just two components. In general we could use just two components to conduct any future predictions.

Split data on Training and Test datasets

```
In [20]: from sklearn.model_selection import train_test_split
         X = data_drop
         yy = df1['target']
         y = yy.values.ravel()
         # Assuming X and y are your preprocessed feature matrix and target variable
         X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, ra
         ndom_state=42)
```

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.

In [21]:
```
from sklearn.svm import SVC
from sklearn.metrics import accuracy_score, precision_score, recall_score,
f1_score, roc_auc_score, confusion_matrix
```

In [22]:
```
# SVM with Linear Kernel
svm_linear = SVC(kernel='linear', C=1.0, probability=True)
svm_linear.fit(X_train, y_train)

# SVM with Polynomial Kernel
svm_poly = SVC(kernel='poly', degree=3, C=1.0, probability=True)
svm_poly.fit(X_train, y_train)

# SVM with Radial Basis Function (RBF) Kernel
svm_rbf = SVC(kernel='rbf', C=1.0, probability=True)
svm_rbf.fit(X_train, y_train)

# Make predictions on the test set
y_pred_linear = svm_linear.predict(X_test)
y_prob_linear = svm_linear.predict_proba(X_test)[:, 1]

y_pred_poly = svm_poly.predict(X_test)
y_prob_poly = svm_poly.predict_proba(X_test)[:, 1]

y_pred_rbf = svm_rbf.predict(X_test)
y_prob_rbf = svm_rbf.predict_proba(X_test)[:, 1]
```

In [23]:
```python
# Evaluate the models
def evaluate_model(y_true, y_pred, y_prob, model_name):
    accuracy = accuracy_score(y_true, y_pred)
    precision = precision_score(y_true, y_pred)
    recall = recall_score(y_true, y_pred)
    f1 = f1_score(y_true, y_pred)
    roc_auc = roc_auc_score(y_true, y_prob)

    print(f"Evaluation Metrics for {model_name}:")
    print(f"Accuracy: {accuracy:.4f}")
    print(f"Precision: {precision:.4f}")
    print(f"Recall: {recall:.4f}")
    print(f"F1-Score: {f1:.4f}")
    print(f"ROC-AUC: {roc_auc:.4f}")
    print("Confusion Matrix:")
    print(confusion_matrix(y_true, y_pred))
    print("\n")

# Evaluate Linear Kernel SVM
evaluate_model(y_test, y_pred_linear, y_prob_linear, "Linear Kernel SVM")

# Evaluate Polynomial Kernel SVM
evaluate_model(y_test, y_pred_poly, y_prob_poly, "Polynomial Kernel SVM")

# Evaluate RBF Kernel SVM
evaluate_model(y_test, y_pred_rbf, y_prob_rbf, "RBF Kernel SVM")
```

```
Evaluation Metrics for Linear Kernel SVM:
Accuracy: 0.9474
Precision: 0.9577
Recall: 0.9577
F1-Score: 0.9577
ROC-AUC: 0.9948
Confusion Matrix:
[[40  3]
 [ 3 68]]


Evaluation Metrics for Polynomial Kernel SVM:
Accuracy: 0.8509
Precision: 0.8068
Recall: 1.0000
F1-Score: 0.8931
ROC-AUC: 0.9895
Confusion Matrix:
[[26 17]
 [ 0 71]]


Evaluation Metrics for RBF Kernel SVM:
Accuracy: 0.9649
Precision: 0.9718
Recall: 0.9718
F1-Score: 0.9718
ROC-AUC: 0.9934
Confusion Matrix:
[[41  2]
 [ 2 69]]
```

The Best performed model with Radial Basis Function (RBF) Kernel.

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.

Random Forest

In [24]:
```
from sklearn.model_selection import train_test_split, GridSearchCV
from sklearn.ensemble import RandomForestClassifier, GradientBoostingClassi
fier
from sklearn.metrics import accuracy_score, precision_score, recall_score,
f1_score, roc_auc_score, confusion_matrix
```

In [25]:
```
# Random Forest
rf_classifier = RandomForestClassifier(random_state=42)

# Define hyperparameters to tune
rf_param_grid = {
    'n_estimators': [40, 50, 100],
    'max_depth': [4, 5, 10, 20],
    'min_samples_split': [2, 3, 5, 10],
    'min_samples_leaf': [1, 2, 4],
}

# GridSearchCV for hyperparameter tuning
rf_grid_search = GridSearchCV(rf_classifier, param_grid=rf_param_grid, cv=
5, scoring='accuracy')
rf_grid_search.fit(X_train, y_train)

# Get the best parameters
best_rf_params = rf_grid_search.best_params_

# Train the Random Forest model with the best parameters
best_rf_model = RandomForestClassifier(**best_rf_params, random_state=42)
best_rf_model.fit(X_train, y_train)

# Make predictions on the test set
rf_predictions = best_rf_model.predict(X_test)

# Evaluate the Random Forest model
rf_accuracy = accuracy_score(y_test, rf_predictions)
rf_precision = precision_score(y_test, rf_predictions)
rf_recall = recall_score(y_test, rf_predictions)
rf_f1 = f1_score(y_test, rf_predictions)
rf_roc_auc = roc_auc_score(y_test, best_rf_model.predict_proba(X_test)[:,
1])

# Print metrics for Random Forest
print("Random Forest Metrics:")
print("Accuracy:", rf_accuracy)
print("Precision:", rf_precision)
print("Recall:", rf_recall)
print("F1-Score:", rf_f1)
print("ROC-AUC:", rf_roc_auc)
```

```
Random Forest Metrics:
Accuracy: 0.9385964912280702
Precision: 0.9324324324324325
Recall: 0.971830985915493
F1-Score: 0.9517241379310345
ROC-AUC: 0.9919751064526695
```

In [26]:
```
best_rf_params
```

Out[26]:
```
{'max_depth': 20,
 'min_samples_leaf': 1,
 'min_samples_split': 2,
 'n_estimators': 40}
```

Gradient Boosting

In [28]:
```
# Gradient Boosting
gb_classifier = GradientBoostingClassifier(random_state=42)

# Define hyperparameters to tune
gb_param_grid = {
    'n_estimators': [100, 200, 300],
    'learning_rate': [0.1, 0.2, 0.3],
    'max_depth': [3,4, 5],
    'min_samples_split': [5, 10, 12],
    'min_samples_leaf': [2, 4, 5],
}

# GridSearchCV for hyperparameter tuning
gb_grid_search = GridSearchCV(gb_classifier, param_grid=gb_param_grid, cv=
5, scoring='accuracy')
gb_grid_search.fit(X_train, y_train)

# Get the best parameters
best_gb_params = gb_grid_search.best_params_

# Train the Gradient Boosting model with the best parameters
best_gb_model = GradientBoostingClassifier(**best_gb_params, random_state=4
2)
best_gb_model.fit(X_train, y_train)

# Make predictions on the test set
gb_predictions = best_gb_model.predict(X_test)

# Evaluate the Gradient Boosting model
gb_accuracy = accuracy_score(y_test, gb_predictions)
gb_precision = precision_score(y_test, gb_predictions)
gb_recall = recall_score(y_test, gb_predictions)
gb_f1 = f1_score(y_test, gb_predictions)
gb_roc_auc = roc_auc_score(y_test, best_gb_model.predict_proba(X_test)[:,
1])

# Print metrics for Gradient Boosting
print("\nGradient Boosting Metrics:")
print("Accuracy:", gb_accuracy)
print("Precision:", gb_precision)
print("Recall:", gb_recall)
print("F1-Score:", gb_f1)
print("ROC-AUC:", gb_roc_auc)
```

```
Gradient Boosting Metrics:
Accuracy: 0.956140350877193
Precision: 0.9583333333333334
Recall: 0.971830985915493
F1-Score: 0.965034965034965
ROC-AUC: 0.9859154929577465
```

In [29]:
```
best_gb_params
```

Out[29]:
```
{'learning_rate': 0.2,
 'max_depth': 3,
 'min_samples_leaf': 4,
 'min_samples_split': 10,
 'n_estimators': 300}
```

4. Classification with Generalized Additive Models (GAMs)

In [30]:
```
!pip install pygam
```

```
Collecting pygam
  Downloading pygam-0.9.0-py3-none-any.whl (522 kB)
     ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 522.2/522.2 kB 7.7 MB/s eta
0:00:00
Collecting numpy<2.0.0,>=1.24.2 (from pygam)
  Downloading numpy-1.26.2-cp310-cp310-manylinux_2_17_x86_64.manylinux2014_
x86_64.whl (18.2 MB)
     ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 18.2/18.2 MB 54.9 MB/s eta 0:
00:00
Requirement already satisfied: progressbar2<5.0.0,>=4.2.0 in /usr/local/li
b/python3.10/dist-packages (from pygam) (4.2.0)
Requirement already satisfied: scipy<2.0.0,>=1.10.1 in /usr/local/lib/pytho
n3.10/dist-packages (from pygam) (1.11.3)
Requirement already satisfied: python-utils>=3.0.0 in /usr/local/lib/python
3.10/dist-packages (from progressbar2<5.0.0,>=4.2.0->pygam) (3.8.1)
Requirement already satisfied: typing-extensions>3.10.0.2 in /usr/local/li
b/python3.10/dist-packages (from python-utils>=3.0.0->progressbar2<5.0.0,>=
4.2.0->pygam) (4.5.0)
Installing collected packages: numpy, pygam
  Attempting uninstall: numpy
    Found existing installation: numpy 1.23.5
    Uninstalling numpy-1.23.5:
      Successfully uninstalled numpy-1.23.5
ERROR: pip's dependency resolver does not currently take into account all t
he packages that are installed. This behaviour is the source of the followi
ng dependency conflicts.
lida 0.0.10 requires fastapi, which is not installed.
lida 0.0.10 requires kaleido, which is not installed.
lida 0.0.10 requires python-multipart, which is not installed.
lida 0.0.10 requires uvicorn, which is not installed.
cupy-cuda11x 11.0.0 requires numpy<1.26,>=1.20, but you have numpy 1.26.2 w
hich is incompatible.
Successfully installed numpy-1.26.2 pygam-0.9.0
```

The link function connects the linear predictor (obtained from the sum of the model's predictors) to the expected value of the response variable. In the case of binary classification, where the response variable is binary (0 or 1), the commonly used link function is the logit function.

The distribution family specifies the type of distribution assumed for the response variable. In binary classification, the response variable is binary, representing two possible outcomes (e.g., 0 or 1, True or False). The binomial distribution is commonly used for binary classification tasks. It models the number of successes in a fixed number of independent Bernoulli trials. In the context of logistic regression (a type of GAM for binary classification), the binomial distribution is appropriate. For other types of response variables or tasks, different distribution families might be used. For example, Gaussian distribution for regression tasks, Poisson distribution for count data, etc.

The choice of link function and distribution family depends on the characteristics of your data. The default link function for binary classification in pyGAM is **the logit link**, which is suitable for logistic regression. The default distribution family is **the binomial distribution.** These choices are often suitable and align with logistic regression.

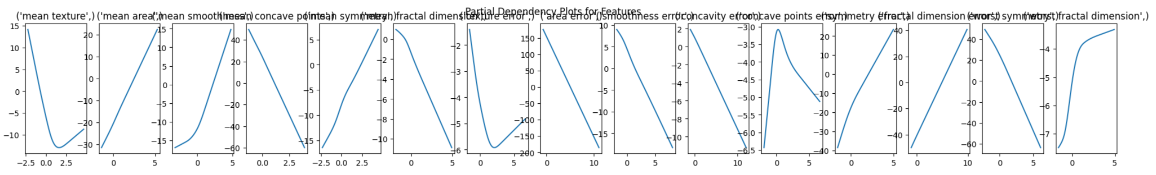
In [31]:
```
from pygam import LogisticGAM

# Fit a Generalized Additive Model
gam_model = LogisticGAM().fit(X_train, y_train)
```

```
/usr/local/lib/python3.10/dist-packages/pygam/links.py:151: RuntimeWarning:
divide by zero encountered in divide
  return dist.levels / (mu * (dist.levels - mu))
/usr/local/lib/python3.10/dist-packages/pygam/pygam.py:629: RuntimeWarning:
invalid value encountered in multiply
  self.link.gradient(mu, self.distribution) ** 2
/usr/local/lib/python3.10/dist-packages/pygam/pygam.py:629: RuntimeWarning:
overflow encountered in square
  self.link.gradient(mu, self.distribution) ** 2
```

Partial Dependency Plots are useful for interpreting complex machine learning models, such as Generalized Additive Models (GAMs), Random Forests, or Gradient Boosted Trees. They allow you to understand the marginal effect of individual features on the model's predictions, providing insights into the model's behavior. Partial Dependency Plots (PDPs) are graphical tools used in machine learning to visualize the relationship between a feature (predictor variable) and the predicted outcome of a model while keeping other features constant.

In the context of a GAM, partial dependency plots visualize the contribution of each feature to the model's predictions while considering the non-linear effects captured by the model. These plots are valuable for explaining the model's behavior to stakeholders and gaining insights into the relationships between features and outcomes.

```
In [32]: # Visualize partial dependency plots for each feature
         fig, axs = plt.subplots(1, X_train.shape[1], figsize=(25, 3))
         titles = X_train.columns
         for i, ax in enumerate(axs):
             XX = gam_model.generate_X_grid(term=i)
             ax.plot(XX[:, i], gam_model.partial_dependence(term=i, X=XX))
             ax.set_title(titles[i])

         plt.suptitle("Partial Dependency Plots for Features")
         plt.show()
```



A target variable has a linear relationship with the most variables.

In general, logistic regression is more powerful and interpretable model for linear relationships in binary classification, while GAMs offer enhanced flexibility to capture non-linear patterns. The choice between them depends on the nature of the data and the complexity of relationships.

In our case there are no much non-linearities and, therefore, we dont need to capture a complex relationships with GAM model.

```
In [33]: # Make predictions on the test set
         gam_pred_prob = gam_model.predict_proba(X_test)
         gam_pred = np.round(gam_pred_prob)

         # Evaluate the model
         accuracy = accuracy_score(y_test, gam_pred)
         precision = precision_score(y_test, gam_pred)
         recall = recall_score(y_test, gam_pred)
         f1 = f1_score(y_test, gam_pred)
         roc_auc_gam = roc_auc_score(y_test, gam_pred_prob)

         # Print metrics
         print("Accuracy:", accuracy)
         print("Precision:", precision)
         print("Recall:", recall)
         print("F1-Score:", f1)
         print("ROC-AUC:", roc_auc_gam)
```

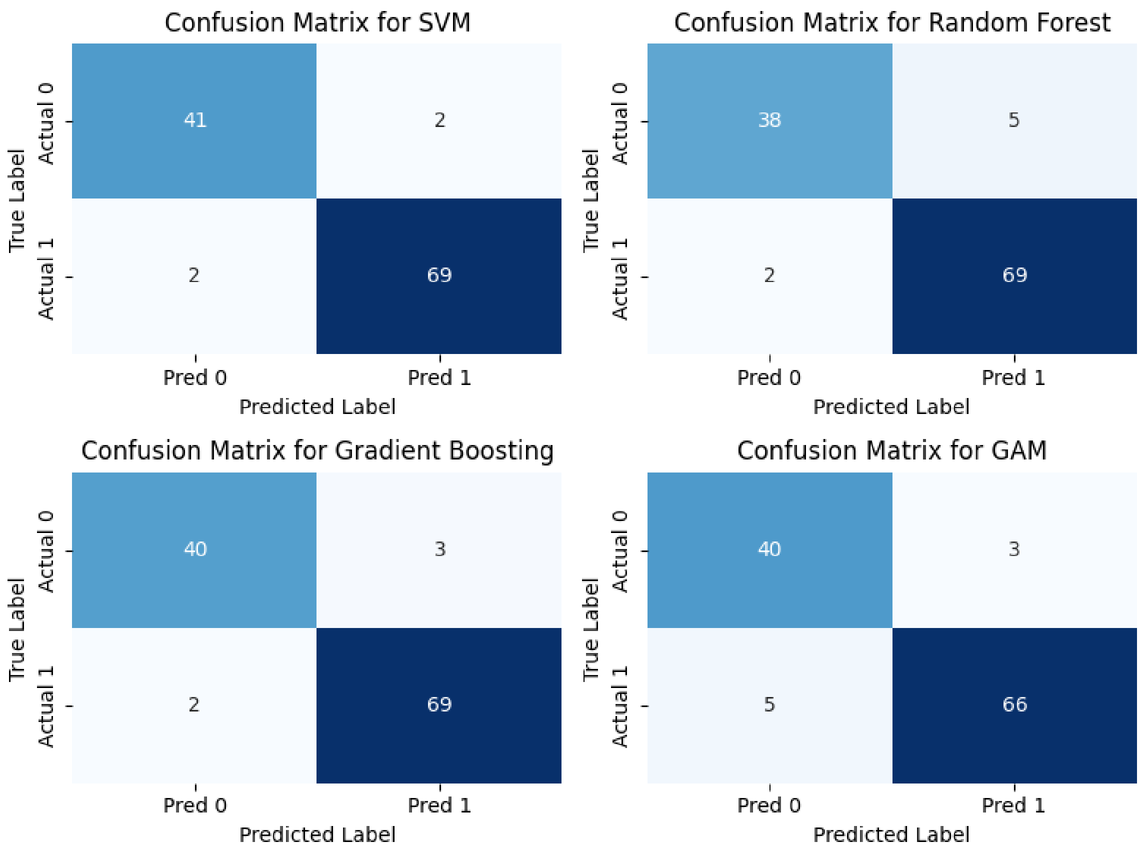
```
Accuracy: 0.9298245614035088
Precision: 0.9565217391304348
Recall: 0.9295774647887324
F1-Score: 0.9428571428571428
ROC-AUC: 0.9882083196855551
```

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.

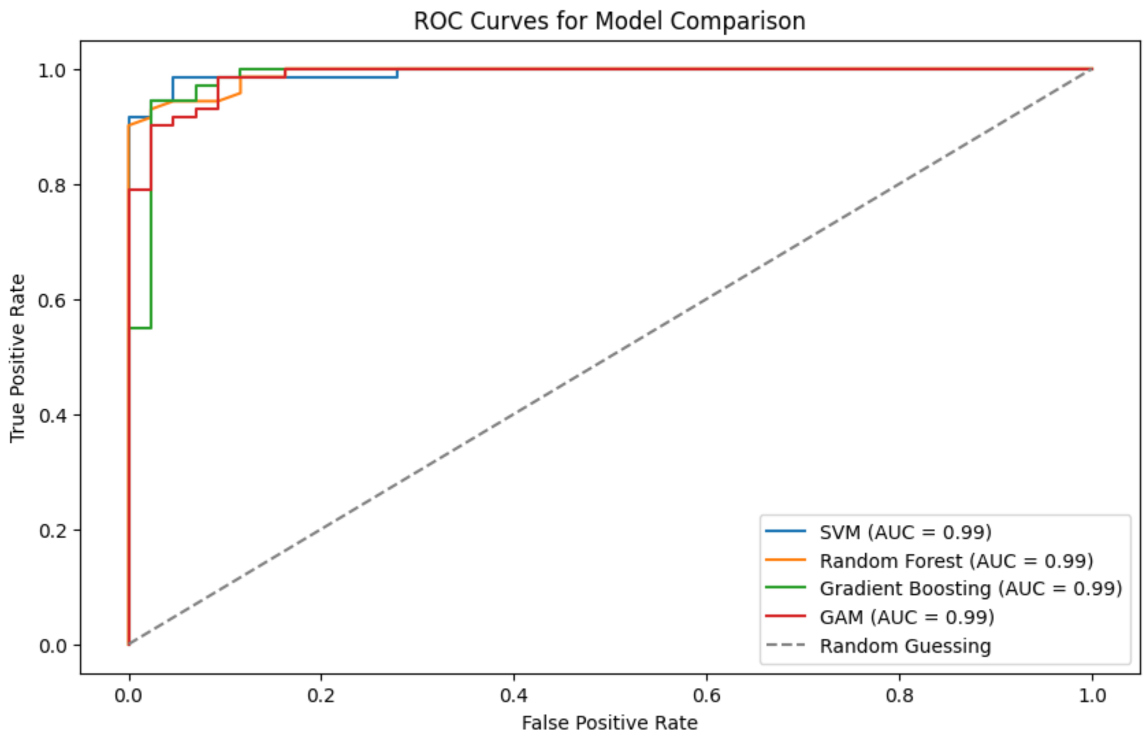
In [69]:
```
from sklearn.metrics import confusion_matrix, roc_curve, roc_auc_score

# Confusion SVD matrix
svdc = confusion_matrix(y_test, y_pred_rbf)
# Confusion RF matrix
cmf = confusion_matrix(y_test, rf_predictions)
# Confusion GB matrix
cm = confusion_matrix(y_test, gb_predictions)
# Confusion GAM Matrix
gam_cm = confusion_matrix(y_test, gam_pred)
```

In [67]:
```python
import seaborn as sns
import matplotlib.pyplot as plt

# Define a function to plot confusion matrix
def plot_confusion_matrix(ax, cm, model_name):
    sns.heatmap(cm, annot=True, fmt="d", cmap="Blues", cbar=False,
                xticklabels=["Pred 0", "Pred 1"],
                yticklabels=["Actual 0", "Actual 1"], ax=ax)
    ax.set_xlabel("Predicted Label")
    ax.set_ylabel("True Label")
    ax.set_title(f"Confusion Matrix for {model_name}")

# Create a 2x2 grid of subplots
fig, axes = plt.subplots(nrows=2, ncols=2, figsize=(8, 6))

# Plot confusion matrices
plot_confusion_matrix(axes[0, 0], svdc, "SVM")
plot_confusion_matrix(axes[0, 1], cmf, "Random Forest")
plot_confusion_matrix(axes[1, 0], cm, "Gradient Boosting")
plot_confusion_matrix(axes[1, 1], gam_cm, "GAM")

# Adjust layout
plt.tight_layout()
plt.show()
```



Let's plot ROC curve graphs for all four models

In [41]:
```
# ROC SVM RBF Curve
y_prob_rbf = svm_rbf.predict_proba(X_test)[:, 1]
fpr_svm, tpr_svm, _ = roc_curve(y_test, y_prob_rbf)
roc_auc_svm = roc_auc_score(y_test, y_prob_rbf)

# ROC Random Forest Curve
y_prob_rf = best_rf_model.predict_proba(X_test)[:, 1]
fpr_rf, tpr_rf, _ = roc_curve(y_test, y_prob_rf)
roc_auc_rf = roc_auc_score(y_test, y_prob_rf)

# ROC Gradient Boosting Curve
gb_prob = best_gb_model.predict_proba(X_test)[:, 1]
fpr_gb, tpr_gb, _ = roc_curve(y_test, gb_prob)
roc_auc_gb = roc_auc_score(y_test, gb_prob)

# ROC GAM Curve
gam_prob = gam_model.predict_proba(X_test)
fpr_gam, tpr_gam, _ = roc_curve(y_test, gam_prob)
roc_auc_gam = roc_auc_score(y_test, gam_prob)

# Plotting ROC Curves
plt.figure(figsize=(10, 6))
plt.plot(fpr_svm, tpr_svm, label=f'SVM (AUC = {roc_auc_svm:.2f})')
plt.plot(fpr_rf, tpr_rf, label=f'Random Forest (AUC = {roc_auc_rf:.2f})')
plt.plot(fpr_gb, tpr_gb, label=f'Gradient Boosting (AUC = {roc_auc_gb:.2
f})')
plt.plot(fpr_gam, tpr_gam, label=f'GAM (AUC = {roc_auc_gam:.2f})')

plt.plot([0, 1], [0, 1], linestyle='--', color='gray', label='Random Guessi
ng')
plt.xlabel('False Positive Rate')
plt.ylabel('True Positive Rate')
plt.title('ROC Curves for Model Comparison')
plt.legend()
plt.show()
```



SVM:

Strengths: Effective in high-dimensional spaces, good generalization.

Weaknesses: Computationally intensive, sensitive to choice of kernel.

Random Forest:

Strengths: Robust, handles non-linearities well, feature importance.

Weaknesses: Can be prone to overfitting, less interpretable.

Gradient Boosting

Strengths: Robust, handles non-linearities well, feature importance.

Weaknesses: Computationally expensive, can be prone to overfitting, less interpretable.

GAM:

Strengths: Captures non-linear relationships, interpretable.

Weaknesses: Limited to additive effects, may not handle complex interactions.

In our case SVM with Radial Basis Function (RBF) Kernel overperformed all other models. On the second place is Gradient Boosting.

SVM with Radial Basis Function (RBF) Kernel: Strengths:

Effective in High-Dimensional Spaces:

SVM, especially with the RBF kernel, is effective in high-dimensional spaces. It is well-suited for tasks with a large number of features. Non-Linear Decision Boundaries:

The RBF kernel allows SVM to model complex, non-linear decision boundaries, making it suitable for datasets with intricate relationships. Good Generalization:

SVM with RBF kernel tends to generalize well to unseen data, making it robust in various applications. Robust to Overfitting:

SVMs, in general, are less prone to overfitting, and the regularization parameter (C) in the RBF kernel SVM allows control over the trade-off between fitting the training data and promoting a smooth decision boundary. Effective for Small to Medium-Sized Datasets:

SVMs, including those with RBF kernels, can perform well on small to medium-sized datasets. Weaknesses:

Computationally Intensive:

Training SVMs, especially with non-linear kernels like RBF, can be computationally intensive, making it less suitable for very large datasets. Sensitive to Hyperparameters:

The performance of SVM with RBF kernel is sensitive to the choice of hyperparameters, including the regularization parameter (C) and the kernel parameter (γ). Black Box Model:

SVMs, particularly with complex kernels, are often considered as black box models, providing less insight into the inner workings compared to simpler models. Difficult to Interpret:

While SVMs provide a decision boundary, interpreting the model's parameters in a meaningful way can be challenging. Memory Requirements:

SVMs may have higher memory requirements, especially when using a large number of support vectors, which can impact scalability. In summary, SVM with Radial Basis Function (RBF) Kernel is a powerful algorithm for non-linear classification tasks, but it comes with computational costs and sensitivity to hyperparameters. It is particularly well-suited for tasks where non-linear decision boundaries are essential and the dataset is not extremely large.