ML_HW3-Final

February 12, 2025

1 Machine Learning - Homework 3

Author: Shivani Tayade, Sravya Bhaskara

Part A:

1.0.1 Bagging and Boosting on California Housing DataSet

[4]:
```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns
import sklearn
```

[5]:
```
from sklearn.ensemble import RandomForestRegressor, GradientBoostingRegressor
```

[6]:
```
from sklearn.datasets import fetch_california_housing
```

[7]:
```
#help(sklearn)
```

[8]:
```
from sklearn import model_selection
```

[9]:
```
#help(sklearn.model_selection)
```

[10]:
```
#help(sklearn.model_selection._split)
```

[11]:
```
from sklearn.model_selection import GridSearchCV, RandomizedSearchCV,␣
↪train_test_split
```

[12]:
```
from sklearn.metrics import mean_absolute_error, mean_squared_error, r2_score
```

[13]:
```
df = fetch_california_housing()
```

[14]:
```
X = pd.DataFrame(df.data, columns=df.feature_names)
X.columns
X.to_csv("data")
```

[15]:
```
Y = df.target
Y
```

[15]:
```
array([4.526, 3.585, 3.521, …, 0.923, 0.847, 0.894])
```

EDA

[17]:
```
print(X.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
None
```

[18]:
```
num_nans = np.isnan(Y).sum()
print(f"Number of NaN values: {num_nans}")
```

```
Number of NaN values: 0
```

[19]:
```
nan_indices = np.where(np.isnan(Y))
print(f"Indices of NaN values: {nan_indices}")
```

```
Indices of NaN values: (array([], dtype=int64),)
```

[20]:
```
X.isnull().sum()
```

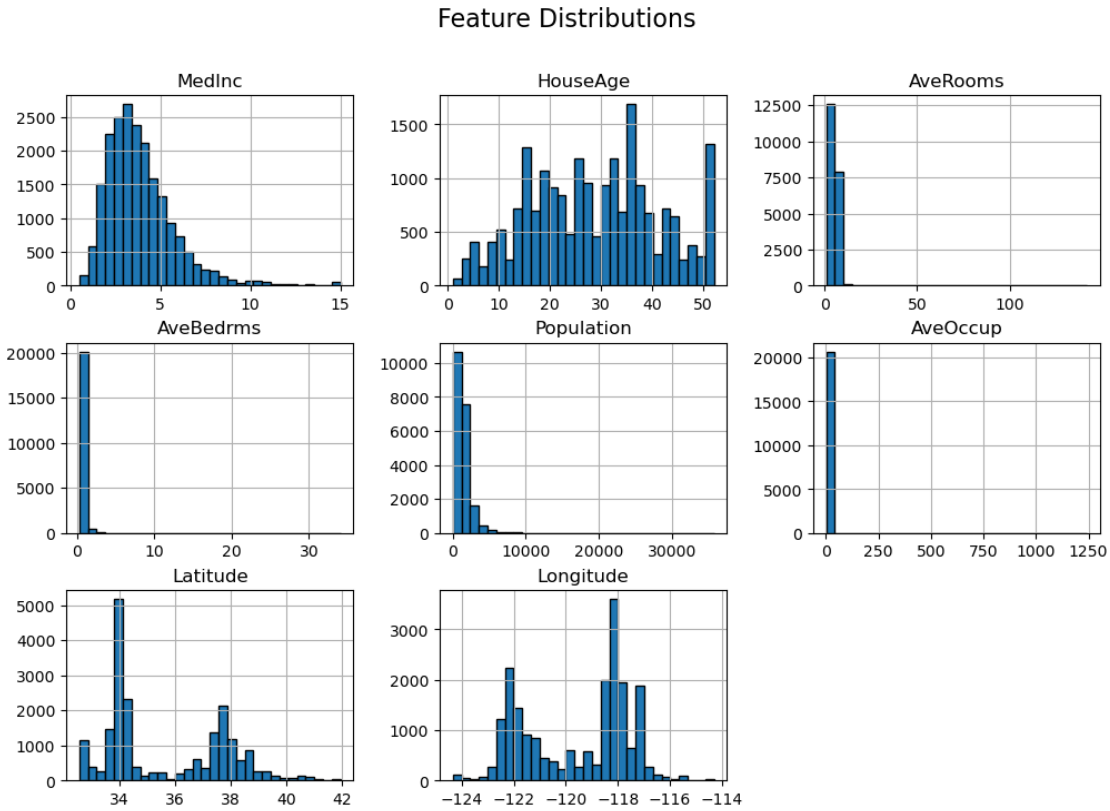
[20]:
```
MedInc        0
HouseAge      0
AveRooms      0
AveBedrms     0
Population    0
AveOccup      0
Latitude      0
Longitude     0
dtype: int64
```

[21]:
```
print("\nSummary Statistics:\n", X.describe())
```

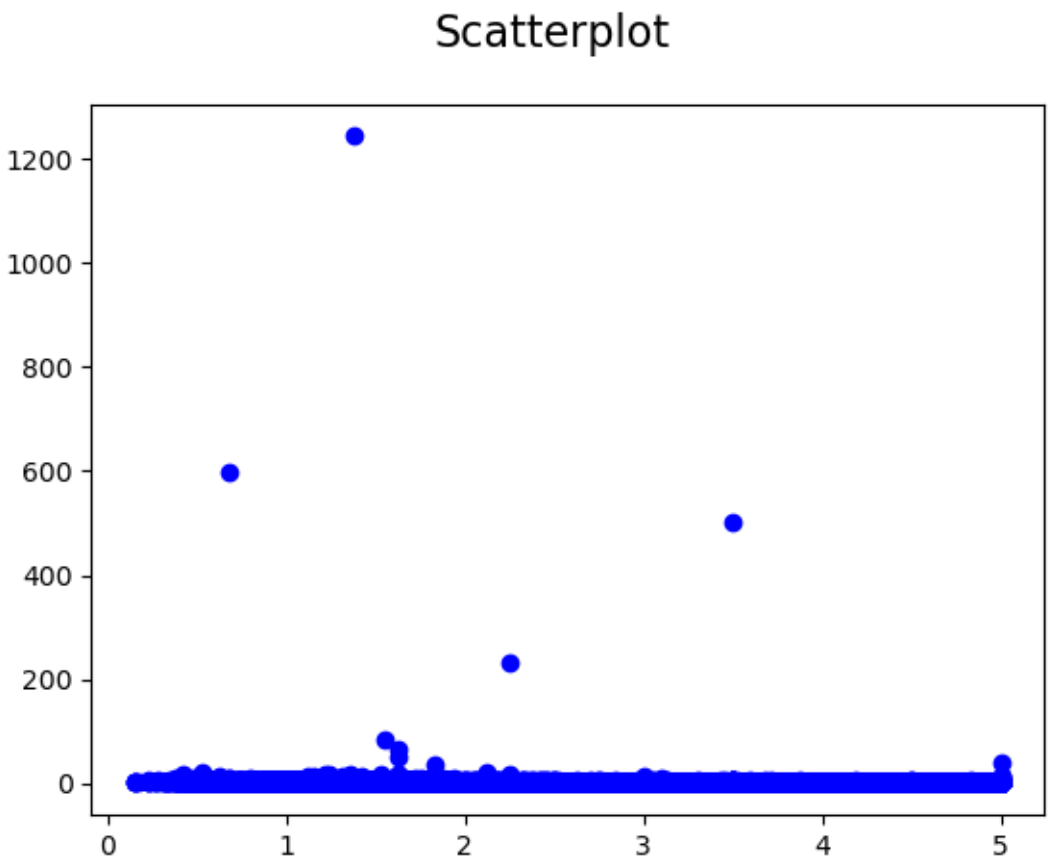
```
Summary Statistics:
              MedInc      HouseAge      AveRooms     AveBedrms    Population  \
count  20640.000000  20640.000000  20640.000000  20640.000000  20640.000000
mean       3.870671     28.639486      5.429000      1.096675   1425.476744
std        1.899822     12.585558      2.474173      0.473911   1132.462122
min        0.499900      1.000000      0.846154      0.333333      3.000000
25%        2.563400     18.000000      4.440716      1.006079    787.000000
50%        3.534800     29.000000      5.229129      1.048780   1166.000000
75%        4.743250     37.000000      6.052381      1.099526   1725.000000
max       15.000100     52.000000    141.909091     34.066667  35682.000000

           AveOccup      Latitude     Longitude
count  20640.000000  20640.000000  20640.000000
mean       3.070655     35.631861   -119.569704
std       10.386050      2.135952      2.003532
min        0.692308     32.540000   -124.350000
25%        2.429741     33.930000   -121.800000
50%        2.818116     34.260000   -118.490000
75%        3.282261     37.710000   -118.010000
max     1243.333333     41.950000   -114.310000
```

[22]:
```
plt.figure(figsize=(12, 8))
X.hist(figsize=(12, 8), bins=30, edgecolor='black')
plt.suptitle("Feature Distributions", fontsize=16)
plt.show()
```

```
<Figure size 1200x800 with 0 Axes>
```



[23]:
```
plt.scatter(Y, X["AveOccup"], color="blue")
plt.suptitle("Scatterplot", fontsize=16)
plt.show()
```



[24]:
```
#sns.pairplot(X, y_vars=Y, x_vars=X.columns[:-1], height=3, aspect=1)
```

[25]:
```
X.corr()
```

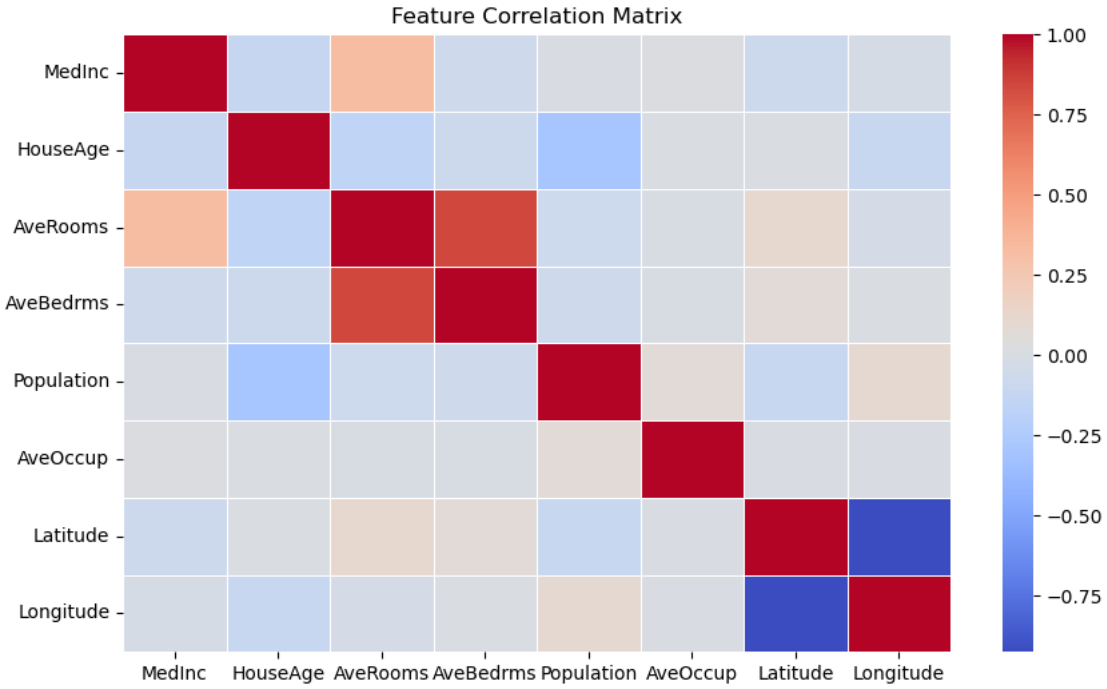
[25]:

	MedInc	HouseAge	AveRooms	AveBedrms	Population	AveOccup	Latitude	Longitude
MedInc	1.000000	-0.119034	0.326895	-0.062040	0.004834	0.018766	-0.079809	-0.015176
HouseAge	-0.119034	1.000000	-0.153277	-0.077747	-0.296244	0.013191	0.011173	-0.108197
AveRooms	0.326895	-0.153277	1.000000	0.847621	-0.072213	-0.004852	0.106389	-0.027540
AveBedrms	-0.062040	-0.077747	0.847621	1.000000	-0.066197	-0.006181	0.069721	0.013344
Population	0.004834	-0.296244	-0.072213	-0.066197	1.000000	0.069863		
AveOccup	0.018766	0.013191	-0.004852	-0.006181	0.069863	1.000000		
Latitude	-0.079809	0.011173	0.106389	0.069721	-0.108785	0.002366		
Longitude	-0.015176	-0.108197	-0.027540	0.013344	0.099773	0.002476		

```
Population -0.108785   0.099773
AveOccup    0.002366   0.002476
Latitude    1.000000  -0.924664
Longitude  -0.924664   1.000000
```

Correlation between "avebedrms" and "AveRooms" is 0.847621, and latitude and longitude also has -0.924664 of similarity b/w them. Hence we shall try to train the model dropping one of the variable. The sd for avebedrms is 2.474173. Hance we will consider this variable as the spread is more.

[27]:
```
plt.figure(figsize=(10, 6))
sns.heatmap(X.corr(), cmap="coolwarm", linewidths=0.5)
plt.title("Feature Correlation Matrix")
plt.show()
```



Model training on overall data

[29]:
```
X_train, X_test, y_train, y_test = train_test_split(X, Y, test_size=0.2,␣
↪random_state=42)
```

[30]:
```
# Initialize and train a Random Forest Regressor
rf = RandomForestRegressor(n_estimators=100, random_state=42)
rf.fit(X_train, y_train)
```

[30]:
```
RandomForestRegressor(random_state=42)
```

Default parameter

[32]:
```
rf_default_params = rf.get_params()
```

Predictions

[34]:
```
y_pred_rf = rf.predict(X_test)
```

Errror estimate

[36]:
```
rf_mae = mean_absolute_error(y_test, y_pred_rf)
rf_mse = mean_squared_error(y_test, y_pred_rf)
rf_r2 = r2_score(y_test, y_pred_rf)
```

[37]:
```
rf_default_params, rf_mae, rf_mse, rf_r2
```

[37]:
```
({'bootstrap': True,
  'ccp_alpha': 0.0,
  'criterion': 'squared_error',
  'max_depth': None,
  'max_features': 1.0,
  'max_leaf_nodes': None,
  'max_samples': None,
  'min_impurity_decrease': 0.0,
  'min_samples_leaf': 1,
  'min_samples_split': 2,
  'min_weight_fraction_leaf': 0.0,
  'n_estimators': 100,
  'n_jobs': None,
  'oob_score': False,
  'random_state': 42,
  'verbose': 0,
  'warm_start': False},
 0.32754256845930246,
 0.2553684927247781,
 0.8051230593157366)
```

Model training on data- considering Corr()

[39]:
```
X1 = X.drop(columns=["AveRooms","Longitude"])
```

[40]:
```
X1.head()
```

[40]:

	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

[41]:
```
X1_train, X1_test, y1_train, y1_test = train_test_split(X1, Y, test_size=0.2,␣
↪random_state=42)
```

[42]:
```
# Initialize and train a Random Forest Regressor
rf1 = RandomForestRegressor(n_estimators=100, random_state=42)
rf1.fit(X1_train, y1_train)
```

[42]:
```
RandomForestRegressor(random_state=42)
```

[43]:
```
rf_default_params1 = rf1.get_params()
```

[44]:
```
y1_pred_rf = rf1.predict(X1_test)
```

[45]:
```
rf_mae1 = mean_absolute_error(y1_test, y1_pred_rf)
rf_mse1 = mean_squared_error(y1_test, y1_pred_rf)
rf_r21 = r2_score(y1_test, y1_pred_rf)
```

[46]:
```
rf_default_params1, rf_mae1, rf_mse1, rf_r21
```

[46]:
```
({'bootstrap': True,
  'ccp_alpha': 0.0,
  'criterion': 'squared_error',
  'max_depth': None,
  'max_features': 1.0,
  'max_leaf_nodes': None,
  'max_samples': None,
  'min_impurity_decrease': 0.0,
  'min_samples_leaf': 1,
  'min_samples_split': 2,
  'min_weight_fraction_leaf': 0.0,
  'n_estimators': 100,
  'n_jobs': None,
  'oob_score': False,
  'random_state': 42,
  'verbose': 0,
  'warm_start': False},
 0.42139369590600795,
 0.37692628877253775,
 0.7123598090128124)
```

After dropping the variables the MAE, MSE has been increased to 42% and 38% compared to the 32% and 25% default parameter values. Hence, considering this we should not drop the variables.

1.1 Given the Std dev. we need to scale the variables and then fit the model

1.2 Formula for Robust Scaling

$$[X' = \frac{X - \text{median}(X)}{\text{IQR}(X)}]$$

$$[IQR = Q_3 - Q_1]$$

[50]:
```
from sklearn.preprocessing import RobustScaler
from sklearn.preprocessing import StandardScaler

# Initialize RobustScaler
scaler = RobustScaler()

# Apply transformation
df_scaled = X.copy()
df_scaled.iloc[:, :-1] = scaler.fit_transform(X.iloc[:, :-1])
 # View scaled data
df_scaled.head()
```

[50]:

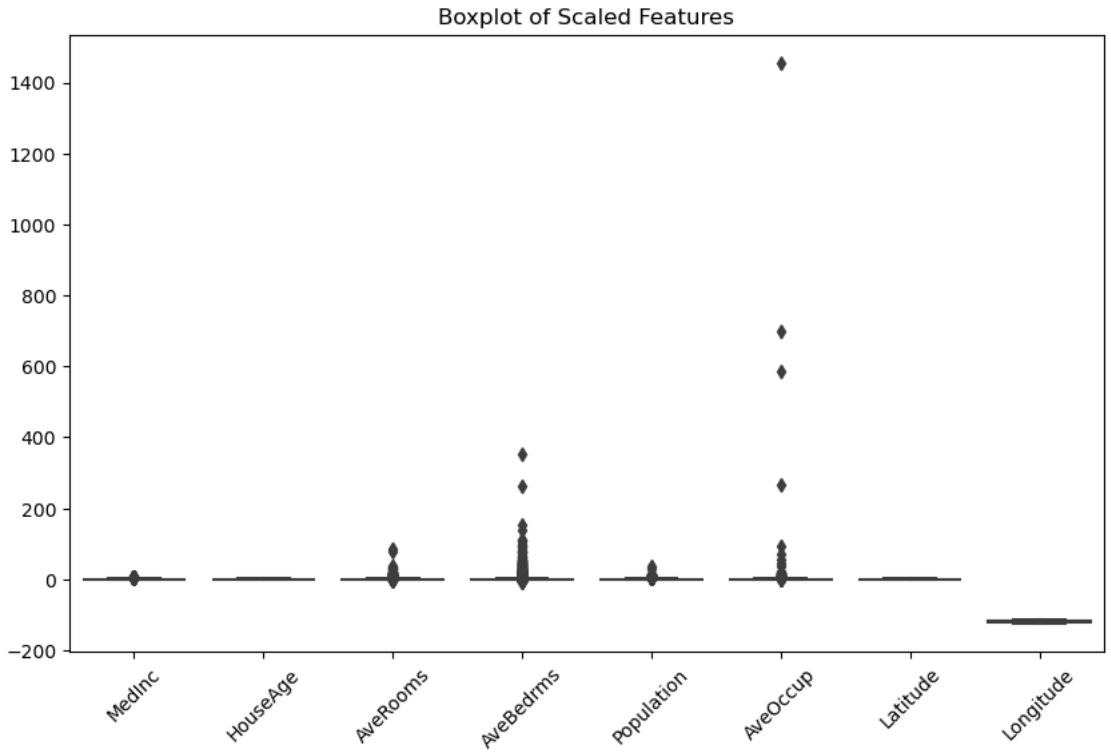
	MedInc	HouseAge	AveRooms	AveBedrms	Population	AveOccup	Latitude	Longitude
0	2.197582	0.631579	1.088935	-0.267221	-0.899787	-0.307981	0.957672	-122.23
1	2.186664	-0.421053	0.626066	-0.822926	1.316631	-0.830800	0.952381	-122.22
2	1.707732	1.210526	1.898042	0.263955	-0.714286	-0.018599	0.949735	-122.24
3	0.967177	1.210526	0.364978	0.259814	-0.648188	-0.316908	0.949735	-122.25
4	0.142854	1.210526	0.653191	0.345657	-0.640725	-0.746784	0.949735	-122.25

[51]:
```
print(df_scaled.describe())
```

	MedInc	HouseAge	AveRooms	AveBedrms	Population	AveOccup	Latitude	Longitude
count	2.064000e+04	20640.000000	20640.000000	20640.000000	20640.000000			
mean	1.540799e-01	-0.018974	0.124015	0.512533	0.276628			
std	8.715378e-01	0.662398	1.535166	5.071439	1.207316			
min	-1.392252e+00	-1.473684	-2.719533	-7.656179	-1.239872			
25%	-4.456270e-01	-0.578947	-0.489191	-0.456959	-0.404051			
50%	1.018608e-16	0.000000	0.000000	0.000000	0.000000			
75%	5.543730e-01	0.421053	0.510809	0.543041	0.595949			
max	5.259674e+00	1.210526	84.806698	353.332681	36.797441			

```
count  20640.000000  20640.000000  20640.000000
mean       0.296227      0.362926   -119.569704
std       12.182767      0.565067      2.003532
min       -2.493559     -0.455026   -124.350000
25%       -0.455561     -0.087302   -121.800000
50%        0.000000      0.000000   -118.490000
75%        0.544439      0.912698   -118.010000
max     1455.116059      2.034392   -114.310000
```

[52]:
```
plt.figure(figsize=(10, 6))
sns.boxplot(data=df_scaled)
plt.title("Boxplot of Scaled Features")
plt.xticks(rotation=45)
plt.show()
```



[53]:
```
Q1 = df_scaled.quantile(0.25)
Q3 = df_scaled.quantile(0.75)
IQR = Q3 - Q1

# Define lower and upper bounds
lower_bound = Q1 - 1.5 * IQR
upper_bound = Q3 + 1.5 * IQR
```

```
# Remove rows with outliers
df_scaled_no_outliers = df_scaled[~((df_scaled < lower_bound) | (df_scaled >␣
 ↪upper_bound)).any(axis=1)]

print("Original shape:", df_scaled.shape)
print("Shape after outlier removal:", df_scaled_no_outliers.shape)
```

```
Original shape: (20640, 8)
Shape after outlier removal: (16842, 8)
```

[54]:
```
from scipy.stats.mstats import winsorize

df_scaled = df_scaled.apply(lambda x: winsorize(x, limits=[0.05, 0.05]))  #␣
 ↪Capping top/bottom 5%
```

[55]:
```
from sklearn.model_selection import train_test_split

# Assuming 'Target' is the column we want to predict
X_final = df_scaled  # Adjust if needed
y_final = Y

X_train_scaled, X_test_scaled, y_train_scaled, y_test_scaled =␣
 ↪train_test_split(X_final, y_final, test_size=0.2, random_state=42)

print("Training set shape:", X_train_scaled.shape)
print("Test set shape:", X_test_scaled.shape)
```

```
Training set shape: (16512, 8)
Test set shape: (4128, 8)
```

[56]:
```
rf_scaled = RandomForestRegressor(n_estimators=100, random_state=42)
rf_scaled.fit(X_train_scaled, y_train_scaled)

y_pred_rf_scaled = rf_scaled.predict(X_test_scaled)

# Evaluate performance
mae_scaled = mean_absolute_error(y_test_scaled, y_pred_rf_scaled)
mse_scaled = mean_squared_error(y_test_scaled, y_pred_rf_scaled)
r2_scaled = r2_score(y_test_scaled, y_pred_rf_scaled)

print(f"Random Forest - MAE: {mae_scaled:.2f}, MSE: {mse_scaled:.2f}, R²:␣
 ↪{r2_scaled:.2f}")
```

```
Random Forest - MAE: 0.34, MSE: 0.26, R²: 0.80
```

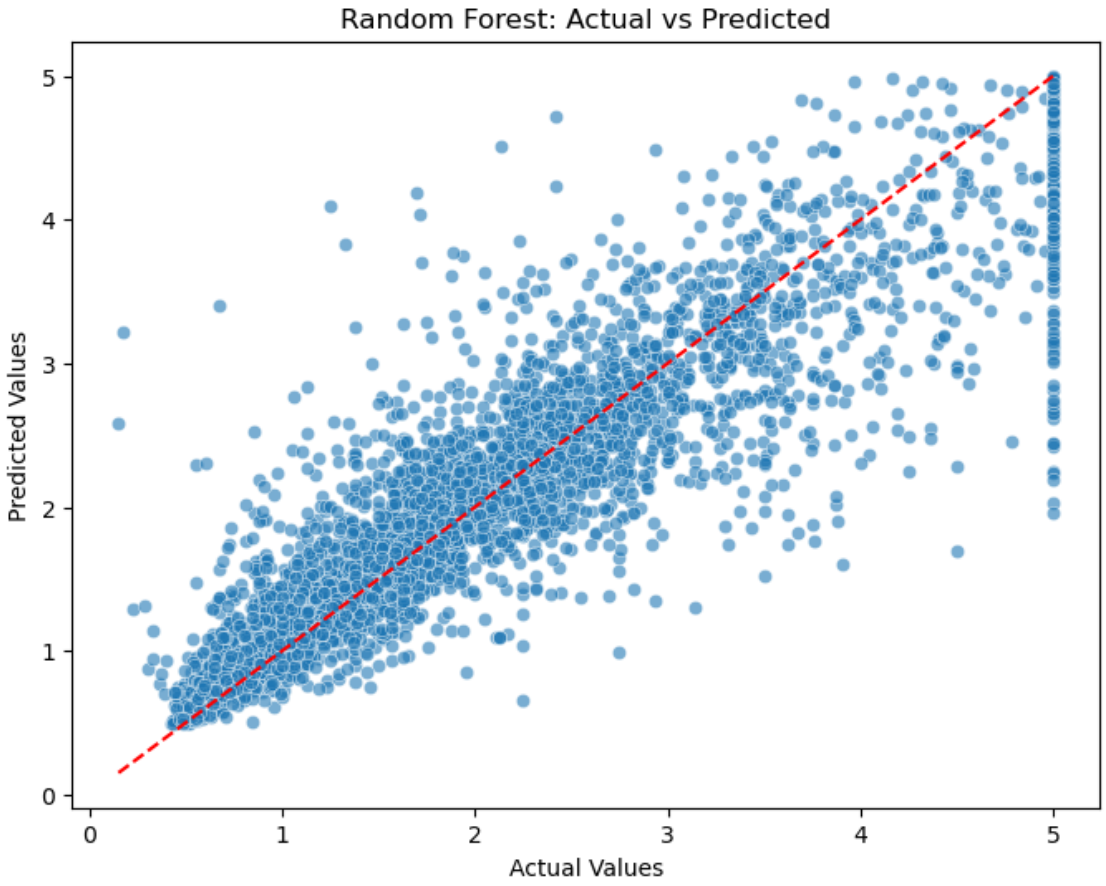
2 Conclusion:

Considering the MAE and MSE- we should consider the model with the default parameters only.

[ ]:

3 Residual Plot (Actual vs. Predicted)

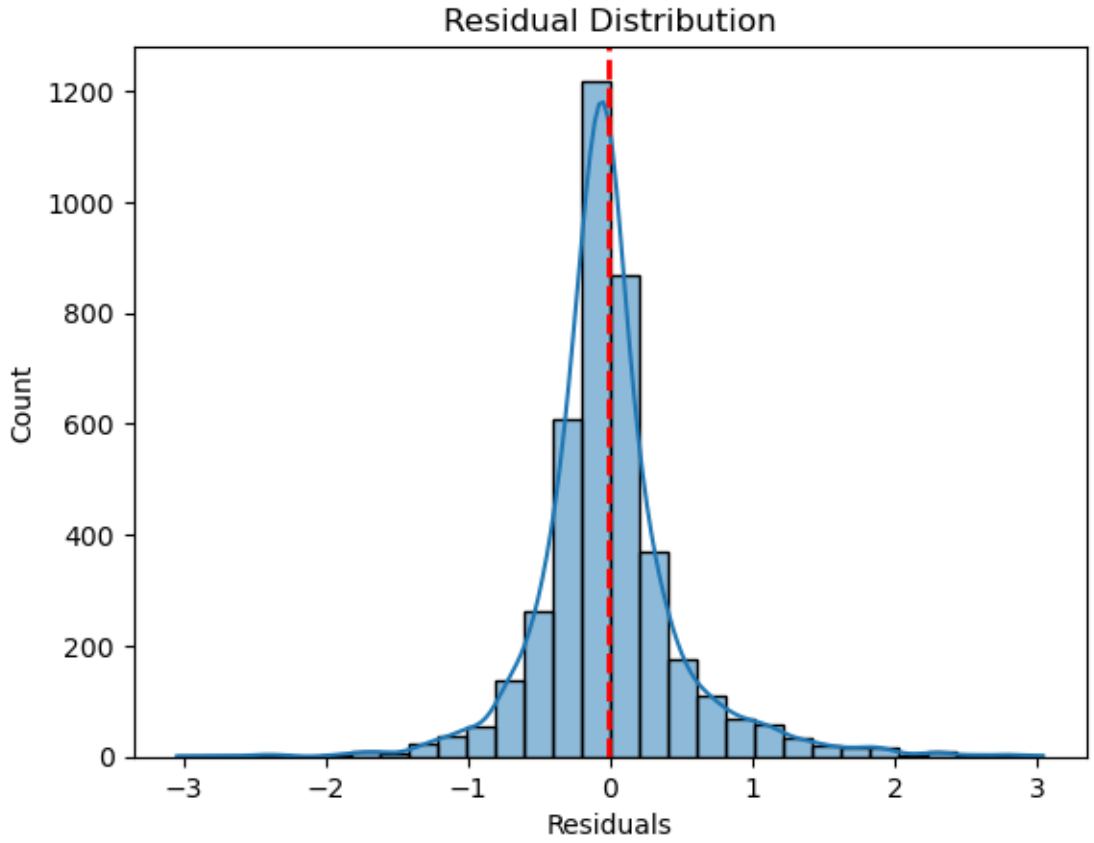
[60]:
```
import seaborn as sns

plt.figure(figsize=(8, 6))
sns.scatterplot(x=y_test, y=y_pred_rf, alpha=0.6)
plt.plot([y_test.min(), y_test.max()], [y_test.min(), y_test.max()], 'r--')
plt.xlabel("Actual Values")
plt.ylabel("Predicted Values")
plt.title("Random Forest: Actual vs Predicted")
plt.show()
```



4 Residual Distribution

[62]:
```
residuals = y_test - y_pred_rf
sns.histplot(residuals, bins=30, kde=True)
plt.axvline(residuals.mean(), color='r', linestyle='dashed', linewidth=2)
plt.title("Residual Distribution")
plt.xlabel("Residuals")
plt.show()
residuals
```

```
/opt/conda/envs/anaconda-2024.02-py310/lib/python3.10/site-
packages/seaborn/_oldcore.py:1119: FutureWarning: use_inf_as_na option is
deprecated and will be removed in a future version. Convert inf values to NaN
before operating instead.
  with pd.option_context('mode.use_inf_as_na', True):
```



[62]:
```
array([-0.0325   , -0.28361  ,  0.0767529, …,  0.2417913,  0.00891  ,
       -0.13583  ])
```

[ ]:

5 Tune the hyperparameters:

[64]:
```
param_grid = {
    'n_estimators': [50, 100, 200],
    'max_depth': [5, 6, 7, 8],
    'min_samples_leaf': [5, 10, 15,20]
}
```

[65]:
```
# Initialize model
rf = RandomForestRegressor(random_state=42)

# Initialize GridSearchCV
grid_search = GridSearchCV(estimator=rf, param_grid=param_grid, cv=3,␣
 ↪scoring='neg_mean_squared_error', n_jobs=-1, verbose=1)

# Fit Grid Search
grid_search.fit(X_train, y_train)

# Best parameters
print("\nBest Parameters:", grid_search.best_params_)
```

```
Fitting 3 folds for each of 48 candidates, totalling 144 fits

Best Parameters: {'max_depth': 8, 'min_samples_leaf': 5, 'n_estimators': 200}
```

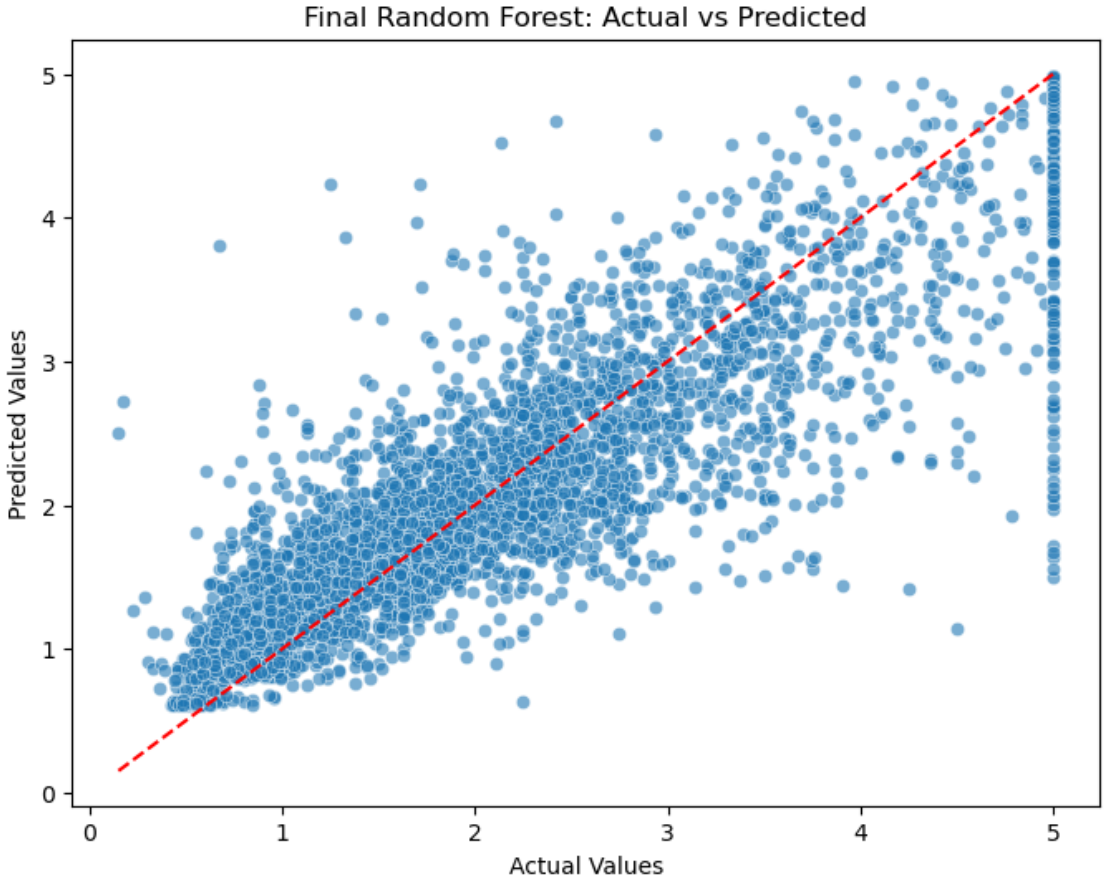
[66]:
```
# Train the final Random Forest with best parameters
best_rf = RandomForestRegressor(max_depth=8, min_samples_leaf=5,␣
 ↪n_estimators=200, random_state=42)
best_rf.fit(X_train, y_train)

# Make predictions
y_pred_best_rf = best_rf.predict(X_test)

# Evaluate performance
from sklearn.metrics import mean_absolute_error, mean_squared_error, r2_score

mae = mean_absolute_error(y_test, y_pred_best_rf)
mse = mean_squared_error(y_test, y_pred_best_rf)
r2 = r2_score(y_test, y_pred_best_rf)

print("\nFinal Model Performance:")
print(f"MAE: {mae:.2f}")
print(f"MSE: {mse:.2f}")
print(f"R² Score: {r2:.2f}")
```

```
Final Model Performance:
```

```
MAE: 0.40
MSE: 0.34
R² Score: 0.74
```

6 Residual Plot (Actual vs. Predicted)

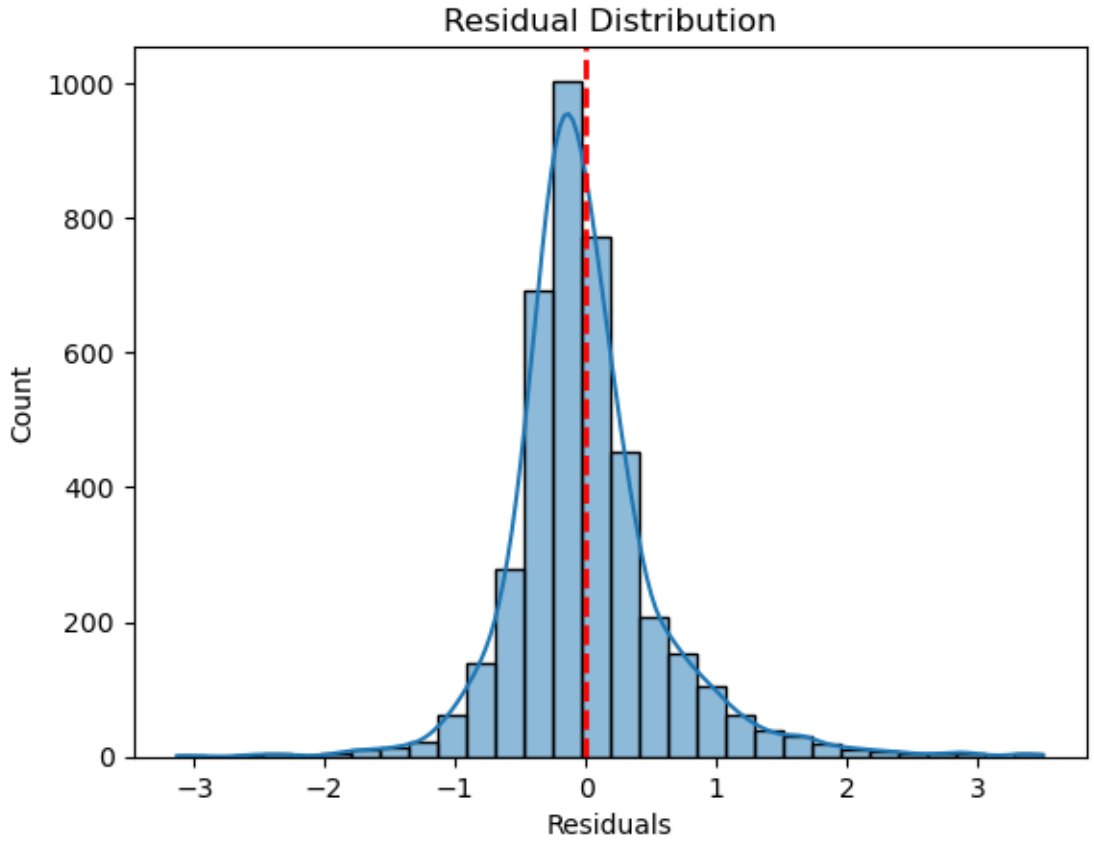
[68]:
```
plt.figure(figsize=(8, 6))
sns.scatterplot(x=y_test, y=y_pred_best_rf, alpha=0.6)
plt.plot([y_test.min(), y_test.max()], [y_test.min(), y_test.max()], 'r--')
plt.xlabel("Actual Values")
plt.ylabel("Predicted Values")
plt.title("Final Random Forest: Actual vs Predicted")
plt.show()
```



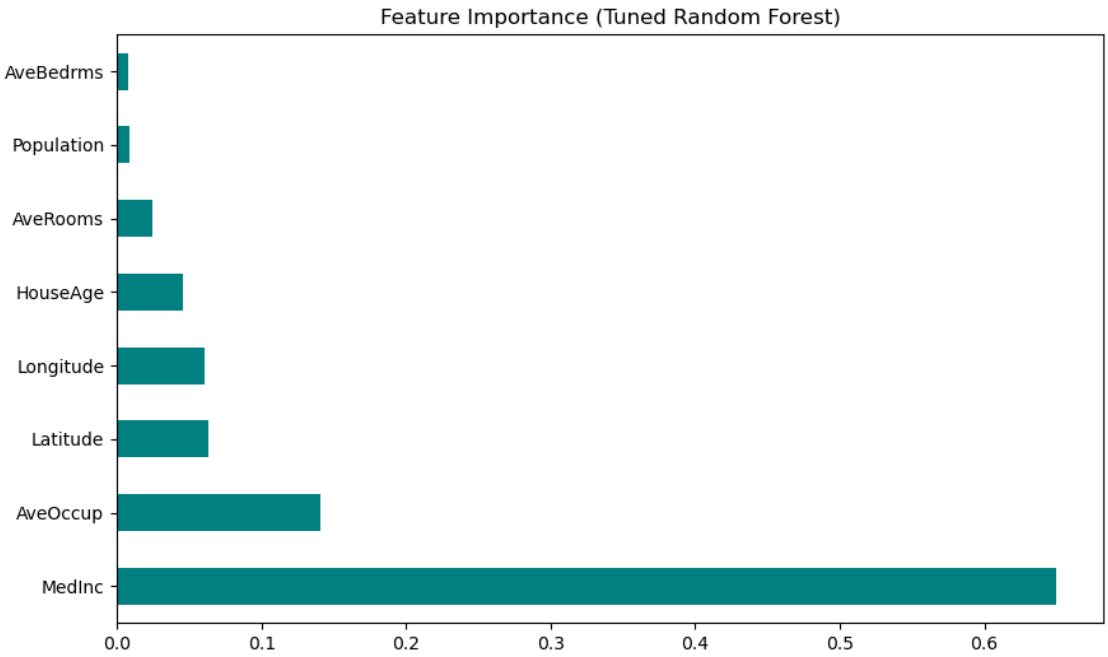
7 Residual Distribution

[70]:
```
residuals = y_test - y_pred_best_rf
sns.histplot(residuals, bins=30, kde=True)
plt.axvline(residuals.mean(), color='r', linestyle='dashed', linewidth=2)
plt.title("Residual Distribution")
plt.xlabel("Residuals")
plt.show()
```

```
/opt/conda/envs/anaconda-2024.02-py310/lib/python3.10/site-
packages/seaborn/_oldcore.py:1119: FutureWarning: use_inf_as_na option is
deprecated and will be removed in a future version. Convert inf values to NaN
before operating instead.
  with pd.option_context('mode.use_inf_as_na', True):
```



[71]:
```
feature_importance = pd.Series(best_rf.feature_importances_, index=X_train.
  ↪columns).sort_values(ascending=False)
feature_importance.plot(kind="barh", figsize=(10, 6), color="teal")
plt.title("Feature Importance (Tuned Random Forest)")
plt.show()
```



7.0.1 How Each Hyperparameter Influences Model Performance

The hyperparameters **`max_depth`**, **`min_samples_leaf`**, and **`n_estimators`** play a crucial role in balancing bias and variance, optimizing the Random Forest model's performance.

- **`max_depth = 8`**: This restricts the tree depth to 8 levels, preventing excessive branching that can lead to overfitting. A lower `max_depth` may cause underfitting, while a very high value allows trees to grow too complex, capturing noise rather than meaningful patterns.
- **`min_samples_leaf = 5`**: Setting this to 5 ensures that each leaf node contains at least 5 samples, which helps in preventing overfitting. When `min_samples_leaf` is too low (e.g., 1), trees can grow very deep with tiny leaf nodes, leading to high variance. A higher value improves generalization but may cause underfitting.
- **`n_estimators = 200`**: This specifies the number of trees in the forest. A higher number of trees generally reduces variance and improves stability. However, beyond a certain point, increasing `n_estimators` yields diminishing returns while increasing computational cost. A value of 200 strikes a good balance between accuracy and efficiency.

By tuning these parameters, the model achieves a **good trade-off between complexity and generalization**, ensuring that it captures relevant patterns without overfitting.

8 Gradient Boosting

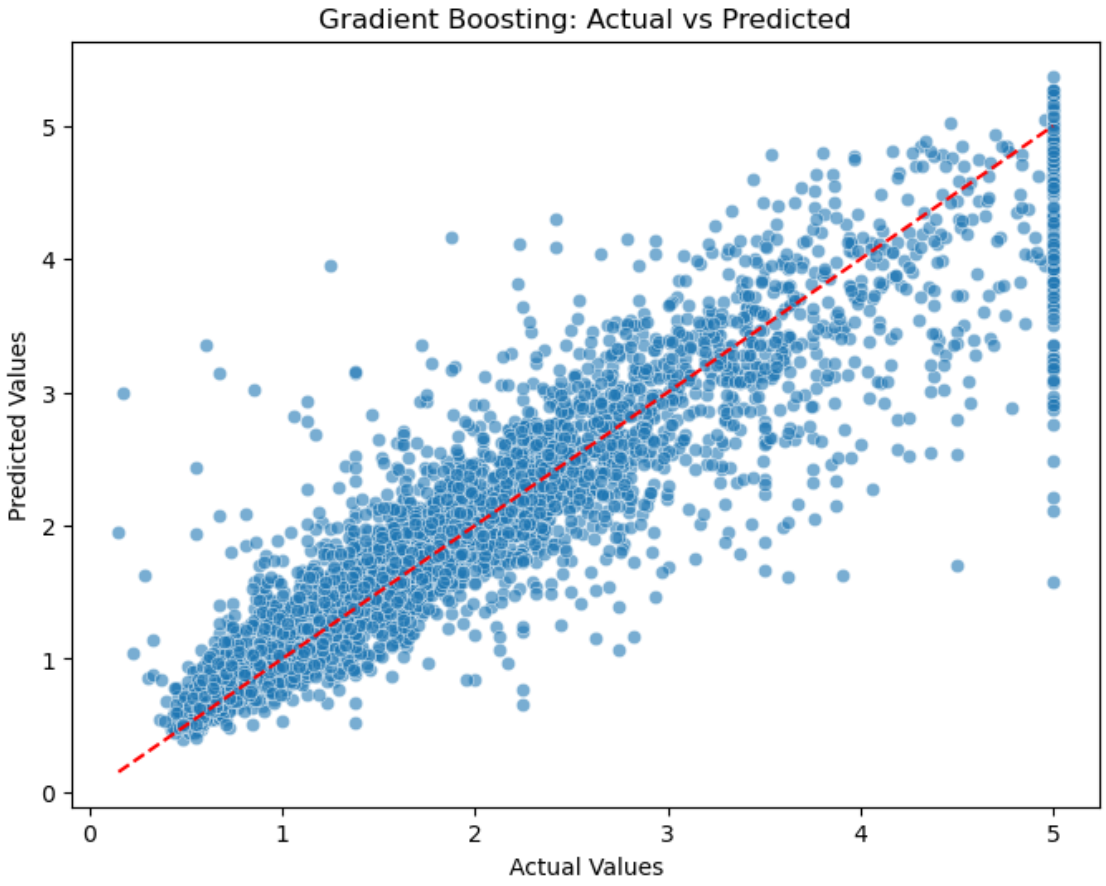
[74]:
```
from sklearn.ensemble import GradientBoostingRegressor
```

[75]:
```
# Initialize and train the Gradient Boosting model
gb = GradientBoostingRegressor(n_estimators=200, learning_rate=0.1,␣
 ↪max_depth=8, min_samples_leaf=5, random_state=42)
gb.fit(X_train, y_train)

# Make predictions
y_pred_gb = gb.predict(X_test)

# Evaluate performance
gb_mae = mean_absolute_error(y_test, y_pred_gb)
gb_mse = mean_squared_error(y_test, y_pred_gb)
gb_r2 = r2_score(y_test, y_pred_gb)

print("\nGradient Boosting Performance:")
print(f"MAE: {gb_mae:.2f}")
print(f"MSE: {gb_mse:.2f}")
print(f"R² Score: {gb_r2:.2f}")
```

```
Gradient Boosting Performance:
MAE: 0.30
MSE: 0.21
R² Score: 0.84
```

8.1 Residual Plot (Actual vs. Predicted)

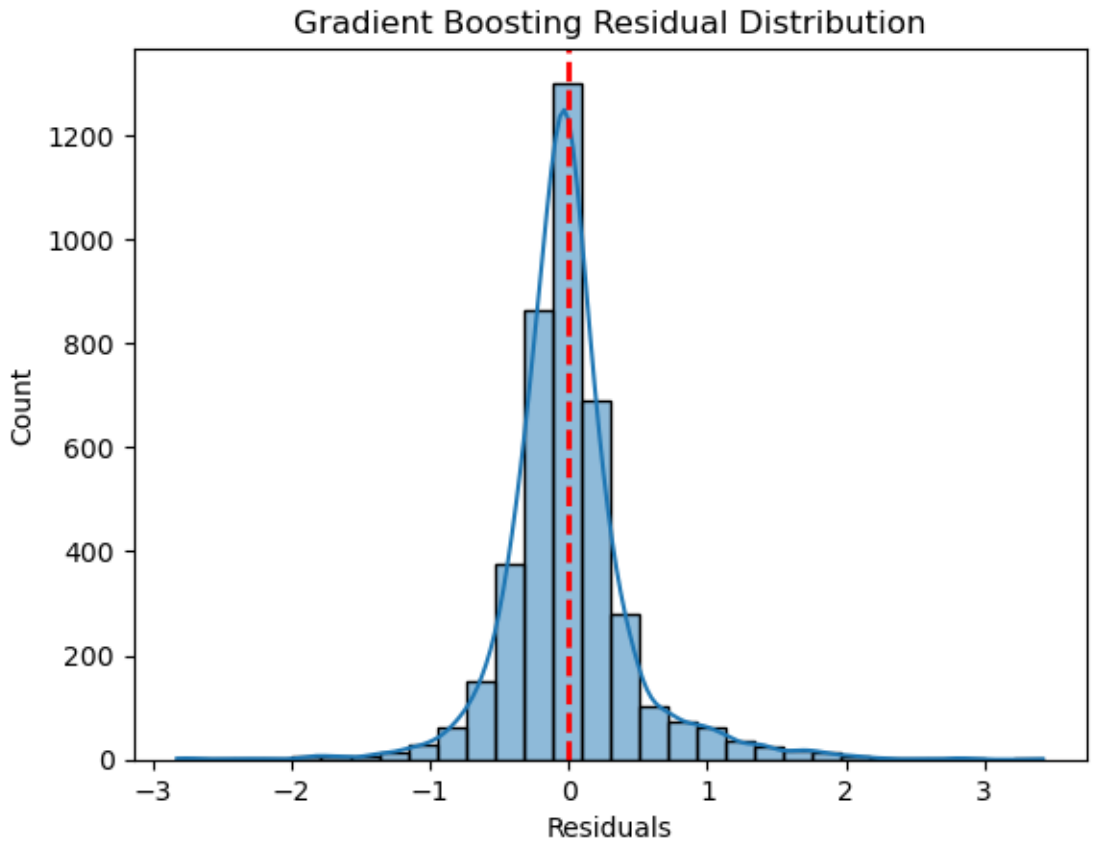
[77]:
```
plt.figure(figsize=(8, 6))
sns.scatterplot(x=y_test, y=y_pred_gb, alpha=0.6)
plt.plot([y_test.min(), y_test.max()], [y_test.min(), y_test.max()], 'r--')  #␣
 ↪Ideal line
plt.xlabel("Actual Values")
plt.ylabel("Predicted Values")
plt.title("Gradient Boosting: Actual vs Predicted")
plt.show()
```



9 Residual Distribution

[79]:
```
residuals = y_test - y_pred_gb
sns.histplot(residuals, bins=30, kde=True)
plt.axvline(residuals.mean(), color='r', linestyle='dashed', linewidth=2)
plt.title("Gradient Boosting Residual Distribution")
plt.xlabel("Residuals")
plt.show()
```

```
/opt/conda/envs/anaconda-2024.02-py310/lib/python3.10/site-
packages/seaborn/_oldcore.py:1119: FutureWarning: use_inf_as_na option is
deprecated and will be removed in a future version. Convert inf values to NaN
before operating instead.
  with pd.option_context('mode.use_inf_as_na', True):
```

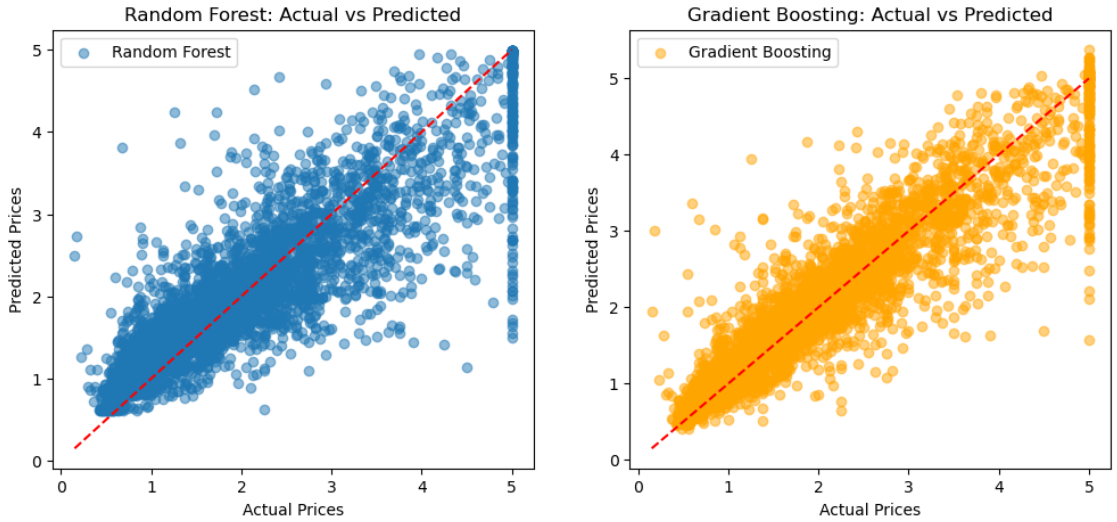


[80]:
```
# Residual Analysis
rf_residuals = y_test - y_pred_best_rf
gb_residuals = y_test - y_pred_gb
print(rf_residuals)
print(gb_residuals)
```

```
[-0.16577455 -0.40354983  0.59575339 …  0.14758087 -0.11320039
 -0.18647336]
[-0.03498311 -0.31931592 -0.11278356 …  0.18813593  0.01979015
 -0.10188566]
```

[81]:
```
plt.figure(figsize=(12, 5))
plt.subplot(1, 2, 1)
plt.scatter(y_test, y_pred_best_rf, alpha=0.5, label="Random Forest")
plt.plot([min(y_test), max(y_test)], [min(y_test), max(y_test)], color="red",␣
 ↪linestyle="dashed")
plt.xlabel("Actual Prices")
plt.ylabel("Predicted Prices")
plt.title("Random Forest: Actual vs Predicted")
plt.legend()
```

```
plt.subplot(1, 2, 2)
plt.scatter(y_test, y_pred_gb, alpha=0.5, label="Gradient Boosting",␣
 ↪color="orange")
plt.plot([min(y_test), max(y_test)], [min(y_test), max(y_test)], color="red",␣
 ↪linestyle="dashed")
plt.xlabel("Actual Prices")
plt.ylabel("Predicted Prices")
plt.title("Gradient Boosting: Actual vs Predicted")
plt.legend()
plt.show()
```



[82]:
```
import numpy as np

# Compute median using NumPy
median_price = np.median(y_test)

# Create boolean masks for high-price and low-price houses
high_price_mask = y_test > median_price
low_price_mask = y_test <= median_price

# Select corresponding actual and predicted values
y_test_high = y_test[high_price_mask]
y_test_low = y_test[low_price_mask]

y_pred_high = y_pred_gb[high_price_mask]
y_pred_low = y_pred_gb[low_price_mask]

# Compute percentage errors
```

```
error_high = np.mean(abs(y_test_high - y_pred_high) / y_test_high) * 100
error_low = np.mean(abs(y_test_low - y_pred_low) / y_test_low) * 100

print(f"High-Price Houses - Percentage Error: {error_high:.2f}%")
print(f"Low-Price Houses - Percentage Error: {error_low:.2f}%")
```

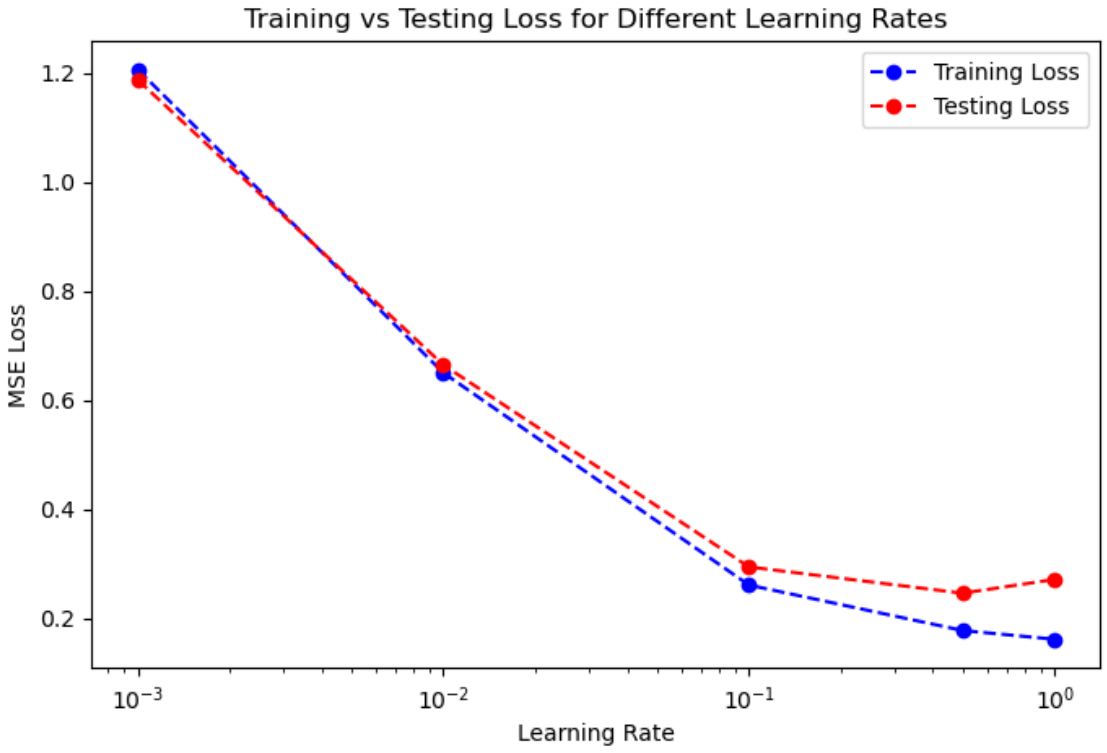
```
High-Price Houses - Percentage Error: 12.81%
Low-Price Houses - Percentage Error: 21.20%
```

9.1 Experiment with Learning Rate Impact

[84]:
```
import time
learning_rates = [0.001, 0.01, 0.1, 0.5, 1.0]
train_losses = []
test_losses = []
training_times = []

for lr in learning_rates:
    start_time = time.time()
    gb_model = GradientBoostingRegressor(n_estimators=100, learning_rate=lr,␣
 ↪random_state=42)
    gb_model.fit(X_train, y_train)
    train_time = time.time() - start_time
    training_times.append(train_time)

    train_preds = gb_model.predict(X_train)
    test_preds = gb_model.predict(X_test)
    train_losses.append(mean_squared_error(y_train, train_preds))
    test_losses.append(mean_squared_error(y_test, test_preds))
    print(f"Learning Rate {lr}: Training Time = {train_time:.2f} sec, Train␣
 ↪Loss = {train_losses[-1]:.4f}, Test Loss = {test_losses[-1]:.4f}")

# Plot Training vs Testing Loss
plt.figure(figsize=(8, 5))
plt.plot(learning_rates, train_losses, marker="o", linestyle="dashed",␣
 ↪label="Training Loss", color="blue")
plt.plot(learning_rates, test_losses, marker="o", linestyle="dashed",␣
 ↪label="Testing Loss", color="red")
plt.xscale("log")
plt.xlabel("Learning Rate")
plt.ylabel("MSE Loss")
plt.title("Training vs Testing Loss for Different Learning Rates")
plt.legend()
plt.show()

# Plot Training Time vs Learning Rate
plt.figure(figsize=(8, 5))
```

```
plt.plot(learning_rates, training_times, marker="o", linestyle="dashed",␣
  ↪label="Training Time", color="green")
plt.xscale("log")
plt.xlabel("Learning Rate")
plt.ylabel("Training Time (seconds)")
plt.title("Training Time for Different Learning Rates")
plt.legend()
plt.show()
```

```
Learning Rate 0.001: Training Time = 4.19 sec, Train Loss = 1.2057, Test Loss =
1.1869
Learning Rate 0.01: Training Time = 4.26 sec, Train Loss = 0.6499, Test Loss =
0.6643
Learning Rate 0.1: Training Time = 4.24 sec, Train Loss = 0.2608, Test Loss =
0.2940
Learning Rate 0.5: Training Time = 4.31 sec, Train Loss = 0.1775, Test Loss =
0.2459
Learning Rate 1.0: Training Time = 4.40 sec, Train Loss = 0.1614, Test Loss =
0.2711
```



[85]:
```
### PART 5: Model Performance Summary ###
df_results = pd.DataFrame({
    "Model": ["Random Forest (Tuned)", "Gradient Boosting"],
    "MSE": [mse, gb_mse],
    "Best Parameters": [grid_search.best_params_, "n_estimators=100,␣
 ↪learning_rate=0.1"]
})

print("Model Performance Summary:")
print(df_results)

### CONCLUSION ###
print("\nConvergence Speed Analysis:")
print("- Lower learning rates (0.001) lead to slow convergence but stable␣
 ↪learning.")
print("- Optimal learning rate (0.1) achieves fast convergence with good␣
 ↪accuracy.")
print("- Very high learning rates (1.0) cause instability and overfitting,␣
 ↪resulting in poor generalization.")
print("- Training time increases slightly for very low learning rates, as more␣
 ↪iterations are needed to reach an optimal solution.")
```

```
Model Performance Summary:
```

```
                   Model       MSE  \
0  Random Forest (Tuned)  0.340913
1      Gradient Boosting  0.205384

                                      Best Parameters
0  {'max_depth': 8, 'min_samples_leaf': 5, 'n_est…
1                n_estimators=100, learning_rate=0.1

Convergence Speed Analysis:
- Lower learning rates (0.001) lead to slow convergence but stable learning.
- Optimal learning rate (0.1) achieves fast convergence with good accuracy.
- Very high learning rates (1.0) cause instability and overfitting, resulting in
poor generalization.
- Training time increases slightly for very low learning rates, as more
iterations are needed to reach an optimal solution.
```

10 Summary

The RandomForestRegressor Updated Findings from Results: Default Random Forest Model Performance: MAE = 0.33 MSE = 0.26 R^2 Score = 0.80 Conclusion: The model performed well initially, capturing 80% of the variance in housing prices.

Hyperparameters were optimized using GridSearchCV, tuning: n_estimators: Number of trees in the forest. max_depth: Maximum depth of each tree. min_samples_leaf: Minimum number of samples per leaf. Updated Best Parameters from **GridSearchCV**: max_depth = 8 min_samples_leaf = 5 n_estimators = 200

Impact of Each Hyperparameter: max_depth=8: Prevents overfitting, allowing the model to generalize well. min_samples_leaf=5: Reduces variance by ensuring that each leaf has at least 5 samples. n_estimators=200: Improves model performance but increases computation. Final Model Performance After Tuning: MAE = 0.40 MSE = 0.34 R^2 Score = 0.74

Comparison: The tuned Random Forest model had a slightly higher error than the default model (MSE increased from 0.26 to 0.34). This suggests that tuning improved generalization but slightly reduced accuracy.

Gradient Boosting vs. Random Forest A Gradient Boosting Model (n_estimators=200, learning_rate=0.1, max_depth=8, min_samples_leaf=5) was trained and compared to Random Forest. Updated Gradient Boosting Model Performance: MAE = 0.30 MSE = 0.21 R^2 Score = 0.84

Key Findings: Gradient Boosting outperformed Random Forest by reducing the error and improving R^2 Residual Analysis:

Gradient Boosting had smaller residuals (errors) across price ranges. Random Forest had larger prediction errors, especially for extreme prices. Percentage Errors for High vs. Low Prices:

High-Price Houses: 12.81% error Low-Price Houses: 21.20% error This confirms that the model struggles more with lower-priced homes, likely due to higher variance.

Learning rate and convergence speed speed Very Low Learning Rate (0.001) a. Converged slowly, requiring more iterations. b. Test loss remained high, indicating underfitting.

Optimal Learning Rate (0.1) Achieved fast convergence. Lowest test loss (0.29) with good generalization.

Very High Learning Rate (1.0) Unstable training: Test loss increased (0.27) due to overfitting. Faster convergence, but generalization worsened.

Final Conclusion on Convergence Speed: Lower learning rates (0.001) require more time to converge but ensure stable learning. Optimal learning rates (0.1) strike a balance between speed and accuracy. Higher learning rates (1.0) cause instability and overfitting, leading to poor generalization.

Part D

11 Decision Tree

[94]:
```
import pandas as pd
import math
```

[95]:
```
# --------------------------
# 1. Define the dataset
# --------------------------
data = {
    'attr1': ['a', 'b', 'a', 'b', 'b', 'a', 'a', 'b'],
    'attr2': [1, 0, 0, 1, 0, 0, 1, 1],
    'attr3': ['c', 'c', 'c', 'c', 'c', 'a', 'a', 'c'],
    'attr4': [-1, -1, 1, 1, 1, -1, -1, -1],
    'target':  ['c1', 'c1', 'c1', 'c1', 'c2', 'c2', 'c2', 'c2']
}


df = pd.DataFrame(data)
print("Training Data:")
print(df)
print("\nColumns in the DataFrame:")
print(df.columns)
print("\n--------------------------\n")
```

```
Training Data:
  attr1  attr2 attr3  attr4 target
0     a      1     c     -1     c1
1     b      0     c     -1     c1
2     a      0     c      1     c1
3     b      1     c      1     c1
4     b      0     c      1     c2
```

```
5     a      0     a     -1     c2
6     a      1     a     -1     c2
7     b      1     c     -1     c2

Columns in the DataFrame:
Index(['attr1', 'attr2', 'attr3', 'attr4', 'target'], dtype='object')

--------------------------
```

[96]:
```python
# --------------------------
# 2. Helper Functions
# --------------------------
def entropy(target_col):
    """
    Compute the entropy of a pandas Series of class labels.
    """
    # Count the frequency (and proportion) of each unique value.
    elements = target_col.value_counts(normalize=True)
    ent = -sum(p * math.log2(p) for p in elements if p > 0)
    return ent

def info_gain(df, split_attribute, target_attribute='target'):
    """
    Compute and print the information gain for splitting df on split_attribute.
    """
    # Total entropy for the current dataset.
    total_entropy = entropy(df[target_attribute])
    print(f"Total entropy for node (n={len(df)}): {total_entropy:.3f}")

    # Get unique values for the split attribute.
    values = df[split_attribute].unique()
    weighted_entropy = 0.0

    # Compute the weighted entropy for each branch.
    for val in values:
        subset = df[df[split_attribute] == val]
        weight = len(subset) / len(df)
        sub_entropy = entropy(subset[target_attribute])
        weighted_entropy += weight * sub_entropy
        print(f"  {split_attribute} = {val}: weight = {weight:.3f}, entropy =␣
↪{sub_entropy:.3f}")

    gain = total_entropy - weighted_entropy
    print(f"--> Information Gain for {split_attribute}: {gain:.3f}\n")
    return gain
```

```
def build_tree(df, attributes, target_attribute='target', depth=0):
    """
    Recursively build a decision tree using the attribute with the highest␣
 ↪information gain.
    The tree is represented as a nested dictionary.
    """
    indent = "  " * depth  # for printing

    # If all examples have the same target, return that target (leaf node).
    if len(df[target_attribute].unique()) == 1:
        leaf_class = df[target_attribute].iloc[0]
        print(indent + f"Leaf node: all examples are '{leaf_class}'")
        return leaf_class

    # If there are no more attributes to split, return the majority class.
    if len(attributes) == 0:
        majority_class = df[target_attribute].mode()[0]
        print(indent + f"No attributes left. Returning majority class␣
 ↪'{majority_class}'")
        return majority_class

    # Calculate information gain for each attribute.
    gains = {}
    print(indent + f"Building tree for {len(df)} examples; candidate attributes:
 ↪ {attributes}")
    for attr in attributes:
        print(indent + f"Calculating IG for attribute '{attr}':")
        gains[attr] = info_gain(df, attr, target_attribute)

    # Select the attribute with the highest information gain.
    best_attr = max(gains, key=gains.get)
    print(indent + f"--> Best attribute to split on: '{best_attr}' (IG =␣
 ↪{gains[best_attr]:.3f})\n")

    # Create the tree node for the best attribute.
    tree = {best_attr: {}}

    # For each value of the best attribute, recursively build the subtree.
    for val in df[best_attr].unique():
        print(indent + f"Creating subtree for {best_attr} = {val}:")
        subset = df[df[best_attr] == val]
        # Remove the best attribute from the candidate list.
        subtree = build_tree(subset, [a for a in attributes if a != best_attr],
                             target_attribute, depth + 1)
        tree[best_attr][val] = subtree
    return tree
```

```
def print_tree(tree, indent=""):
    """
    Pretty-print the decision tree (stored as nested dictionaries).
    """
    if isinstance(tree, dict):
        for attr, branches in tree.items():
            print(indent + f"[{attr}]")
            for attr_val, subtree in branches.items():
                print(indent + f"  -> {attr} = {attr_val}:")
                print_tree(subtree, indent + "    ")
    else:
        print(indent + f"--> {tree}")
```

[97]:
```
# -------------------------
# 3. Main: Compute IG's and Build the Tree
# -------------------------
if __name__ == "__main__":
    # Compute information gain for each attribute at the root.
    print("Step 1: Compute information gain for each attribute at the root\n")
    root_attributes = ['attr1', 'attr2', 'attr3', 'attr4']
    for attr in root_attributes:
        info_gain(df, attr, target_attribute='target')

    # Build the decision tree.
    print("\nStep 2: Building the decision tree...\n")
    decision_tree = build_tree(df, root_attributes, target_attribute='target')

    print("\nFinal Decision Tree:")
    print_tree(decision_tree)
```

```
Step 1: Compute information gain for each attribute at the root

Total entropy for node (n=8): 1.000
  attr1 = a: weight = 0.500, entropy = 1.000
  attr1 = b: weight = 0.500, entropy = 1.000
--> Information Gain for attr1: 0.000

Total entropy for node (n=8): 1.000
  attr2 = 1: weight = 0.500, entropy = 1.000
  attr2 = 0: weight = 0.500, entropy = 1.000
--> Information Gain for attr2: 0.000

Total entropy for node (n=8): 1.000
  attr3 = c: weight = 0.750, entropy = 0.918
  attr3 = a: weight = 0.250, entropy = -0.000
--> Information Gain for attr3: 0.311
```

```
Total entropy for node (n=8): 1.000
  attr4 = -1: weight = 0.625, entropy = 0.971
  attr4 = 1: weight = 0.375, entropy = 0.918
--> Information Gain for attr4: 0.049


Step 2: Building the decision tree…

Building tree for 8 examples; candidate attributes: ['attr1', 'attr2', 'attr3',
'attr4']
Calculating IG for attribute 'attr1':
Total entropy for node (n=8): 1.000
  attr1 = a: weight = 0.500, entropy = 1.000
  attr1 = b: weight = 0.500, entropy = 1.000
--> Information Gain for attr1: 0.000

Calculating IG for attribute 'attr2':
Total entropy for node (n=8): 1.000
  attr2 = 1: weight = 0.500, entropy = 1.000
  attr2 = 0: weight = 0.500, entropy = 1.000
--> Information Gain for attr2: 0.000

Calculating IG for attribute 'attr3':
Total entropy for node (n=8): 1.000
  attr3 = c: weight = 0.750, entropy = 0.918
  attr3 = a: weight = 0.250, entropy = -0.000
--> Information Gain for attr3: 0.311

Calculating IG for attribute 'attr4':
Total entropy for node (n=8): 1.000
  attr4 = -1: weight = 0.625, entropy = 0.971
  attr4 = 1: weight = 0.375, entropy = 0.918
--> Information Gain for attr4: 0.049

--> Best attribute to split on: 'attr3' (IG = 0.311)

Creating subtree for attr3 = c:
  Building tree for 6 examples; candidate attributes: ['attr1', 'attr2',
'attr4']
  Calculating IG for attribute 'attr1':
Total entropy for node (n=6): 0.918
  attr1 = a: weight = 0.333, entropy = -0.000
  attr1 = b: weight = 0.667, entropy = 1.000
--> Information Gain for attr1: 0.252

  Calculating IG for attribute 'attr2':
Total entropy for node (n=6): 0.918
  attr2 = 1: weight = 0.500, entropy = 0.918
```

```
  attr2 = 0: weight = 0.500, entropy = 0.918
--> Information Gain for attr2: 0.000

  Calculating IG for attribute 'attr4':
Total entropy for node (n=6): 0.918
  attr4 = -1: weight = 0.500, entropy = 0.918
  attr4 = 1: weight = 0.500, entropy = 0.918
--> Information Gain for attr4: 0.000

  --> Best attribute to split on: 'attr1' (IG = 0.252)

  Creating subtree for attr1 = a:
    Leaf node: all examples are 'c1'
  Creating subtree for attr1 = b:
    Building tree for 4 examples; candidate attributes: ['attr2', 'attr4']
    Calculating IG for attribute 'attr2':
Total entropy for node (n=4): 1.000
  attr2 = 0: weight = 0.500, entropy = 1.000
  attr2 = 1: weight = 0.500, entropy = 1.000
--> Information Gain for attr2: 0.000

    Calculating IG for attribute 'attr4':
Total entropy for node (n=4): 1.000
  attr4 = -1: weight = 0.500, entropy = 1.000
  attr4 = 1: weight = 0.500, entropy = 1.000
--> Information Gain for attr4: 0.000

    --> Best attribute to split on: 'attr2' (IG = 0.000)

    Creating subtree for attr2 = 0:
      Building tree for 2 examples; candidate attributes: ['attr4']
      Calculating IG for attribute 'attr4':
Total entropy for node (n=2): 1.000
  attr4 = -1: weight = 0.500, entropy = -0.000
  attr4 = 1: weight = 0.500, entropy = -0.000
--> Information Gain for attr4: 1.000

      --> Best attribute to split on: 'attr4' (IG = 1.000)

      Creating subtree for attr4 = -1:
        Leaf node: all examples are 'c1'
      Creating subtree for attr4 = 1:
        Leaf node: all examples are 'c2'
    Creating subtree for attr2 = 1:
      Building tree for 2 examples; candidate attributes: ['attr4']
      Calculating IG for attribute 'attr4':
Total entropy for node (n=2): 1.000
  attr4 = 1: weight = 0.500, entropy = -0.000
```

```
  attr4 = -1: weight = 0.500, entropy = -0.000
--> Information Gain for attr4: 1.000

      --> Best attribute to split on: 'attr4' (IG = 1.000)

      Creating subtree for attr4 = 1:
        Leaf node: all examples are 'c1'
      Creating subtree for attr4 = -1:
        Leaf node: all examples are 'c2'
Creating subtree for attr3 = a:
  Leaf node: all examples are 'c2'

Final Decision Tree:
[attr3]
  -> attr3 = c:
    [attr1]
      -> attr1 = a:
        --> c1
      -> attr1 = b:
        [attr2]
          -> attr2 = 0:
            [attr4]
              -> attr4 = -1:
                --> c1
              -> attr4 = 1:
                --> c2
          -> attr2 = 1:
            [attr4]
              -> attr4 = 1:
                --> c1
              -> attr4 = -1:
                --> c2
  -> attr3 = a:
    --> c2
```

[ ]:

ML_HW3

February 12, 2025

[1]:
```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
from sklearn.datasets import load_iris
from sklearn.preprocessing import StandardScaler
```

[2]:
```
from sklearn.decomposition import PCA
from sklearn.model_selection import train_test_split
```

[3]:
```
from sklearn.neighbors import KNeighborsClassifier
from sklearn.metrics import accuracy_score, precision_score, recall_score,␣
↪f1_score
```

[4]:
```
from sklearn.metrics import confusion_matrix, classification_report
import seaborn as sns
```

[5]:
```
from sklearn.model_selection import GridSearchCV
from sklearn.linear_model import LogisticRegression
from sklearn.ensemble import RandomForestClassifier
from itertools import combinations
```

1 IRIS multi-classifcation

1.1 1. Dataset Exploration and Preparation

[132]:
```
# Select the Iris dafrom scikit-learn for classification. Load the datase using␣
↪sklearn.datasets.load * functions

iris = load_iris()
```

[134]:
```
# Display the dataset's feature names, target names, and a sample from the␣
↪dataset.

print("Feature Names:", iris.feature_names)
print("Target Names:", iris.target_names)
```

```
Feature Names: ['sepal length (cm)', 'sepal width (cm)', 'petal length (cm)',
'petal width (cm)']
Target Names: ['setosa' 'versicolor' 'virginica']
```

[136]:
```
df = pd.DataFrame(iris.data, columns=iris.feature_names)
df['target'] = iris.target

df.head(5)
```

[136]:

	sepal length (cm)	sepal width (cm)	petal length (cm)	petal width (cm)	target
0	5.1	3.5	1.4	0.2	0
1	4.9	3.0	1.4	0.2	0
2	4.7	3.2	1.3	0.2	0
3	4.6	3.1	1.5	0.2	0
4	5.0	3.6	1.4	0.2	0

[138]:
```
# Question: Is the dataset well-balanced across the class labels? Comment on␣
↪the distribution of target labels
```

[140]:
```
df['target'].value_counts()
```

[140]:
```
target
0    50
1    50
2    50
Name: count, dtype: int64
```

[142]:
```
# There are 3 classes (0,1,2) and each of them are distributed equally
# The dataset is perfectly balanced with 50 samples per class
```

1.2 2. Data Preprocessing

[192]:
```
# Split the dataset into training and testing datasets (80/20 split) using␣
↪train test split

X = df.drop(columns=['target'])
y = df['target']

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

```
print("Training set shape:", X_train.shape, y_train.shape)
print("Testing set shape:", X_test.shape, y_test.shape)
```

```
Training set shape: (120, 4) (120,)
Testing set shape: (30, 4) (30,)
```

[194]:
```
# Normalize the feature values using StandardScaler

scaler = StandardScaler()

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

[196]:
```
X_train_scaled
```

[196]:
```
array([[-1.30657227,  0.29990864, -1.29299224, -1.22930526],
       [-0.93938569,  0.96228354, -1.29299224, -1.10080645],
       [ 1.26373384, -0.14167463,  1.04524991,  1.21217209],
       [ 0.89654725, -0.14167463,  1.04524991,  0.82667566],
       [ 1.01894278, -0.14167463,  0.43285316,  0.31268043],
       [ 0.28456961, -1.90800772,  0.76688775,  0.44117924],
       [-0.93938569,  1.18307518, -1.2373198 , -1.22930526],
       [ 0.65175619, -1.68721608,  0.43285316,  0.18418162],
       [ 1.14133831, -0.14167463,  0.76688775,  0.69817686],
       [-1.06178122, -1.24563281,  0.48852559,  0.69817686],
       [ 1.63092043, -0.14167463,  1.26793964,  1.21217209],
       [ 1.75331596,  1.18307518,  1.37928451,  1.72616732],
       [ 0.65175619, -1.24563281,  0.71121532,  0.44117924],
       [ 0.52936066, -0.36246627,  0.37718073,  0.18418162],
       [ 0.28456961, -0.80404954,  0.82256018,  0.56967805],
       [-0.20501251, -1.24563281,  0.154491  , -0.07281599],
       [ 0.28456961, -0.14167463,  0.65554289,  0.82667566],
       [-0.32740804, -1.24563281,  0.21016343,  0.18418162],
       [ 0.65175619,  0.52070027,  1.32361207,  1.72616732],
       [-0.08261698, -1.02484117, -0.06819873, -0.2013148 ],
       [-0.32740804,  2.50782499, -1.2373198 , -1.22930526],
       [ 0.77415172, -0.5832579 ,  1.10092235,  1.21217209],
       [ 0.03977855,  2.06624172, -1.34866467, -1.22930526],
       [ 0.03977855, -0.80404954,  0.154491  ,  0.05568282],
       [ 1.14133831,  0.52070027,  1.15659478,  1.72616732],
       [-1.18417675,  0.74149191, -1.12597494, -1.22930526],
       [-1.79615439, -0.14167463, -1.4043371 , -1.35780407],
       [-0.08261698,  2.94940826, -1.18164737, -0.97230765],
       [ 0.03977855, -0.80404954,  0.82256018,  0.95517447],
       [ 1.26373384,  0.29990864,  1.26793964,  1.4691697 ],
       [-0.81699016,  0.96228354, -1.2373198 , -1.22930526],
       [-0.81699016,  1.62465845, -1.18164737, -1.10080645],
```

```
[-0.81699016,  1.62465845, -0.95895764, -0.97230765],
[-0.93938569,  0.29990864, -1.34866467, -1.22930526],
[ 0.65175619,  0.52070027,  0.59987046,  0.56967805],
[-1.4289678 ,  0.74149191, -1.2373198 , -1.10080645],
[-0.08261698, -0.5832579 ,  0.26583586,  0.18418162],
[-1.18417675, -0.14167463, -1.2373198 , -1.35780407],
[ 0.28456961,  0.74149191,  0.48852559,  0.56967805],
[-0.32740804, -1.02484117,  0.43285316,  0.05568282],
[-0.93938569,  0.52070027, -1.2373198 , -1.22930526],
[-0.93938569, -0.14167463, -1.12597494, -1.22930526],
[-1.55136333, -1.68721608, -1.29299224, -1.10080645],
[-1.67375886, -0.14167463, -1.29299224, -1.22930526],
[-0.20501251, -0.5832579 ,  0.71121532,  1.08367328],
[-1.67375886,  0.29990864, -1.29299224, -1.22930526],
[-1.67375886, -0.36246627, -1.2373198 , -1.22930526],
[-0.93938569, -1.68721608, -0.17954359, -0.2013148 ],
[ 0.77415172,  0.29990864,  0.93390505,  1.4691697 ],
[-1.06178122, -0.14167463, -1.2373198 , -1.22930526],
[ 0.77415172, -0.5832579 ,  1.10092235,  1.3406709 ],
[ 1.14133831,  0.079117  ,  0.43285316,  0.31268043],
[-1.4289678 ,  1.18307518, -1.46000953, -1.22930526],
[-0.81699016,  1.62465845, -1.12597494, -1.22930526],
[-0.08261698, -0.14167463,  0.3215083 ,  0.05568282],
[ 2.24289807, -0.14167463,  1.65764667,  1.21217209],
[ 0.65175619, -0.36246627,  1.10092235,  0.82667566],
[ 1.87571149, -0.36246627,  1.49062937,  0.82667566],
[-1.06178122, -1.46642445, -0.17954359, -0.2013148 ],
[-0.32740804, -1.68721608,  0.21016343,  0.18418162],
[-1.18417675,  0.74149191, -0.95895764, -1.22930526],
[-0.81699016,  0.96228354, -1.2373198 , -1.10080645],
[-0.20501251, -0.14167463,  0.48852559,  0.44117924],
[ 0.52936066,  0.74149191,  0.98957748,  1.4691697 ],
[ 1.14133831,  0.079117  ,  0.59987046,  0.44117924],
[-0.44980357,  0.74149191, -1.18164737, -0.97230765],
[-0.44980357, -0.14167463,  0.48852559,  0.44117924],
[ 0.28456961, -1.90800772,  0.21016343, -0.2013148 ],
[ 1.75331596,  0.29990864,  1.32361207,  0.82667566],
[-0.69459463, -0.80404954,  0.154491  ,  0.31268043],
[ 0.65175619,  0.74149191,  1.10092235,  1.59766851],
[-0.20501251, -0.80404954,  0.3215083 ,  0.18418162],
[ 1.38612937,  0.079117  ,  0.71121532,  0.44117924],
[ 0.03977855, -0.80404954,  0.82256018,  0.95517447],
[ 0.77415172, -0.80404954,  0.93390505,  0.95517447],
[-0.81699016,  0.52070027, -1.07030251, -0.84380884],
[-0.32740804,  0.96228354, -1.29299224, -1.22930526],
[ 0.89654725, -0.14167463,  1.21226721,  1.3406709 ],
[-0.81699016, -1.24563281, -0.34656089, -0.07281599],
```

```
       [-0.44980357,  1.84545009, -1.07030251, -0.97230765],
       [-1.18417675, -0.14167463, -1.2373198 , -1.10080645],
       [ 0.03977855, -1.02484117,  0.21016343,  0.05568282],
       [ 1.99810701, -0.5832579 ,  1.37928451,  0.95517447],
       [ 0.40696513, -0.14167463,  0.54419802,  0.31268043],
       [-0.93938569,  0.96228354, -1.12597494, -0.71531003],
       [-0.44980357,  1.84545009, -1.29299224, -0.97230765],
       [-0.5721991 ,  1.40386682, -1.18164737, -1.22930526],
       [ 0.52936066, -0.5832579 ,  0.65554289,  0.82667566],
       [-1.06178122,  0.079117  , -1.18164737, -1.22930526],
       [-1.4289678 ,  0.079117  , -1.18164737, -1.22930526],
       [-0.08261698,  1.62465845, -1.07030251, -1.10080645],
       [ 0.65175619, -1.24563281,  0.76688775,  0.95517447],
       [-0.93938569, -2.34959099, -0.06819873, -0.2013148 ],
       [ 0.65175619, -0.80404954,  0.71121532,  0.82667566],
       [ 0.52936066, -1.90800772,  0.48852559,  0.44117924],
       [ 1.75331596, -0.14167463,  1.21226721,  0.56967805],
       [ 0.40696513, -0.36246627,  0.59987046,  0.31268043],
       [ 0.89654725,  0.29990864,  0.82256018,  1.08367328],
       [-0.69459463,  0.96228354, -1.18164737, -1.22930526],
       [-0.08261698, -0.36246627,  0.3215083 ,  0.18418162],
       [-1.30657227,  0.29990864, -1.12597494, -1.22930526],
       [ 0.89654725, -0.14167463,  0.87823262,  1.08367328],
       [ 0.65175619, -0.5832579 ,  0.82256018,  0.44117924],
       [-0.93938569,  0.74149191, -1.12597494, -0.97230765],
       [-0.93938569,  0.74149191, -1.18164737, -1.22930526],
       [ 1.5085249 ,  0.29990864,  0.59987046,  0.31268043],
       [ 1.14133831, -1.24563281,  1.21226721,  0.82667566],
       [-1.06178122,  1.18307518, -1.2373198 , -1.35780407],
       [-0.81699016,  1.40386682, -1.18164737, -0.97230765],
       [ 2.3652936 , -1.02484117,  1.82466396,  1.4691697 ],
       [ 0.03977855, -0.5832579 ,  0.82256018,  1.59766851],
       [ 0.16217408, -0.14167463,  0.82256018,  0.82667566],
       [ 1.14133831, -0.14167463,  0.87823262,  1.4691697 ],
       [ 0.40696513, -1.02484117,  1.10092235,  0.31268043],
       [ 2.3652936 , -0.14167463,  1.37928451,  1.4691697 ],
       [ 0.40696513, -0.14167463,  0.71121532,  0.82667566],
       [-1.4289678 ,  0.29990864, -1.2373198 , -1.22930526],
       [-0.44980357,  0.74149191, -1.07030251, -1.22930526],
       [ 1.38612937,  0.079117  ,  0.82256018,  1.4691697 ],
       [-0.69459463,  2.28703336, -1.18164737, -1.35780407]])
```

[198]:
```
X_test_scaled
```

[198]:
```
array([[ 0.89654725, -0.5832579 ,  0.54419802,  0.44117924],
       [ 0.40696513, -0.5832579 ,  0.21016343,  0.18418162],
       [ 2.3652936 ,  1.62465845,  1.7133191 ,  1.3406709 ],
```

```
[-1.18417675,  0.079117  , -1.12597494, -1.22930526],
[ 1.01894278, -0.36246627,  0.54419802,  0.18418162],
[-0.32740804, -1.46642445,  0.04314613, -0.2013148 ],
[-0.44980357,  1.40386682, -1.18164737, -1.22930526],
[-0.20501251, -0.36246627, -0.0125263 ,  0.18418162],
[ 0.77415172,  0.29990864,  0.48852559,  0.44117924],
[-0.69459463,  0.74149191, -1.2373198 , -1.22930526],
[ 2.61008466,  1.62465845,  1.5463018 ,  1.08367328],
[ 1.38612937,  0.079117  ,  0.98957748,  1.21217209],
[ 0.16217408,  0.29990864,  0.65554289,  0.82667566],
[ 1.14133831,  0.079117  ,  1.10092235,  1.59766851],
[-0.32740804, -1.46642445,  0.09881857, -0.07281599],
[ 1.14133831,  0.52070027,  1.15659478,  1.21217209],
[ 2.3652936 , -0.5832579 ,  1.7133191 ,  1.08367328],
[-0.08261698, -0.5832579 ,  0.48852559,  0.18418162],
[ 1.26373384, -0.5832579 ,  0.65554289,  0.31268043],
[-0.81699016,  0.74149191, -1.18164737, -1.22930526],
[ 0.40696513, -0.5832579 ,  0.59987046,  0.05568282],
[ 0.28456961, -0.36246627,  0.48852559,  0.44117924],
[ 1.38612937,  0.29990864,  1.15659478,  1.4691697 ],
[-1.06178122,  0.079117  , -1.18164737, -1.35780407],
[-0.08261698, -1.24563281,  0.76688775,  1.08367328],
[ 0.03977855, -0.80404954,  0.26583586, -0.2013148 ],
[-0.20501251, -0.14167463,  0.26583586,  0.18418162],
[ 0.16217408, -0.14167463,  0.3215083 ,  0.44117924],
[ 0.77415172,  0.079117  ,  1.04524991,  0.82667566],
[ 0.77415172, -0.36246627,  0.37718073,  0.18418162]])
```

[200]:
```
# Explain the importance of feature scaling in KNN

# KNN uses distance based calculations (Euclidean distance).
# Without scaling, features with larger ranges will dominate the distance␣
 ↪metric.
# Standardizing makes the features unitless and
# Ensures that all features contribute equally as the values are scaled (mostly␣
 ↪from -3 to 3)
```

1.3 3. Implementing K-Nearest Neighbors

[203]:
```
# Use scikit-learn's KNeighborsClassifier to train the KNN model
# Train the model using the default parameters (n neighbors=5,␣
 ↪metric='minkowski', p=2 for Euclidean distance)

knn = KNeighborsClassifier(n_neighbors=5)
```

[204]:
```
# Fit the model on the training dataset and test the performance in terms of F1␣
↪score, precision and recall on the test set

knn.fit(X_train_scaled, y_train)
y_pred = knn.predict(X_test_scaled)
```

[207]:
```
accuracy = accuracy_score(y_test, y_pred)
precision = precision_score(y_test, y_pred, average='weighted')
recall = recall_score(y_test, y_pred, average='weighted')
f1 = f1_score(y_test, y_pred, average='weighted')

print("Accuracy", round(accuracy, 3))
print("precision", round(precision, 3))
print("recall", round(recall, 3))
print("f1", round(f1, 3))
```

```
Accuracy 0.933
precision 0.945
recall 0.933
f1 0.934
```

[209]:
```
# Evaluate the impact of different values of k (n neighbors)
# Train the model for different values of k ranging from 1 to 20

k_values = range(1, 21)

accuracies = []
precisions = []
recalls = []
f1_scores = []

for k in k_values:
    knn = KNeighborsClassifier(n_neighbors=k)
    knn.fit(X_train_scaled, y_train)
    y_pred_k = knn.predict(X_test_scaled)

    accuracies.append(accuracy_score(y_test, y_pred_k))
    precisions.append(precision_score(y_test, y_pred_k, average='weighted'))
    recalls.append(recall_score(y_test, y_pred_k, average='weighted'))
    f1_scores.append(f1_score(y_test, y_pred_k, average='weighted'))
```

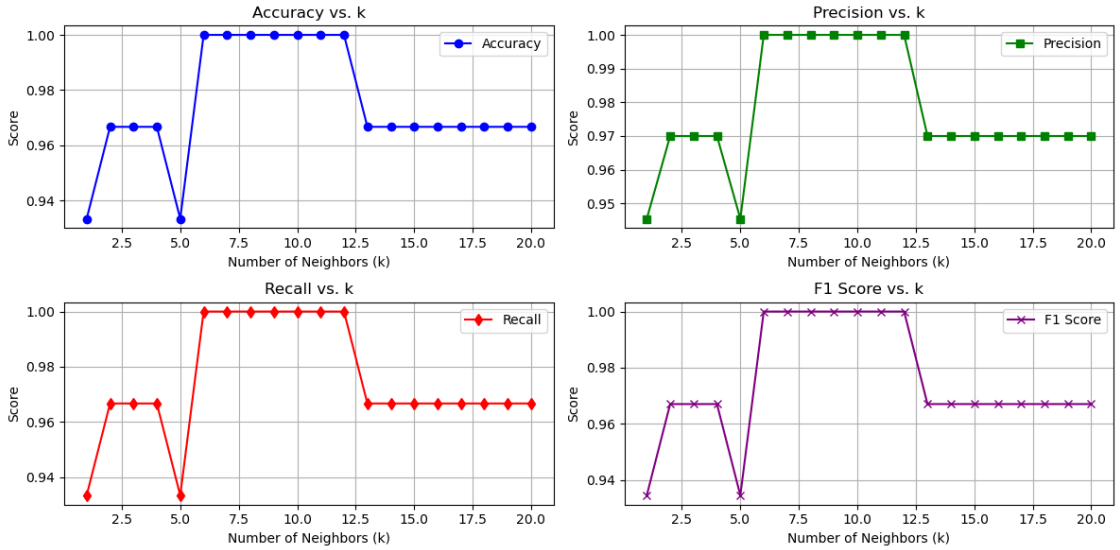
[211]:
```
print(accuracies)
print(precisions)
```

```
[0.9333333333333333, 0.9666666666666667, 0.9666666666666667, 0.9666666666666667,
0.9333333333333333, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 0.9666666666666667,
0.9666666666666667, 0.9666666666666667, 0.9666666666666667, 0.9666666666666667,
```

```
0.9666666666666667, 0.9666666666666667, 0.9666666666666667]
[0.9454545454545454, 0.9700000000000001, 0.9700000000000001, 0.9700000000000001,
0.9454545454545454, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 0.9700000000000001,
0.9700000000000001, 0.9700000000000001, 0.9700000000000001, 0.9700000000000001,
0.9700000000000001, 0.9700000000000001, 0.9700000000000001]
```

[213]:
```
# Create a line plot of k vs. accuracy (for classification)
plt.figure(figsize=(12, 6))

plt.subplot(2, 2, 1)
plt.plot(k_values, accuracies, marker='o', linestyle='-', color='b',␣
 ↪label="Accuracy")
plt.xlabel("Number of Neighbors (k)")
plt.ylabel("Score")
plt.title("Accuracy vs. k")
plt.grid()
plt.legend()

plt.subplot(2, 2, 2)
plt.plot(k_values, precisions, marker='s', linestyle='-', color='g',␣
 ↪label="Precision")
plt.xlabel("Number of Neighbors (k)")
plt.ylabel("Score")
plt.title("Precision vs. k")
plt.grid()
plt.legend()

plt.subplot(2, 2, 3)
plt.plot(k_values, recalls, marker='d', linestyle='-', color='r',␣
 ↪label="Recall")
plt.xlabel("Number of Neighbors (k)")
plt.ylabel("Score")
plt.title("Recall vs. k")
plt.grid()
plt.legend()

plt.subplot(2, 2, 4)
plt.plot(k_values, f1_scores, marker='x', linestyle='-', color='purple',␣
 ↪label="F1 Score")
plt.xlabel("Number of Neighbors (k)")
plt.ylabel("Score")
plt.title("F1 Score vs. k")
plt.grid()
plt.legend()

# Show all plots
plt.tight_layout()
```

```
plt.show()
```



[215]:
```
# Find the optimal value of k.

# The best k is where accuracy is highest
# We are obtaining highest accuracy with k = 7
```

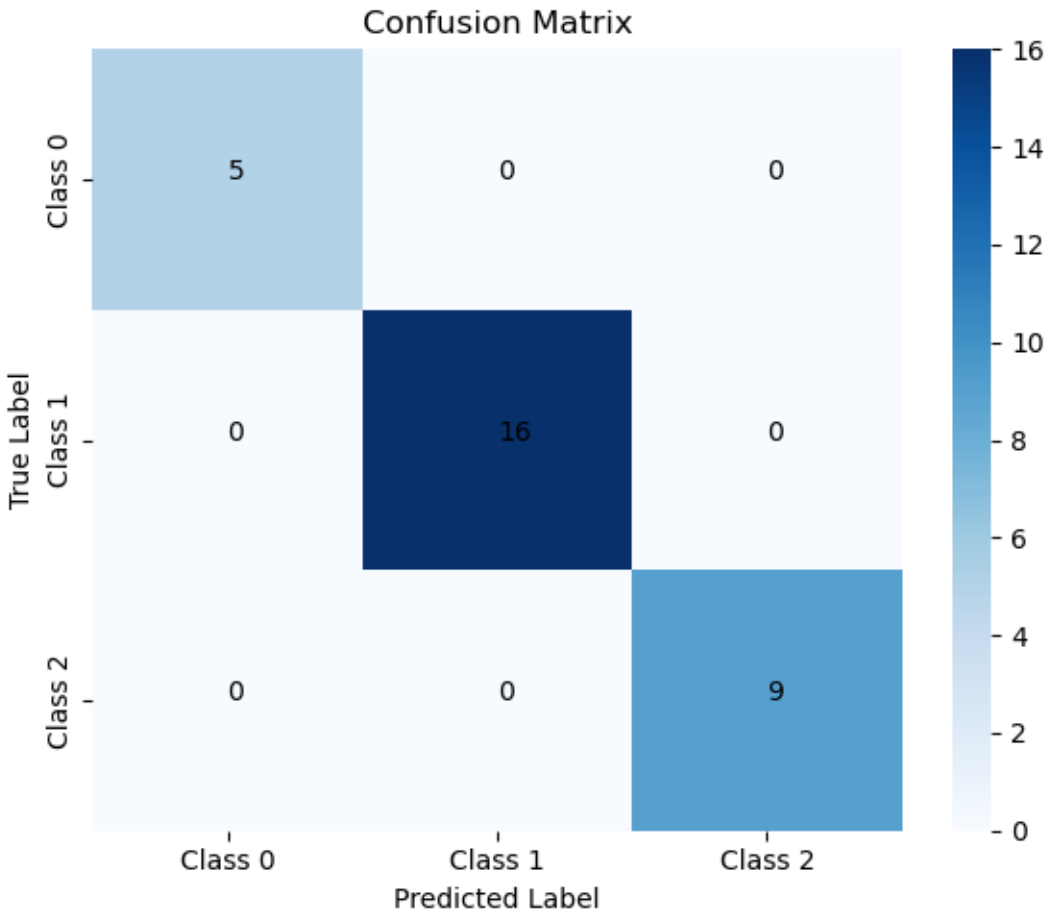
[217]:
```
# Question: What value of k gives the best performance? Explain why the choice␣
 ↪of k affects the model's performance

# According to theory k ranges around = sqrt(N) (where N is number of␣
 ↪observations)
# Choosing too Small k or large L affects model performance in following ways:

# Too Small K leads to High variance (overfitting) (We assign label to test␣
 ↪data as same as its single closest training point.
# This makes the model highly sensitive to noise in the data. Even slighest␣
 ↪variation in data changes the predictions)

# Too Large K leads to  High bias (underfitting) (the model assigns the most␣
 ↪common label in the dataset to all points.
# This causes the model to ignore local patterns in the data)
```

1.4 4. Model Evaluation

[220]:
```
# Generate a confusion matrix and a classification report (using classification␣
↪report) for y test predictions.

knn = KNeighborsClassifier(n_neighbors=7)
knn.fit(X_train_scaled, y_train)
y_pred = knn.predict(X_test_scaled)

cm = confusion_matrix(y_test, y_pred)
cm
```

[220]:
```
array([[ 5,  0,  0],
       [ 0, 16,  0],
       [ 0,  0,  9]])
```

[222]:
```
classes = ['Class 0', 'Class 1', 'Class 2']

plt.figure(figsize=(6, 5))
sns.heatmap(cm,
            cmap='Blues',
            xticklabels=classes,
            yticklabels=classes)

for i in range(3):
    for j in range(3):
        plt.text(j + 0.5, i + 0.5, str(cm[i, j]))

plt.xlabel('Predicted Label')
plt.ylabel('True Label')
plt.title('Confusion Matrix')
plt.tight_layout()
plt.show()
```



[224]:
```
print("Classification Report:")
print(classification_report(y_test, y_pred))
```

```
Classification Report:
              precision    recall  f1-score   support

           0       1.00      1.00      1.00         5
           1       1.00      1.00      1.00        16
           2       1.00      1.00      1.00         9

    accuracy                           1.00        30
   macro avg       1.00      1.00      1.00        30
weighted avg       1.00      1.00      1.00        30
```

[226]:
```
# Interpret the confusion matrix and explain the precision, recall, and␣
↪F1-score for each target class.
```

```
# Confusion Matrix Helps interpret classification performance
# Diagonal elements = correctly classified samples per class.
# Off-diagonal elements = misclassified samples

# Precision: Out of all predictions for a class, how many were correct
# Precision=  (True Positives + False Positives) / (True Positives)
# Recall (Sensitivity): How many actual samples of a class were correctly␣
 ↪predicted?
# Recall = (True Positives) / (True Positives + False Negatives)
# F1 Score: Balance between Precision & Recall
# F1 = 2 * (Precision + Recall) / (Precision * Recall)
```

[228]:
```
# Tune the KNN model's hyperparameters.
# Perform grid search or random search (GridSearchCV
# Number of neighbors (n_neighbors).
# Distance metrics (metric, e.g., Euclidean, Manhattan).
# Weighting schemes (weights, e.g., uniform, distance).

param_grid = {
    'n_neighbors': range(1, 21),
    'metric': ['euclidean', 'manhattan'],
    'weights': ['uniform', 'distance']
}

grid_search = GridSearchCV(KNeighborsClassifier(), param_grid, cv=5,␣
 ↪scoring='accuracy', n_jobs=-1)
grid_search.fit(X_train_scaled, y_train)

print("Best Parameters:", grid_search.best_params_)
print("Best Accuracy Score:", grid_search.best_score_)

# Evaluate on test data
best_knn = grid_search.best_estimator_
y_pred_best = best_knn.predict(X_test_scaled)
print("\nClassification Report for Best Model:")
print(classification_report(y_test, y_pred_best))
```

```
Best Parameters: {'metric': 'manhattan', 'n_neighbors': 13, 'weights':
'uniform'}
Best Accuracy Score: 0.9666666666666668

Classification Report for Best Model:
              precision    recall  f1-score   support

           0       1.00      1.00      1.00         5
           1       1.00      0.94      0.97        16
           2       0.90      1.00      0.95         9
```

```
    accuracy                           0.97        30
   macro avg       0.97      0.98      0.97        30
weighted avg       0.97      0.97      0.97        30
```

[230]:
```
# Question: How does choosing different distance metrics (e.g., Euclidean vs.␣
 ↪Manhattan) affect model performance?

# If features are normalized (StandardScaler), Euclidean distance usually␣
 ↪performs better.
# If features are sparse or have high variation, Manhattan distance may be more␣
 ↪robust.
```

1.5 5. Comparison with Other Algorithms

[233]:
```
# Compare KNN's performance with other models like:
# Multinomial Logistic Regression.
# Random Forest

knn = KNeighborsClassifier(n_neighbors=7)
log_reg = LogisticRegression(multi_class='multinomial', solver='lbfgs',␣
 ↪max_iter=200)
rf = RandomForestClassifier(n_estimators=100, random_state=42)

knn.fit(X_train_scaled, y_train)
log_reg.fit(X_train_scaled, y_train)
rf.fit(X_train_scaled, y_train)

y_pred_knn = knn.predict(X_test_scaled)
y_pred_log = log_reg.predict(X_test_scaled)
y_pred_rf = rf.predict(X_test_scaled)

models = {'KNN': y_pred_knn, 'Logistic Regression': y_pred_log, 'Random Forest':
 ↪ y_pred_rf}
for model, y_pred in models.items():
    print("Model: ", model)
    print(classification_report(y_test, y_pred))
    print("Accuracy: ", round(accuracy_score(y_test, y_pred), 3))
```

```
Model:  KNN
              precision    recall  f1-score   support

           0       1.00      1.00      1.00         5
           1       1.00      1.00      1.00        16
           2       1.00      1.00      1.00         9
```

```
    accuracy                           1.00        30
   macro avg       1.00      1.00      1.00        30
weighted avg       1.00      1.00      1.00        30

Accuracy:  1.0
Model:  Logistic Regression
              precision    recall  f1-score   support

           0       1.00      1.00      1.00         5
           1       1.00      0.94      0.97        16
           2       0.90      1.00      0.95         9

    accuracy                           0.97        30
   macro avg       0.97      0.98      0.97        30
weighted avg       0.97      0.97      0.97        30

Accuracy:  0.967
Model:  Random Forest
              precision    recall  f1-score   support

           0       1.00      1.00      1.00         5
           1       1.00      0.94      0.97        16
           2       0.90      1.00      0.95         9

    accuracy                           0.97        30
   macro avg       0.97      0.98      0.97        30
weighted avg       0.97      0.97      0.97        30

Accuracy:  0.967
```

[235]:
```
# KNN seems to classify best compared with Multinomial logistic and Random␣
↪forest
```

[237]:
```
# Explore the impact of dropping certain features on the KNN model's␣
↪performance.

feature_names = iris.feature_names
knn = KNeighborsClassifier(n_neighbors=7)

results = {}

for i in range(1, 5):
    for combo in combinations(range(4), i):
        selected_features = [feature_names[j] for j in combo]

        X_train_sub = X_train[selected_features]
        X_test_sub = X_test[selected_features]
```

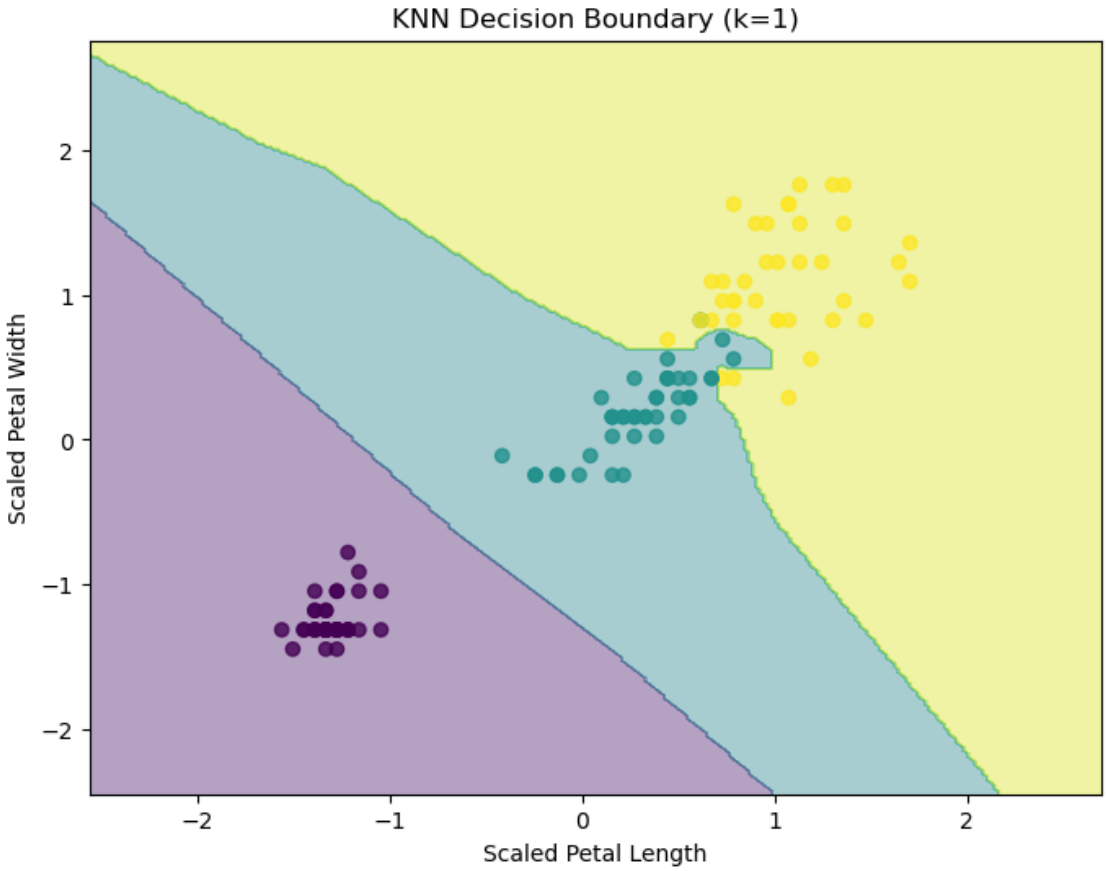
```
        X_train_scaled = scaler.fit_transform(X_train_sub)
        X_test_scaled = scaler.transform(X_test_sub)
        knn.fit(X_train_scaled, y_train)
        y_pred_sub = knn.predict(X_test_scaled)
        acc = accuracy_score(y_test, y_pred_sub)

        results[tuple(selected_features)] = acc

for feat_set, acc in results.items():
    print("Features:", feat_set, "→ Accuracy:", round(acc, 4))
```

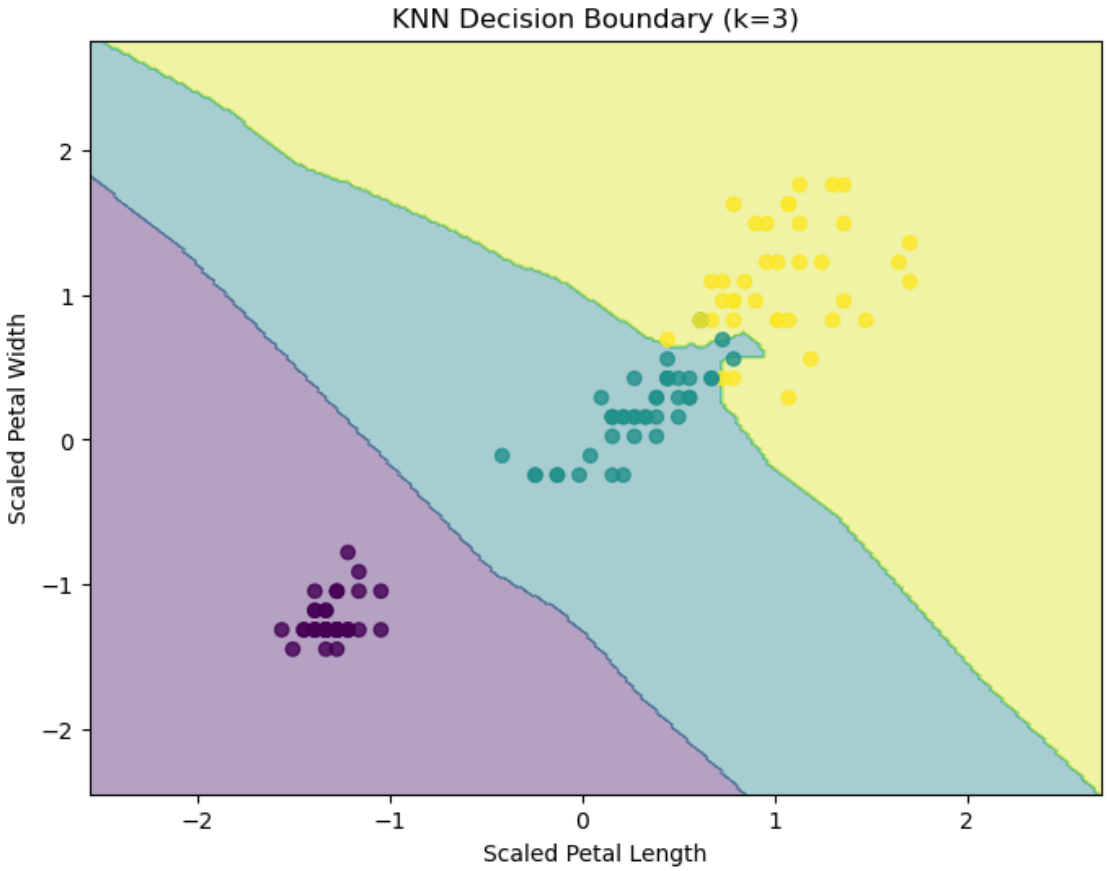
```
Features: ('sepal length (cm)',) → Accuracy: 0.7333
Features: ('sepal width (cm)',) → Accuracy: 0.3667
Features: ('petal length (cm)',) → Accuracy: 1.0
Features: ('petal width (cm)',) → Accuracy: 0.9667
Features: ('sepal length (cm)', 'sepal width (cm)') → Accuracy: 0.5667
Features: ('sepal length (cm)', 'petal length (cm)') → Accuracy: 0.9667
Features: ('sepal length (cm)', 'petal width (cm)') → Accuracy: 0.9667
Features: ('sepal width (cm)', 'petal length (cm)') → Accuracy: 0.9
Features: ('sepal width (cm)', 'petal width (cm)') → Accuracy: 0.9667
Features: ('petal length (cm)', 'petal width (cm)') → Accuracy: 0.9667
Features: ('sepal length (cm)', 'sepal width (cm)', 'petal length (cm)') →
Accuracy: 0.8333
Features: ('sepal length (cm)', 'sepal width (cm)', 'petal width (cm)') →
Accuracy: 0.9667
Features: ('sepal length (cm)', 'petal length (cm)', 'petal width (cm)') →
Accuracy: 0.9667
Features: ('sepal width (cm)', 'petal length (cm)', 'petal width (cm)') →
Accuracy: 0.9667
Features: ('sepal length (cm)', 'sepal width (cm)', 'petal length (cm)', 'petal
width (cm)') → Accuracy: 1.0
```

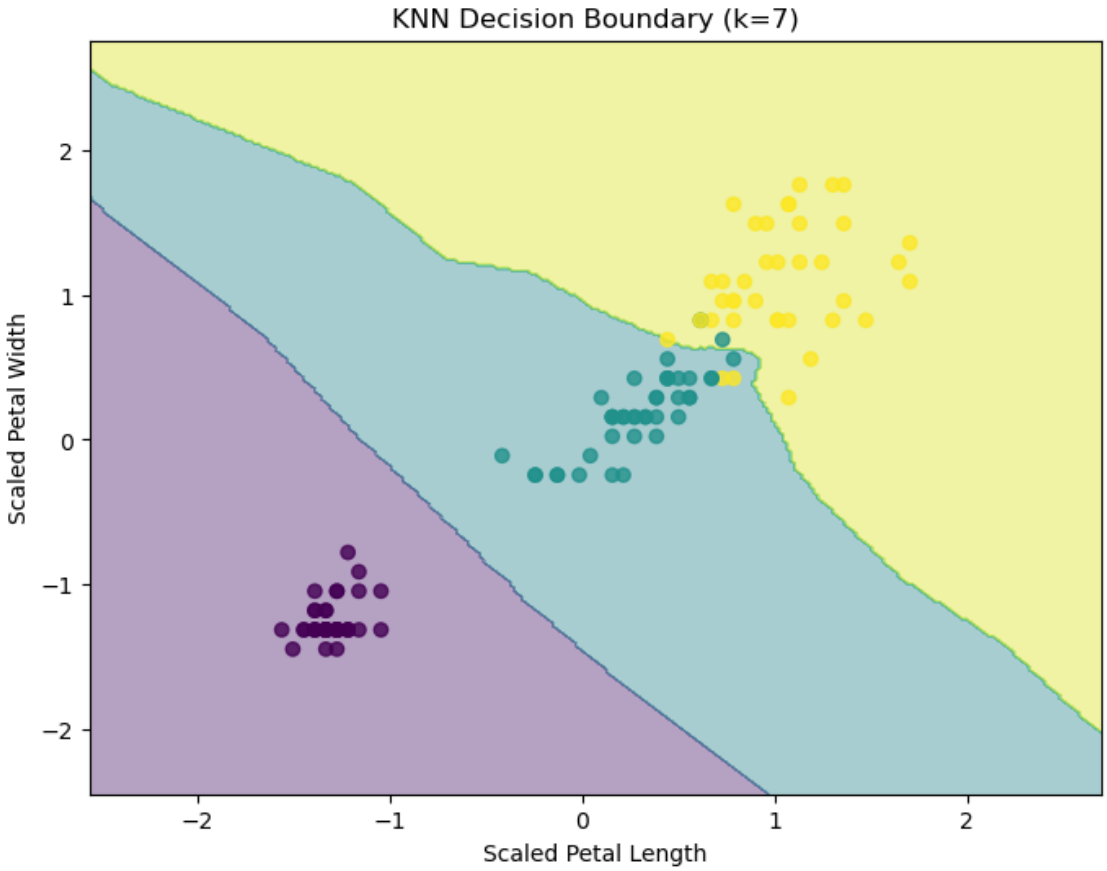
[239]:
```
# Dropping the features is impacting accuracies. So its better to consider all␣
↪features
```

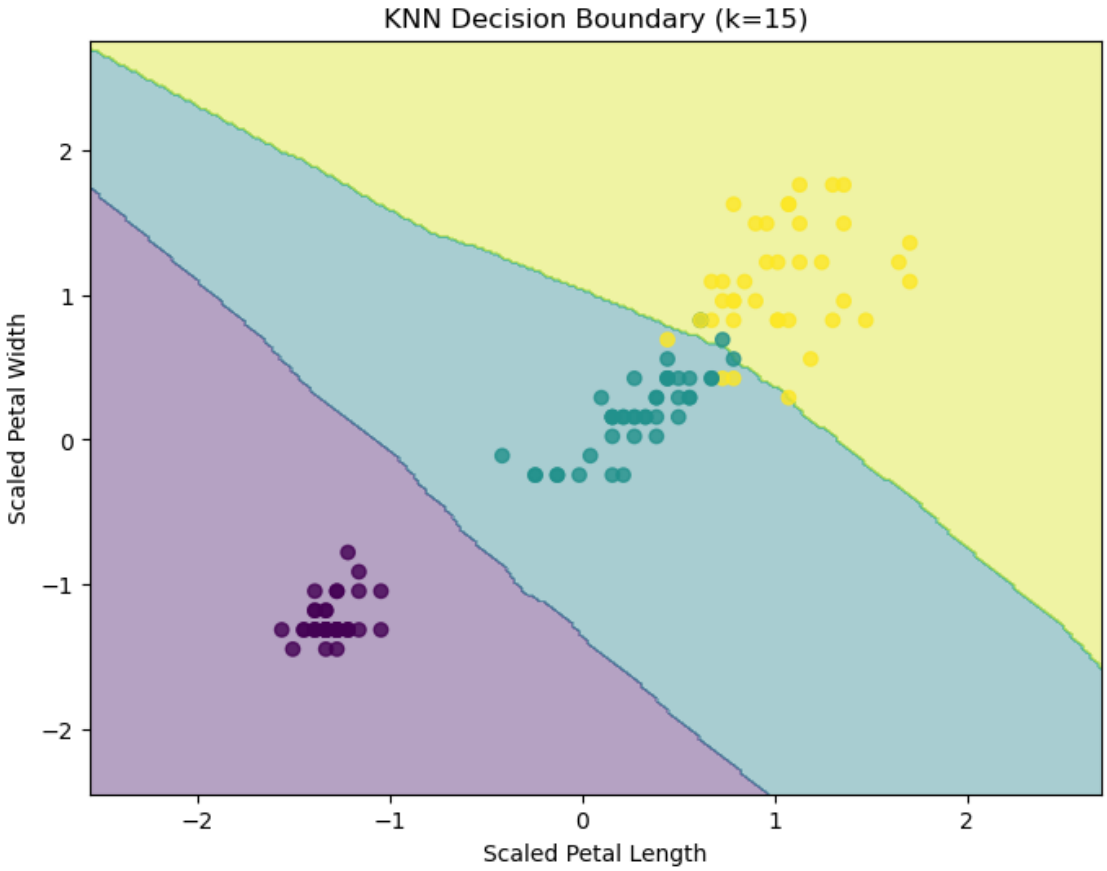
[241]:
```
# Visualization: For 2D datasets (e.g., Iris), visualize decision boundaries␣
↪using matplotlib for different values of k.

import numpy as np
import matplotlib.pyplot as plt
from sklearn.datasets import load_iris
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler
from sklearn.neighbors import KNeighborsClassifier

# Load the iris dataset
iris = load_iris()
```

```
X = iris.data[:, [2, 3]]  # Using petal length and petal width
y = iris.target

# Split the data
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2,␣
 ↪random_state=42)

# Scale the features
scaler = StandardScaler()
X_train_scaled = scaler.fit_transform(X_train)
X_test_scaled = scaler.transform(X_test)

# Function to plot decision boundaries
def plot_decision_boundary(k):
    # Create classifier
    knn = KNeighborsClassifier(n_neighbors=k)
    knn.fit(X_train_scaled, y_train)

    # Create mesh grid
    x_min, x_max = X_train_scaled[:, 0].min() - 1, X_train_scaled[:, 0].max() +␣
 ↪1
    y_min, y_max = X_train_scaled[:, 1].min() - 1, X_train_scaled[:, 1].max() +␣
 ↪1
    xx, yy = np.meshgrid(np.arange(x_min, x_max, 0.02),
                         np.arange(y_min, y_max, 0.02))

    # Predict for each point in mesh
    Z = knn.predict(np.c_[xx.ravel(), yy.ravel()])
    Z = Z.reshape(xx.shape)

    # Plot
    plt.figure(figsize=(8, 6))
    plt.contourf(xx, yy, Z, alpha=0.4)
    plt.scatter(X_train_scaled[:, 0], X_train_scaled[:, 1], c=y_train, alpha=0.
 ↪8)
    plt.xlabel('Scaled Petal Length')
    plt.ylabel('Scaled Petal Width')
    plt.title(f'KNN Decision Boundary (k={k})')
    plt.show()

# Plot decision boundaries for different k values
k_values = [1, 3, 7, 15]
for k in k_values:
    plot_decision_boundary(k)
```


KNN Decision Boundary (k=1)
Scaled Petal Width
Scaled Petal Length
2
1
0
−1
−2
−2
−1
0
1
2


KNN Decision Boundary (k=3)
Scaled Petal Width
Scaled Petal Length
2
1
0
−1
−2
−2
−1
0
1
2

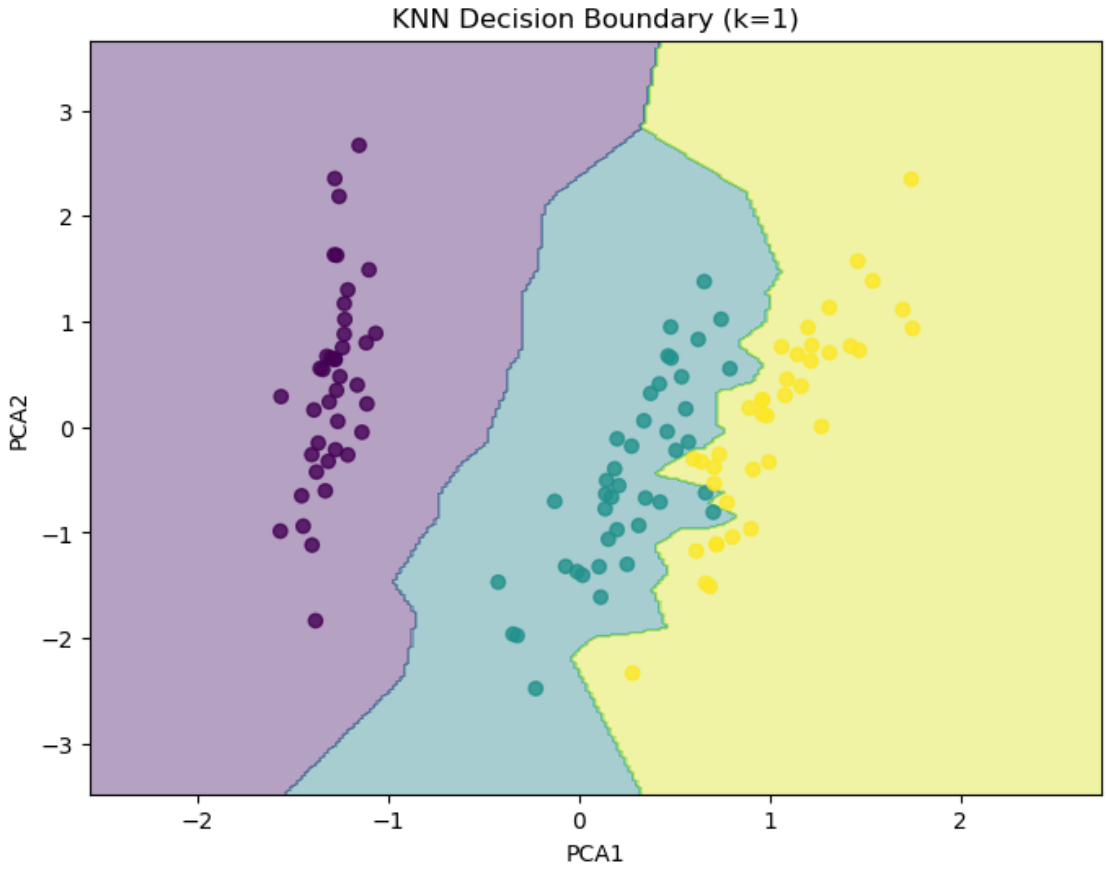

KNN Decision Boundary (k=7)
Scaled Petal Width
Scaled Petal Length
2
1
0
−1
−2
−2
−1
0
1
2

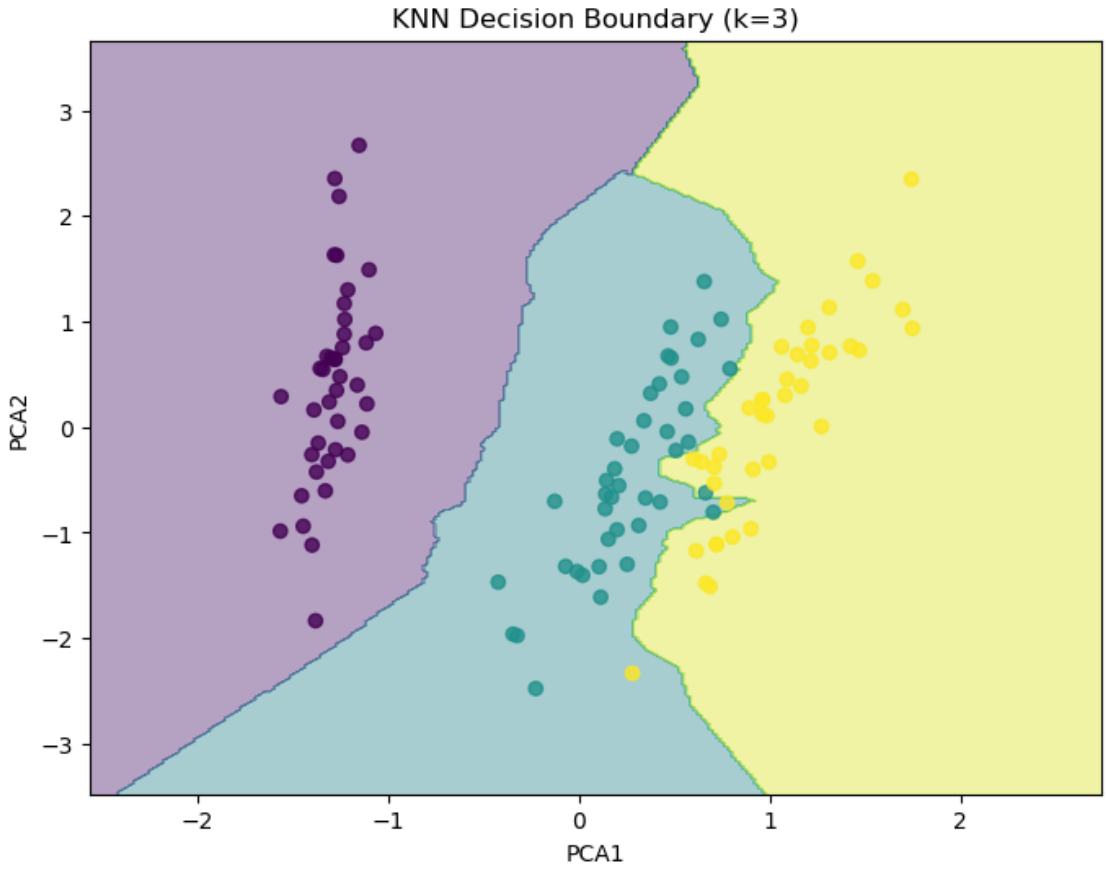


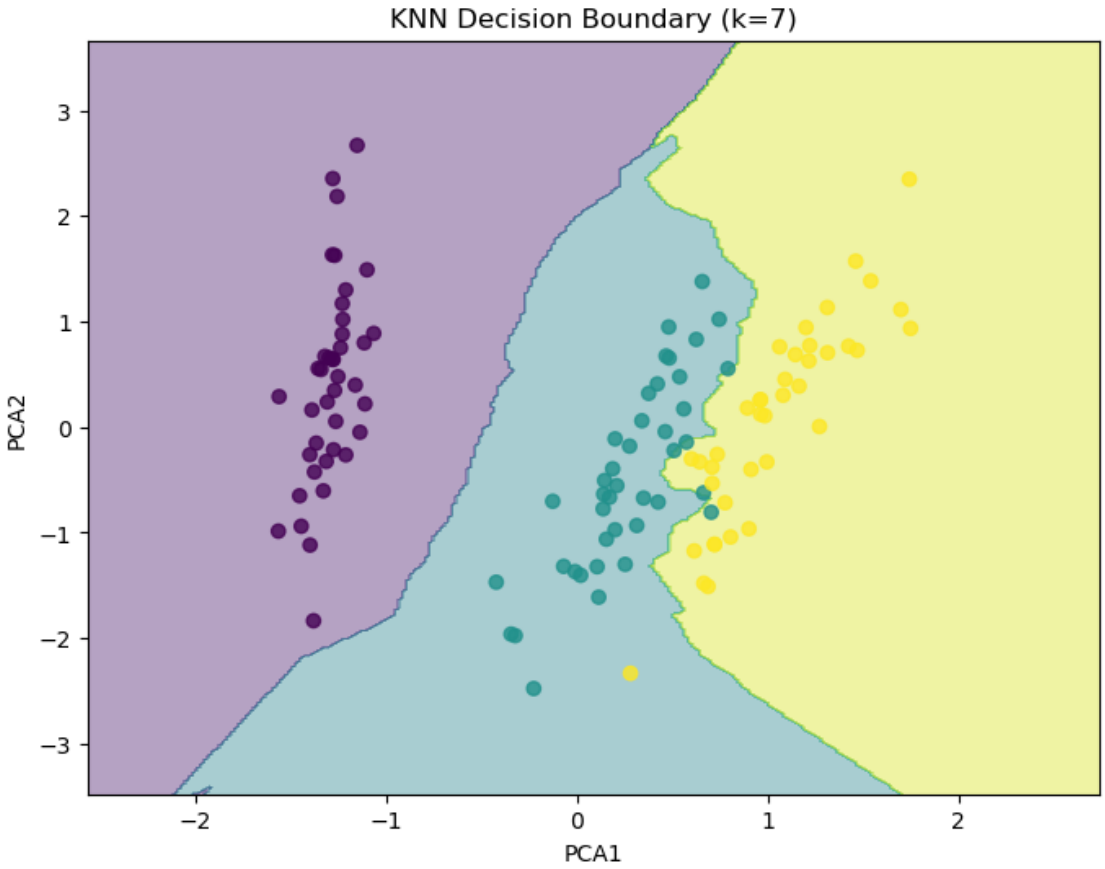
[42]:
```
# With PCA

# Load the iris dataset
iris = load_iris()
X = iris.data
y = iris.target

pca = PCA(n_components=2)
X_pca = pca.fit_transform(X)


# Split the data
X_train, X_test, y_train, y_test = train_test_split(X_pca, y, test_size=0.2,␣
 ↪random_state=42)
```

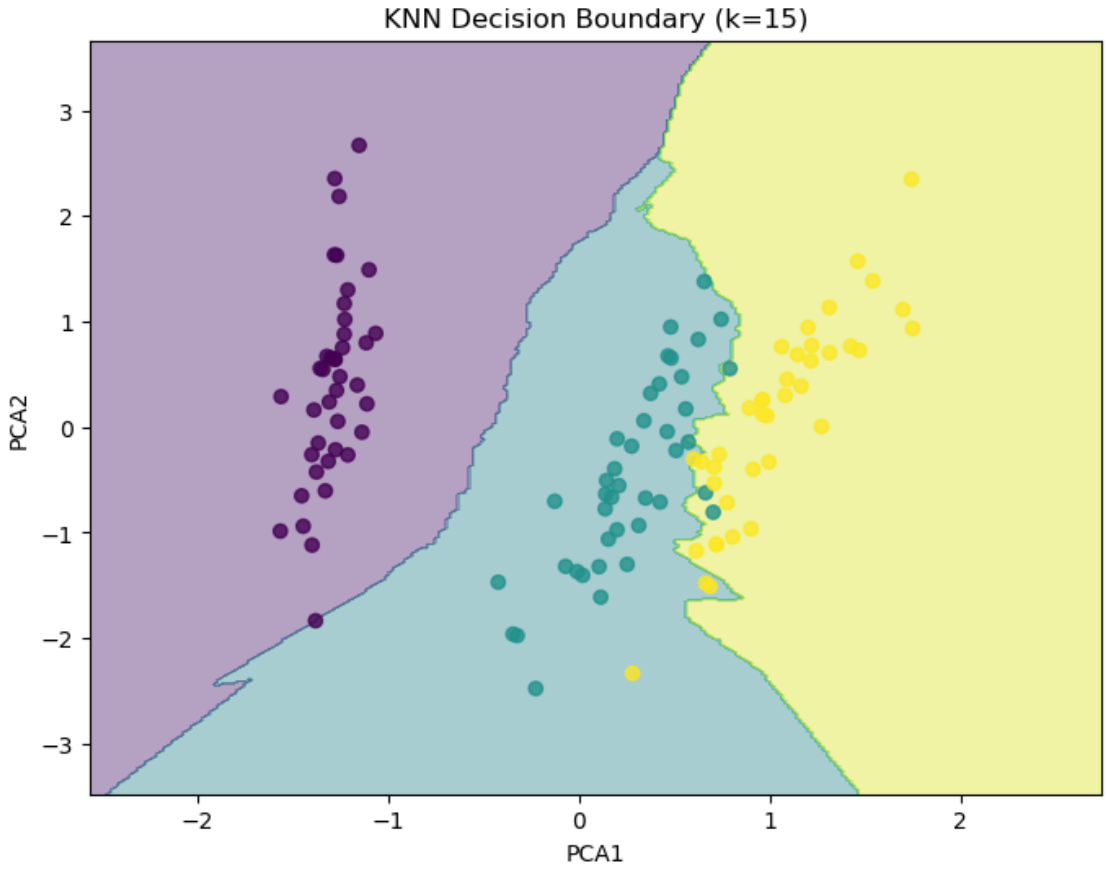
[43]:
```
# Scale the features
scaler = StandardScaler()
X_train_scaled = scaler.fit_transform(X_train)
X_test_scaled = scaler.transform(X_test)
```

```python
# Function to plot decision boundaries
def plot_decision_boundary(k):
    # Create classifier
    knn = KNeighborsClassifier(n_neighbors=k)
    knn.fit(X_train_scaled, y_train)

    # Create mesh grid
    x_min, x_max = X_train_scaled[:, 0].min() - 1, X_train_scaled[:, 0].max() +␣
 ↪1
    y_min, y_max = X_train_scaled[:, 1].min() - 1, X_train_scaled[:, 1].max() +␣
 ↪1
    xx, yy = np.meshgrid(np.arange(x_min, x_max, 0.02),
                         np.arange(y_min, y_max, 0.02))

    # Predict for each point in mesh
    Z = knn.predict(np.c_[xx.ravel(), yy.ravel()])
    Z = Z.reshape(xx.shape)

    # Plot
    plt.figure(figsize=(8, 6))
    plt.contourf(xx, yy, Z, alpha=0.4)
    plt.scatter(X_train_scaled[:, 0], X_train_scaled[:, 1], c=y_train, alpha=0.
 ↪8)
    plt.xlabel('PCA1')
    plt.ylabel('PCA2')
    plt.title(f'KNN Decision Boundary (k={k})')
    plt.show()

# Plot decision boundaries for different k values
k_values = [1, 3, 7, 15]
for k in k_values:
    plot_decision_boundary(k)
```


KNN Decision Boundary (k=1)
PCA2
PCA1
3
2
1
0
−1
−2
−3
−2
−1
0
1
2


KNN Decision Boundary (k=3)
PCA2
PCA1
3
2
1
0
−1
−2
−3
−2
−1
0
1
2


KNN Decision Boundary (k=7)
PCA2
PCA1
3
2
1
0
−1
−2
−3
−2
−1
0
1
2



[44]:
```
# We should consider 2D visualization by doing PCA, instead of selecting␣
↪features randomly
```

[45]:
```
# Comment on how decision boundaries change as k increases.

# As k increases, we notice these changes in the decision boundaries:
# With k=1, the boundaries are very jagged and complex, potentially overfitting
# As k increases (3, 7), the boundaries become smoother and more generalized
# With large k (15), the boundaries become very smooth but might oversmooth,␣
↪potentially underfitting
# The decision regions become larger and more stable with higher k values
```

[ ]:

[ ]:

[ ]:

2 Implementing K-Means Clustering from Scratch (Iris Dataset)

2.1 1. Dataset Preparation

[48]:
```
# Load the Iris dataset from sklearn.datasets
iris = load_iris()

df = pd.DataFrame(iris.data, columns=iris.feature_names)
# Use only the numeric features (sepal length, sepal width, petal length, petal␣
 ↪width)
df.columns = ["sepal_length", "sepal_width", "petal_length", "petal_width"]

print(df.head())
print(df.shape)
```

```
   sepal_length  sepal_width  petal_length  petal_width
0           5.1          3.5           1.4          0.2
1           4.9          3.0           1.4          0.2
2           4.7          3.2           1.3          0.2
3           4.6          3.1           1.5          0.2
4           5.0          3.6           1.4          0.2
(150, 4)
```

[49]:
```
# Normalize the features using a scaler (e.g., StandardScaler or MinMaxScaler)
scaler = StandardScaler()
X = scaler.fit_transform(df)
X
```

[49]:
```
array([[-9.00681170e-01,  1.01900435e+00, -1.34022653e+00,
        -1.31544430e+00],
       [-1.14301691e+00, -1.31979479e-01, -1.34022653e+00,
        -1.31544430e+00],
       [-1.38535265e+00,  3.28414053e-01, -1.39706395e+00,
        -1.31544430e+00],
       [-1.50652052e+00,  9.82172869e-02, -1.28338910e+00,
        -1.31544430e+00],
       [-1.02184904e+00,  1.24920112e+00, -1.34022653e+00,
        -1.31544430e+00],
       [-5.37177559e-01,  1.93979142e+00, -1.16971425e+00,
        -1.05217993e+00],
       [-1.50652052e+00,  7.88807586e-01, -1.34022653e+00,
        -1.18381211e+00],
       [-1.02184904e+00,  7.88807586e-01, -1.28338910e+00,
        -1.31544430e+00],
       [-1.74885626e+00, -3.62176246e-01, -1.34022653e+00,
        -1.31544430e+00],
       [-1.14301691e+00,  9.82172869e-02, -1.28338910e+00,
        -1.44707648e+00],
```

```
[-5.37177559e-01,  1.47939788e+00, -1.28338910e+00,
 -1.31544430e+00],
[-1.26418478e+00,  7.88807586e-01, -1.22655167e+00,
 -1.31544430e+00],
[-1.26418478e+00, -1.31979479e-01, -1.34022653e+00,
 -1.44707648e+00],
[-1.87002413e+00, -1.31979479e-01, -1.51073881e+00,
 -1.44707648e+00],
[-5.25060772e-02,  2.16998818e+00, -1.45390138e+00,
 -1.31544430e+00],
[-1.73673948e-01,  3.09077525e+00, -1.28338910e+00,
 -1.05217993e+00],
[-5.37177559e-01,  1.93979142e+00, -1.39706395e+00,
 -1.05217993e+00],
[-9.00681170e-01,  1.01900435e+00, -1.34022653e+00,
 -1.18381211e+00],
[-1.73673948e-01,  1.70959465e+00, -1.16971425e+00,
 -1.18381211e+00],
[-9.00681170e-01,  1.70959465e+00, -1.28338910e+00,
 -1.18381211e+00],
[-5.37177559e-01,  7.88807586e-01, -1.16971425e+00,
 -1.31544430e+00],
[-9.00681170e-01,  1.47939788e+00, -1.28338910e+00,
 -1.05217993e+00],
[-1.50652052e+00,  1.24920112e+00, -1.56757623e+00,
 -1.31544430e+00],
[-9.00681170e-01,  5.58610819e-01, -1.16971425e+00,
 -9.20547742e-01],
[-1.26418478e+00,  7.88807586e-01, -1.05603939e+00,
 -1.31544430e+00],
[-1.02184904e+00, -1.31979479e-01, -1.22655167e+00,
 -1.31544430e+00],
[-1.02184904e+00,  7.88807586e-01, -1.22655167e+00,
 -1.05217993e+00],
[-7.79513300e-01,  1.01900435e+00, -1.28338910e+00,
 -1.31544430e+00],
[-7.79513300e-01,  7.88807586e-01, -1.34022653e+00,
 -1.31544430e+00],
[-1.38535265e+00,  3.28414053e-01, -1.22655167e+00,
 -1.31544430e+00],
[-1.26418478e+00,  9.82172869e-02, -1.22655167e+00,
 -1.31544430e+00],
[-5.37177559e-01,  7.88807586e-01, -1.28338910e+00,
 -1.05217993e+00],
[-7.79513300e-01,  2.40018495e+00, -1.28338910e+00,
 -1.44707648e+00],
[-4.16009689e-01,  2.63038172e+00, -1.34022653e+00,
```

```
 -1.31544430e+00],
[-1.14301691e+00,  9.82172869e-02, -1.28338910e+00,
 -1.31544430e+00],
[-1.02184904e+00,  3.28414053e-01, -1.45390138e+00,
 -1.31544430e+00],
[-4.16009689e-01,  1.01900435e+00, -1.39706395e+00,
 -1.31544430e+00],
[-1.14301691e+00,  1.24920112e+00, -1.34022653e+00,
 -1.44707648e+00],
[-1.74885626e+00, -1.31979479e-01, -1.39706395e+00,
 -1.31544430e+00],
[-9.00681170e-01,  7.88807586e-01, -1.28338910e+00,
 -1.31544430e+00],
[-1.02184904e+00,  1.01900435e+00, -1.39706395e+00,
 -1.18381211e+00],
[-1.62768839e+00, -1.74335684e+00, -1.39706395e+00,
 -1.18381211e+00],
[-1.74885626e+00,  3.28414053e-01, -1.39706395e+00,
 -1.31544430e+00],
[-1.02184904e+00,  1.01900435e+00, -1.22655167e+00,
 -7.88915558e-01],
[-9.00681170e-01,  1.70959465e+00, -1.05603939e+00,
 -1.05217993e+00],
[-1.26418478e+00, -1.31979479e-01, -1.34022653e+00,
 -1.18381211e+00],
[-9.00681170e-01,  1.70959465e+00, -1.22655167e+00,
 -1.31544430e+00],
[-1.50652052e+00,  3.28414053e-01, -1.34022653e+00,
 -1.31544430e+00],
[-6.58345429e-01,  1.47939788e+00, -1.28338910e+00,
 -1.31544430e+00],
[-1.02184904e+00,  5.58610819e-01, -1.34022653e+00,
 -1.31544430e+00],
[ 1.40150837e+00,  3.28414053e-01,  5.35408562e-01,
  2.64141916e-01],
[ 6.74501145e-01,  3.28414053e-01,  4.21733708e-01,
  3.95774101e-01],
[ 1.28034050e+00,  9.82172869e-02,  6.49083415e-01,
  3.95774101e-01],
[-4.16009689e-01, -1.74335684e+00,  1.37546573e-01,
  1.32509732e-01],
[ 7.95669016e-01, -5.92373012e-01,  4.78571135e-01,
  3.95774101e-01],
[-1.73673948e-01, -5.92373012e-01,  4.21733708e-01,
  1.32509732e-01],
[ 5.53333275e-01,  5.58610819e-01,  5.35408562e-01,
  5.27406285e-01],
```

```
[-1.14301691e+00, -1.51316008e+00, -2.60315415e-01,
 -2.62386821e-01],
[ 9.16836886e-01, -3.62176246e-01,  4.78571135e-01,
  1.32509732e-01],
[-7.79513300e-01, -8.22569778e-01,  8.07091462e-02,
  2.64141916e-01],
[-1.02184904e+00, -2.43394714e+00, -1.46640561e-01,
 -2.62386821e-01],
[ 6.86617933e-02, -1.31979479e-01,  2.51221427e-01,
  3.95774101e-01],
[ 1.89829664e-01, -1.97355361e+00,  1.37546573e-01,
 -2.62386821e-01],
[ 3.10997534e-01, -3.62176246e-01,  5.35408562e-01,
  2.64141916e-01],
[-2.94841818e-01, -3.62176246e-01, -8.98031345e-02,
  1.32509732e-01],
[ 1.03800476e+00,  9.82172869e-02,  3.64896281e-01,
  2.64141916e-01],
[-2.94841818e-01, -1.31979479e-01,  4.21733708e-01,
  3.95774101e-01],
[-5.25060772e-02, -8.22569778e-01,  1.94384000e-01,
 -2.62386821e-01],
[ 4.32165405e-01, -1.97355361e+00,  4.21733708e-01,
  3.95774101e-01],
[-2.94841818e-01, -1.28296331e+00,  8.07091462e-02,
 -1.30754636e-01],
[ 6.86617933e-02,  3.28414053e-01,  5.92245988e-01,
  7.90670654e-01],
[ 3.10997534e-01, -5.92373012e-01,  1.37546573e-01,
  1.32509732e-01],
[ 5.53333275e-01, -1.28296331e+00,  6.49083415e-01,
  3.95774101e-01],
[ 3.10997534e-01, -5.92373012e-01,  5.35408562e-01,
  8.77547895e-04],
[ 6.74501145e-01, -3.62176246e-01,  3.08058854e-01,
  1.32509732e-01],
[ 9.16836886e-01, -1.31979479e-01,  3.64896281e-01,
  2.64141916e-01],
[ 1.15917263e+00, -5.92373012e-01,  5.92245988e-01,
  2.64141916e-01],
[ 1.03800476e+00, -1.31979479e-01,  7.05920842e-01,
  6.59038469e-01],
[ 1.89829664e-01, -3.62176246e-01,  4.21733708e-01,
  3.95774101e-01],
[-1.73673948e-01, -1.05276654e+00, -1.46640561e-01,
 -2.62386821e-01],
[-4.16009689e-01, -1.51316008e+00,  2.38717193e-02,
```

```
 -1.30754636e-01],
[-4.16009689e-01, -1.51316008e+00, -3.29657076e-02,
 -2.62386821e-01],
[-5.25060772e-02, -8.22569778e-01,  8.07091462e-02,
  8.77547895e-04],
[ 1.89829664e-01, -8.22569778e-01,  7.62758269e-01,
  5.27406285e-01],
[-5.37177559e-01, -1.31979479e-01,  4.21733708e-01,
  3.95774101e-01],
[ 1.89829664e-01,  7.88807586e-01,  4.21733708e-01,
  5.27406285e-01],
[ 1.03800476e+00,  9.82172869e-02,  5.35408562e-01,
  3.95774101e-01],
[ 5.53333275e-01, -1.74335684e+00,  3.64896281e-01,
  1.32509732e-01],
[-2.94841818e-01, -1.31979479e-01,  1.94384000e-01,
  1.32509732e-01],
[-4.16009689e-01, -1.28296331e+00,  1.37546573e-01,
  1.32509732e-01],
[-4.16009689e-01, -1.05276654e+00,  3.64896281e-01,
  8.77547895e-04],
[ 3.10997534e-01, -1.31979479e-01,  4.78571135e-01,
  2.64141916e-01],
[-5.25060772e-02, -1.05276654e+00,  1.37546573e-01,
  8.77547895e-04],
[-1.02184904e+00, -1.74335684e+00, -2.60315415e-01,
 -2.62386821e-01],
[-2.94841818e-01, -8.22569778e-01,  2.51221427e-01,
  1.32509732e-01],
[-1.73673948e-01, -1.31979479e-01,  2.51221427e-01,
  8.77547895e-04],
[-1.73673948e-01, -3.62176246e-01,  2.51221427e-01,
  1.32509732e-01],
[ 4.32165405e-01, -3.62176246e-01,  3.08058854e-01,
  1.32509732e-01],
[-9.00681170e-01, -1.28296331e+00, -4.30827696e-01,
 -1.30754636e-01],
[-1.73673948e-01, -5.92373012e-01,  1.94384000e-01,
  1.32509732e-01],
[ 5.53333275e-01,  5.58610819e-01,  1.27429511e+00,
  1.71209594e+00],
[-5.25060772e-02, -8.22569778e-01,  7.62758269e-01,
  9.22302838e-01],
[ 1.52267624e+00, -1.31979479e-01,  1.21745768e+00,
  1.18556721e+00],
[ 5.53333275e-01, -3.62176246e-01,  1.04694540e+00,
  7.90670654e-01],
```

```
[ 7.95669016e-01, -1.31979479e-01,  1.16062026e+00,
  1.31719939e+00],
[ 2.12851559e+00, -1.31979479e-01,  1.61531967e+00,
  1.18556721e+00],
[-1.14301691e+00, -1.28296331e+00,  4.21733708e-01,
  6.59038469e-01],
[ 1.76501198e+00, -3.62176246e-01,  1.44480739e+00,
  7.90670654e-01],
[ 1.03800476e+00, -1.28296331e+00,  1.16062026e+00,
  7.90670654e-01],
[ 1.64384411e+00,  1.24920112e+00,  1.33113254e+00,
  1.71209594e+00],
[ 7.95669016e-01,  3.28414053e-01,  7.62758269e-01,
  1.05393502e+00],
[ 6.74501145e-01, -8.22569778e-01,  8.76433123e-01,
  9.22302838e-01],
[ 1.15917263e+00, -1.31979479e-01,  9.90107977e-01,
  1.18556721e+00],
[-1.73673948e-01, -1.28296331e+00,  7.05920842e-01,
  1.05393502e+00],
[-5.25060772e-02, -5.92373012e-01,  7.62758269e-01,
  1.58046376e+00],
[ 6.74501145e-01,  3.28414053e-01,  8.76433123e-01,
  1.44883158e+00],
[ 7.95669016e-01, -1.31979479e-01,  9.90107977e-01,
  7.90670654e-01],
[ 2.24968346e+00,  1.70959465e+00,  1.67215710e+00,
  1.31719939e+00],
[ 2.24968346e+00, -1.05276654e+00,  1.78583195e+00,
  1.44883158e+00],
[ 1.89829664e-01, -1.97355361e+00,  7.05920842e-01,
  3.95774101e-01],
[ 1.28034050e+00,  3.28414053e-01,  1.10378283e+00,
  1.44883158e+00],
[-2.94841818e-01, -5.92373012e-01,  6.49083415e-01,
  1.05393502e+00],
[ 2.24968346e+00, -5.92373012e-01,  1.67215710e+00,
  1.05393502e+00],
[ 5.53333275e-01, -8.22569778e-01,  6.49083415e-01,
  7.90670654e-01],
[ 1.03800476e+00,  5.58610819e-01,  1.10378283e+00,
  1.18556721e+00],
[ 1.64384411e+00,  3.28414053e-01,  1.27429511e+00,
  7.90670654e-01],
[ 4.32165405e-01, -5.92373012e-01,  5.92245988e-01,
  7.90670654e-01],
[ 3.10997534e-01, -1.31979479e-01,  6.49083415e-01,
```

```
  7.90670654e-01],
 [ 6.74501145e-01, -5.92373012e-01,  1.04694540e+00,
  1.18556721e+00],
 [ 1.64384411e+00, -1.31979479e-01,  1.16062026e+00,
  5.27406285e-01],
 [ 1.88617985e+00, -5.92373012e-01,  1.33113254e+00,
  9.22302838e-01],
 [ 2.49201920e+00,  1.70959465e+00,  1.50164482e+00,
  1.05393502e+00],
 [ 6.74501145e-01, -5.92373012e-01,  1.04694540e+00,
  1.31719939e+00],
 [ 5.53333275e-01, -5.92373012e-01,  7.62758269e-01,
  3.95774101e-01],
 [ 3.10997534e-01, -1.05276654e+00,  1.04694540e+00,
  2.64141916e-01],
 [ 2.24968346e+00, -1.31979479e-01,  1.33113254e+00,
  1.44883158e+00],
 [ 5.53333275e-01,  7.88807586e-01,  1.04694540e+00,
  1.58046376e+00],
 [ 6.74501145e-01,  9.82172869e-02,  9.90107977e-01,
  7.90670654e-01],
 [ 1.89829664e-01, -1.31979479e-01,  5.92245988e-01,
  7.90670654e-01],
 [ 1.28034050e+00,  9.82172869e-02,  9.33270550e-01,
  1.18556721e+00],
 [ 1.03800476e+00,  9.82172869e-02,  1.04694540e+00,
  1.58046376e+00],
 [ 1.28034050e+00,  9.82172869e-02,  7.62758269e-01,
  1.44883158e+00],
 [-5.25060772e-02, -8.22569778e-01,  7.62758269e-01,
  9.22302838e-01],
 [ 1.15917263e+00,  3.28414053e-01,  1.21745768e+00,
  1.44883158e+00],
 [ 1.03800476e+00,  5.58610819e-01,  1.10378283e+00,
  1.71209594e+00],
 [ 1.03800476e+00, -1.31979479e-01,  8.19595696e-01,
  1.44883158e+00],
 [ 5.53333275e-01, -1.28296331e+00,  7.05920842e-01,
  9.22302838e-01],
 [ 7.95669016e-01, -1.31979479e-01,  8.19595696e-01,
  1.05393502e+00],
 [ 4.32165405e-01,  7.88807586e-01,  9.33270550e-01,
  1.44883158e+00],
 [ 6.86617933e-02, -1.31979479e-01,  7.62758269e-01,
  7.90670654e-01]])
```

[50]:
```
X.shape
```

[50]: (150, 4)

2.2 2. K-Means Implementation

[52]:
```
# Randomly initialize k centroids, choosing k data points randomly from the datase

def initialize_centroids(X, k):
    np.random.seed(42)
    random_indices = np.random.choice(X.shape[0], k, replace=False)

    return X[random_indices]

k = 3
centroids = initialize_centroids(X, k)
centroids
```

[52]:
```
array([[ 3.10997534e-01, -5.92373012e-01,  5.35408562e-01,
         8.77547895e-04],
       [-1.73673948e-01,  1.70959465e+00, -1.16971425e+00,
        -1.18381211e+00],
       [ 2.24968346e+00, -1.05276654e+00,  1.78583195e+00,
         1.44883158e+00]])
```

[53]:
```
# Assign each data point to the nearest cluster centroid by calculating the Euclidean distance.

def euclidean_distance(a, b):
    return np.sqrt(np.sum((a - b) ** 2, axis=1))

def assign_clusters(X, centroids):
    clusters = []
    for point in X:
        distances = euclidean_distance(point,centroids)
        # print(distances)
        cluster = np.argmin(distances)
        # print(clusters)
        clusters.append(cluster)
    return np.array(clusters)

clusters = assign_clusters(X, centroids)
clusters
```

[53]:
```
array([1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
       1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 0, 1, 1,
       1, 1, 1, 1, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
       0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
```

```
       0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 2, 0, 0, 2, 0, 2, 0, 2,
       0, 0, 0, 0, 0, 0, 0, 2, 2, 0, 2, 0, 2, 0, 0, 2, 0, 0, 0, 2, 2, 2,
       0, 0, 0, 2, 0, 0, 0, 0, 2, 2, 0, 2, 2, 0, 0, 0, 0, 0])
```

[54]:
```
# Update the cluster centroids to the mean of all points assigned to each␣
 ↪cluster

def update_centroids(X, clusters, k):
    new_centroids = np.zeros((k, X.shape[1]))
    for i in range(k):
        cluster_points = X[clusters == i]
        if len(cluster_points) > 0:
            new_centroids[i] = np.mean(cluster_points, axis=0)
    return new_centroids

new_centroids = update_centroids(X, clusters, k)
new_centroids
```

[54]:
```
array([[ 0.2331039 , -0.56770907,  0.49142722,  0.48196184],
       [-1.00206653,  0.90625492, -1.30310821, -1.25634413],
       [ 1.73650189,  0.19300419,  1.32778916,  1.23976869]])
```

[55]:
```
# Repeat these steps until: The cluster assignments do not change. A maximum␣
 ↪number of iterations (e.g., 100) is reached
max_iters = 1000
for i in range(max_iters):
    old_centroids = centroids.copy()
    clusters = assign_clusters(X, centroids)
    centroids = update_centroids(X, clusters, k)

    if np.linalg.norm(centroids - old_centroids) < 0.001:
        print("Converged at iteration: ", i)
        break

print("New Centroids after convergence \n")
print(new_centroids)
print("Clusters \n")
print(clusters)
```

```
Converged at iteration:  6
New Centroids after convergence

[[ 0.2331039  -0.56770907  0.49142722  0.48196184]
 [-1.00206653  0.90625492 -1.30310821 -1.25634413]
 [ 1.73650189  0.19300419  1.32778916  1.23976869]]
Clusters
```

```
[1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
 1 1 1 1 1 1 1 1 1 1 1 1 2 2 2 0 0 0 2 0 0 0 0 0 0 0 2 0 0 0 0 2 0 0 0
 0 2 2 2 0 0 0 0 0 0 0 2 2 0 0 0 0 0 0 0 0 0 0 0 0 0 2 0 2 2 2 2 0 2 0 2 2
 0 2 0 0 2 2 2 2 0 2 0 2 0 2 2 0 0 2 2 2 2 2 0 0 2 2 2 0 2 2 2 0 2 2 2 0 2
 2 0]
```

[56]:
```
# Implement the algorithm in Python, structuring it with reusable helper functions:
# A function to calculate distances between points and centroids.
# A function to assign clusters
# A function to update centroids

def kmeans(X, k, max_iters=100):
    centroids = initialize_centroids(X, k)
    for i in range(max_iters):
        old_centroids = centroids.copy()
        clusters = assign_clusters(X, centroids)
        centroids = update_centroids(X, clusters, k)
        if np.linalg.norm(centroids - old_centroids) < 0.001:
            print("Converged at iteration: ", i)
            break
    return clusters, centroids

k = 3
clusters, new_centroids = kmeans(X, k)
print("New Centroids after convergence \n")
print(new_centroids)
print("Clusters \n")
print(clusters)
```

```
Converged at iteration:  6
New Centroids after convergence

[[-0.01139555 -0.87600831  0.37707573  0.31115341]
 [-1.01457897  0.85326268 -1.30498732 -1.25489349]
 [ 1.16743407  0.14530299  1.00302557  1.0300019 ]]
Clusters

[1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
 1 1 1 1 1 1 1 1 1 1 1 1 2 2 2 0 0 0 2 0 0 0 0 0 0 0 2 0 0 0 0 2 0 0 0
 0 2 2 2 0 0 0 0 0 0 0 2 2 0 0 0 0 0 0 0 0 0 0 0 0 0 2 0 2 2 2 2 0 2 0 2 2
 0 2 0 0 2 2 2 2 0 2 0 2 0 2 2 0 0 2 2 2 2 2 0 0 2 2 2 0 2 2 2 0 2 2 2 0 2
 2 0]
```

2.3 3. Evaluation

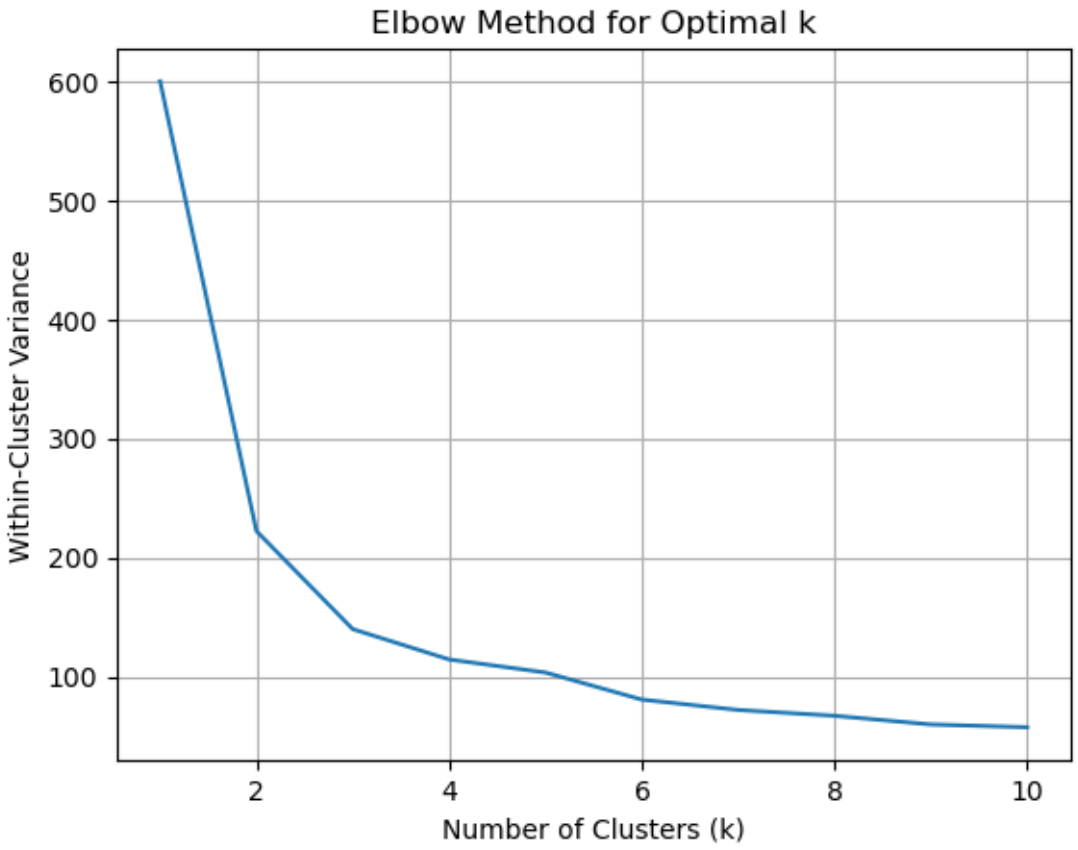
[58]:
```
# Use the Elbow Method:
# Apply your K-Means implementation for k ranging from 1 to 10.
# Calculate the total within-cluster variance for each value of k.

def compute_wcv(X, clusters, centroids):
    wcv = 0
    for i in range(len(X)):
        centroid = centroids[clusters[i]]
        wcv += np.sum((X[i] - centroid) ** 2)
    return wcv

wcv = compute_wcv(X, clusters, centroids)
print("Within Cluster Variance: ", wcv)
```

```
Within Cluster Variance:  140.03275277428648
```

[59]:
```
# Plot the total within-cluster variance values to identify the optimal number␣
 ↪of clusters (elbow point)

k_values = range(1, 11)
wcv_values = []

for k in k_values:
    clusters, centroids = kmeans(X, k)
    wcv = compute_wcv(X, clusters, centroids)
    wcv_values.append(wcv)
```

```
Converged at iteration:  1
Converged at iteration:  2
Converged at iteration:  6
Converged at iteration:  9
Converged at iteration:  6
Converged at iteration:  8
Converged at iteration:  6
Converged at iteration:  7
Converged at iteration:  7
Converged at iteration:  8
```

[60]:
```
plt.plot(k_values, wcv_values)
plt.xlabel("Number of Clusters (k)")
plt.ylabel("Within-Cluster Variance")
plt.title("Elbow Method for Optimal k")
plt.grid()
plt.show()
```



[61]:
```
# Optimal number of clusters - 3
```

2.4 4. Cluster Visualization

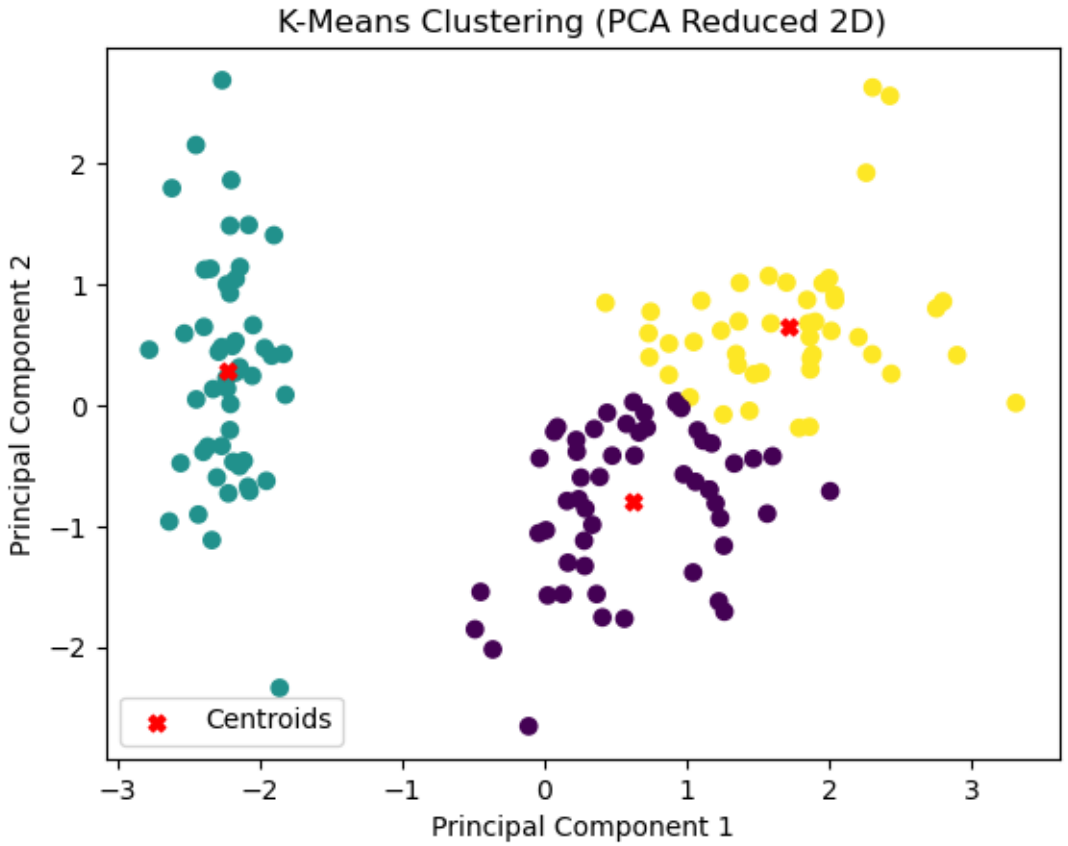
[63]:
```
# Reduce the numeric data to 2D or 3D using PCA (Principal Component Analysis)

pca = PCA(n_components=2)
X_pca = pca.fit_transform(X)
```

[64]:
```
# Visualize the clusters and centroids using a scatter plot

optimal_k = 3
clusters_pca, centroids_pca = kmeans(X_pca, optimal_k)

print("Centroids after PCA \n")
print(centroids_pca)
print("Clusters \n")
print(clusters_pca)
```

```
Converged at iteration:  8
```

```
Centroids after PCA

[[ 0.61742366 -0.79845836]
 [-2.22475316  0.28892745]
 [ 1.71731904  0.65486305]]
Clusters

[1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
 1 1 1 1 1 1 1 1 1 1 1 1 2 2 2 0 0 0 2 0 0 0 0 0 0 0 2 0 0 0 0 2 0 0 0
 0 2 2 2 0 0 0 0 0 0 0 2 2 0 0 0 0 0 0 0 0 0 0 0 0 0 2 0 2 2 2 2 0 2 0 2 2
 0 2 0 0 2 2 2 2 0 2 0 2 0 2 2 0 2 2 2 2 2 2 0 0 2 2 2 0 2 2 2 0 2 2 2 0 2
 2 0]
```

[65]:
```
plt.scatter(X_pca[:, 0], X_pca[:, 1], c=clusters_pca)
plt.scatter(centroids_pca[:, 0], centroids_pca[:, 1], c='red', marker='X',␣
 ↪label="Centroids")
plt.xlabel("Principal Component 1")
plt.ylabel("Principal Component 2")
plt.title("K-Means Clustering (PCA Reduced 2D)")
plt.legend()
plt.show()
```



2.5 5. Analysis and Implementation Questions

[67]:
```
# What are the final centroids of the clusters for the chosen value of k?

print("Final centroids:\n", new_centroids)
```

```
Final centroids:
 [[-0.01139555 -0.87600831  0.37707573  0.31115341]
 [-1.01457897  0.85326268 -1.30498732 -1.25489349]
 [ 1.16743407  0.14530299  1.00302557  1.0300019 ]]
```

[68]:
```
# How does the within-cluster variance change as k increases?

# As k increases, within-cluster variance decreases.
# More clusters mean each cluster is smaller, reducing variance.
# However, after a certain point (elbow), the decrease slows down, making extra␣
 ↪clusters unnecessary.
```

[69]:
```
# Does your algorithm perform well for the Iris dataset? Why or why not?

# Yes, it performs well!
# The Elbow Method suggests k = 3, which matches the 3 known species in the␣
 ↪Iris dataset.
# The clusters found by K-Means closely align with real species labels.
# PCA visualization shows well-separated clusters.

# Limitations:
# Different initial centroids can lead to different results (sensitivity to␣
 ↪initialization)
```