a) Apply Decision Tree, Random Forest, and KNN

```
# Import necessary libraries
import pandas as pd
import numpy as np
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import LabelEncoder, StandardScaler
from sklearn.tree import DecisionTreeClassifier
from sklearn.ensemble import RandomForestClassifier
from sklearn.neighbors import KNeighborsClassifier
from sklearn.metrics import confusion_matrix, accuracy_score,
f1_score, precision_score, recall_score, roc_curve, auc
from imblearn.over_sampling import SMOTE
import matplotlib.pyplot as plt
import seaborn as sns

# Load dataset
data = pd.read_csv(r"C:\Users\bolla\Downloads\msba sem1\scm\Prostate
Cancer.csv")

data.head()

   Gender Smoking  Prostate Volume  PSA Level  Age Prostate Cancer
0    Male      No            50.89       9.64   57              No
1  Female      No            27.95       2.89   41              No
2    Male     Yes            20.22       5.22   74              No
3    Male      No            52.62       3.36   55              No
4    Male     Yes            48.27       3.21   80              No

data.describe()

       Prostate Volume  PSA Level        Age
count        70.000000  70.000000  70.000000
mean         39.675571   4.975714  66.557143
std          11.716884   2.844250  13.562134
min          20.220000   0.550000  41.000000
25%          29.355000   2.677500  57.250000
50%          40.045000   5.250000  69.000000
75%          50.737500   7.030000  74.750000
max          58.870000   9.860000  88.000000

data.info()

<class 'pandas.core.frame.DataFrame'>
RangeIndex: 70 entries, 0 to 69
Data columns (total 6 columns):
 #   Column           Non-Null Count  Dtype
```

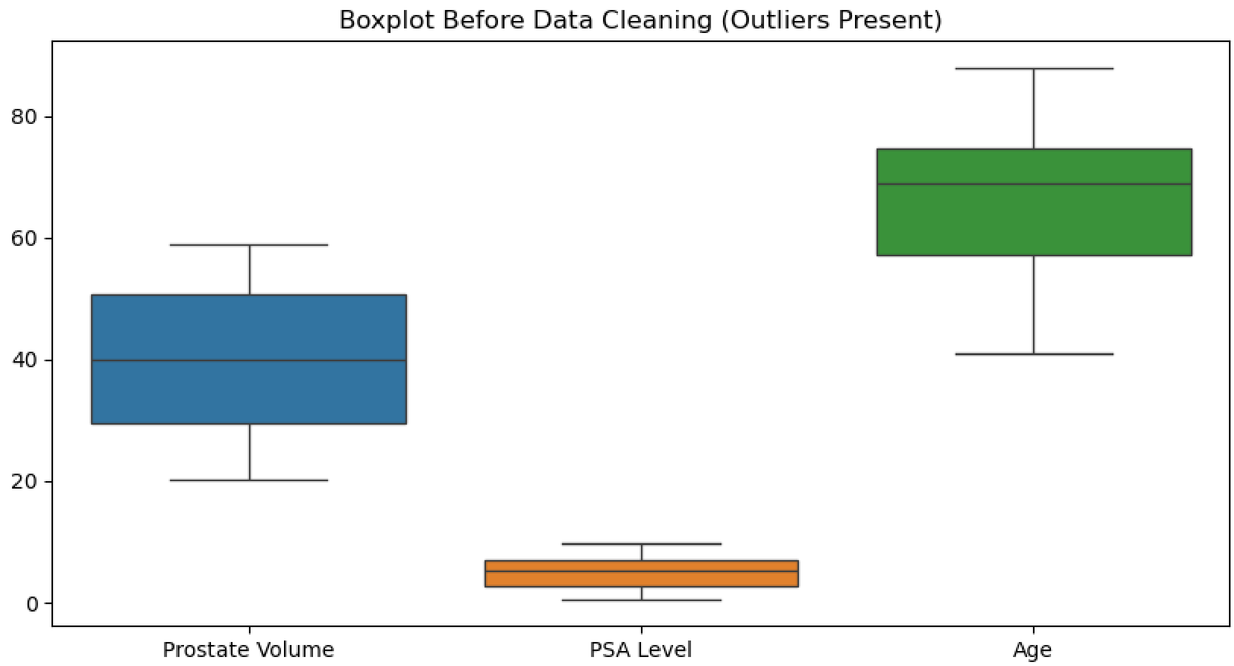
```
---  ------           --------------  -----
 0   Gender           70 non-null     object
 1   Smoking          70 non-null     object
 2   Prostate Volume  70 non-null     float64
 3   PSA Level        70 non-null     float64
 4   Age              70 non-null     int64
 5   Prostate Cancer  70 non-null     object
dtypes: float64(2), int64(1), object(3)
memory usage: 3.4+ KB
```

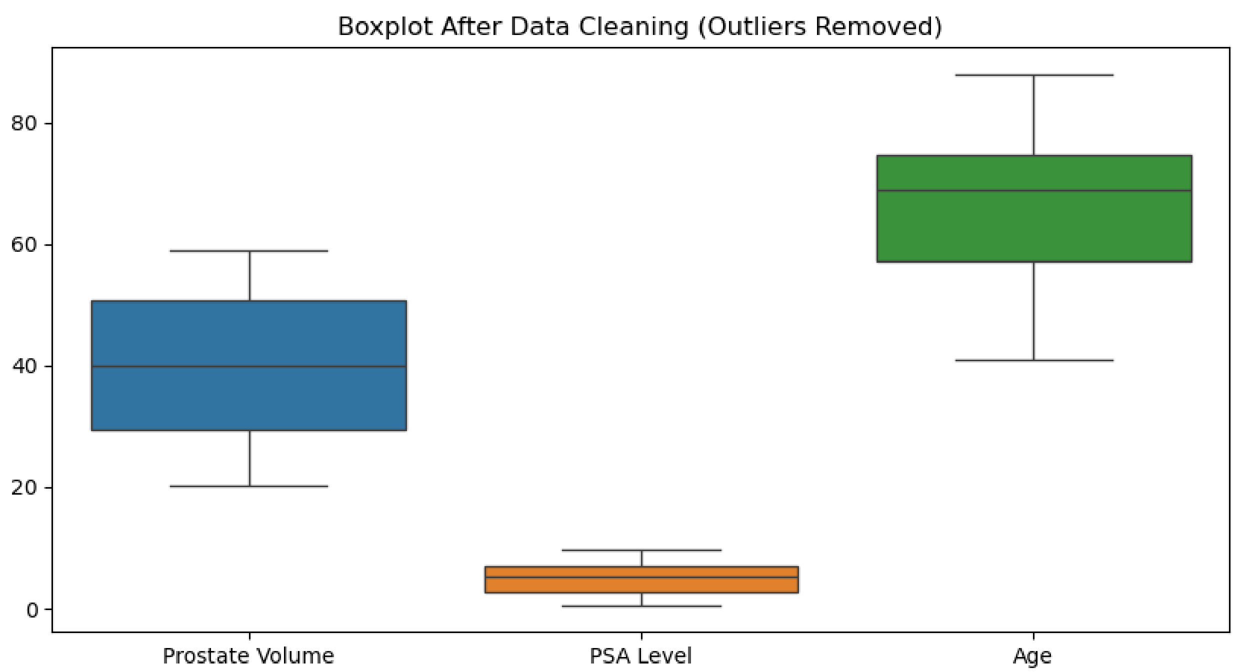
```
data.isnull().sum()# there is no missing values in the data set
```

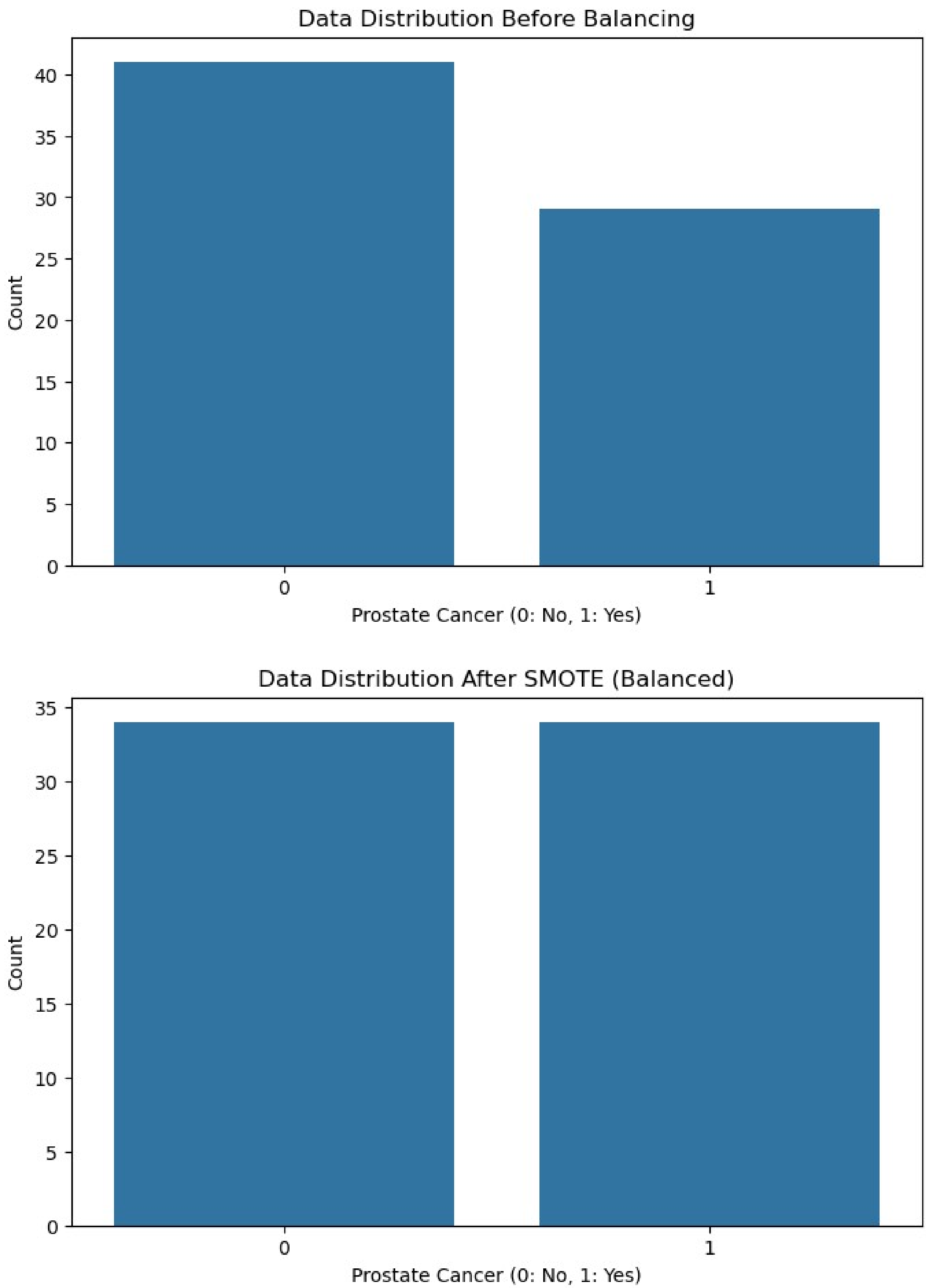
```
Gender             0
Smoking            0
Prostate Volume    0
PSA Level          0
Age                0
Prostate Cancer    0
dtype: int64
```

```
def remove_outliers(df):
    columns_to_check = ['Prostate Volume', 'PSA Level', 'Age']
    for column in columns_to_check:
        Q1 = df[column].quantile(0.25)
        Q3 = df[column].quantile(0.75)
        IQR = Q3 - Q1
        df = df[(df[column] >= (Q1 - 1.5 * IQR)) & (df[column] <= (Q3
+ 1.5 * IQR))]
    return df

# Visualize the data before cleaning
plt.figure(figsize=(10, 5))
sns.boxplot(data=data[['Prostate Volume', 'PSA Level', 'Age']])
plt.title('Boxplot Before Data Cleaning (Outliers Present)')
plt.show()

data_cleaned = remove_outliers(data)

# Visualize the data after cleaning
plt.figure(figsize=(10, 5))
sns.boxplot(data=data_cleaned[['Prostate Volume', 'PSA Level',
'Age']])
plt.title('Boxplot After Data Cleaning (Outliers Removed)')
plt.show()

# Encode categorical variables
encoder = LabelEncoder()
data_cleaned['Gender'] = encoder.fit_transform(data_cleaned['Gender'])

data_cleaned['Smoking'] =
```

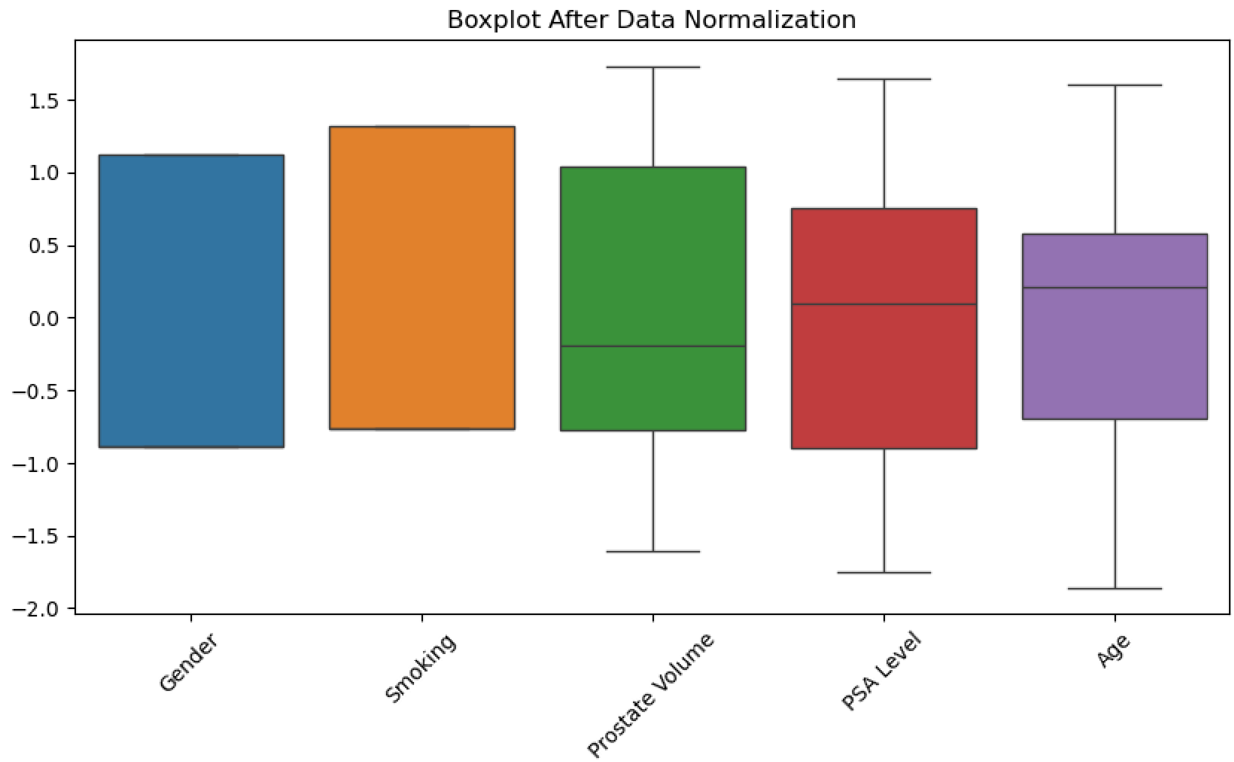
```
encoder.fit_transform(data_cleaned['Smoking'])
data_cleaned['Prostate Cancer'] =
encoder.fit_transform(data_cleaned['Prostate Cancer'])

# Split the dataset into features and target variable
X = data_cleaned.drop('Prostate Cancer', axis=1)
y = data_cleaned['Prostate Cancer']

# Visualize data distribution before balancing
plt.figure(figsize=(8, 5))
sns.countplot(x=y)
plt.title('Data Distribution Before Balancing')
plt.xlabel('Prostate Cancer (0: No, 1: Yes)')
plt.ylabel('Count')
plt.show()

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

# Apply SMOTE to the training data
smote = SMOTE(random_state=42)
X_train_balanced, y_train_balanced = smote.fit_resample(X_train,
y_train)

# Visualize the data after SMOTE
plt.figure(figsize=(8, 5))
sns.countplot(x=y_train_balanced)
plt.title('Data Distribution After SMOTE (Balanced)')
plt.xlabel('Prostate Cancer (0: No, 1: Yes)')
plt.ylabel('Count')
plt.show()

# Normalize the data for KNN
scaler = StandardScaler()
X_train_balanced = scaler.fit_transform(X_train_balanced)
X_test = scaler.transform(X_test)

# Visualize the data after scaling
plt.figure(figsize=(10, 5))
sns.boxplot(data=X_train_balanced)
plt.title('Boxplot After Data Normalization')
plt.xticks(ticks=np.arange(len(X.columns)), labels=X.columns,
rotation=45)
plt.show()
```


Boxplot Before Data Cleaning (Outliers Present)
80
60
40
20
0
Prostate Volume
PSA Level
Age


Boxplot After Data Cleaning (Outliers Removed)
80
60
40
20
0
Prostate Volume
PSA Level
Age



Data Distribution Before Balancing
Count
40
35
30
25
20
15
10
5
0
0
1
Prostate Cancer (0: No, 1: Yes)
Data Distribution After SMOTE (Balanced)
Count
35
30
25
20
15
10
5
0
0
1
Prostate Cancer (0: No, 1: Yes)



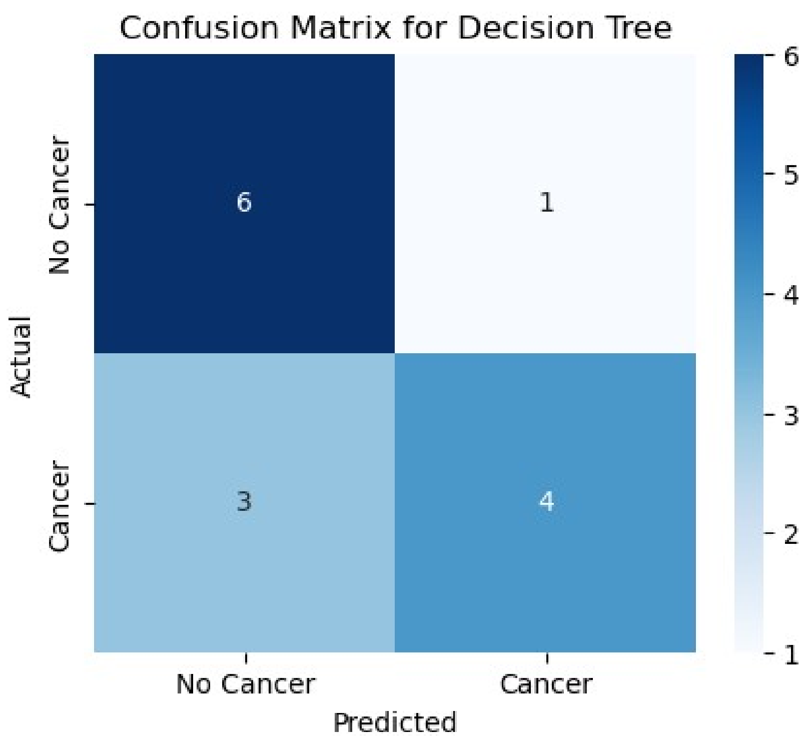
b) Show the confusion matrix, accuracy, F1 score, precision, and recall for training and testing datasets

```
# Initialize models
decision_tree = DecisionTreeClassifier(random_state=42)
random_forest = RandomForestClassifier(n_estimators=100,
criterion='gini', random_state=42)
knn = KNeighborsClassifier()

# Train models on balanced data
decision_tree.fit(X_train_balanced, y_train_balanced)
random_forest.fit(X_train_balanced, y_train_balanced)
knn.fit(X_train_balanced, y_train_balanced)

# Define a function to calculate and display model metrics
def display_metrics(model, X_train, y_train, X_test, y_test,
model_name):
    # Predictions
    y_train_pred = model.predict(X_train)
    y_test_pred = model.predict(X_test)

    # Metrics
```

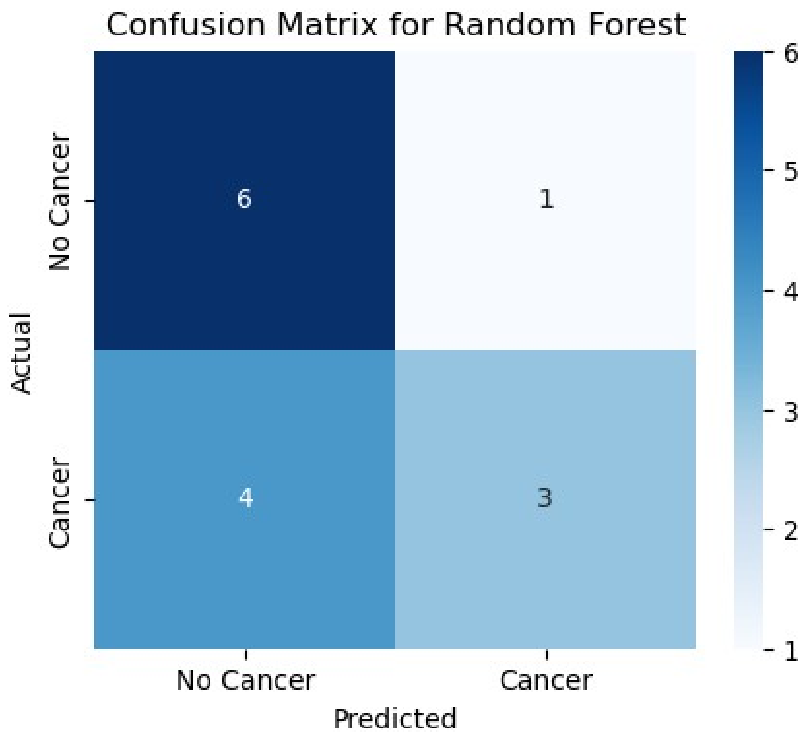
```
    metrics = {
        'Accuracy': [accuracy_score(y_train, y_train_pred),
accuracy_score(y_test, y_test_pred)],
        'F1 Score': [f1_score(y_train, y_train_pred), f1_score(y_test,
y_test_pred)],
        'Precision': [precision_score(y_train, y_train_pred),
precision_score(y_test, y_test_pred)],
        'Recall': [recall_score(y_train, y_train_pred),
recall_score(y_test, y_test_pred)],
    }

    print(f'\n{model_name} Performance:')
    for metric, values in metrics.items():
        print(f'{metric}: Train = {values[0]:.2f}, Test =
{values[1]:.2f}')

    # Confusion Matrix
    cm = confusion_matrix(y_test, y_test_pred)
    plt.figure(figsize=(5, 4))
    sns.heatmap(cm, annot=True, fmt='d', cmap='Blues',
                xticklabels=['No Cancer', 'Cancer'],
                yticklabels=['No Cancer', 'Cancer'])
    plt.title(f'Confusion Matrix for {model_name}')
    plt.xlabel('Predicted')
    plt.ylabel('Actual')
    plt.show()

# Display metrics for all models
display_metrics(decision_tree, X_train_balanced, y_train_balanced,
X_test, y_test, 'Decision Tree')
display_metrics(random_forest, X_train_balanced, y_train_balanced,
X_test, y_test, 'Random Forest')
display_metrics(knn, X_train_balanced, y_train_balanced, X_test,
y_test, 'KNN')

# Finding the optimal k value for KNN
k_values = range(1, 21)
accuracy_scores = []

for k in k_values:
    knn = KNeighborsClassifier(n_neighbors=k)
    knn.fit(X_train_balanced, y_train_balanced)
    score = knn.score(X_test, y_test)
    accuracy_scores.append(score)

# Plotting accuracy for different k values
plt.figure(figsize=(10, 5))
plt.plot(k_values, accuracy_scores, marker='o')
plt.title('KNN Accuracy vs. K Values')
plt.xlabel('K Value')
```

```
plt.ylabel('Accuracy')
plt.xticks(k_values)
plt.grid()
plt.show()

# Identify the optimal k
optimal_k = k_values[np.argmax(accuracy_scores)]
print(f'The optimal k value for KNN is: {optimal_k}')

# Re-train the KNN model using the optimal k value
knn_optimal = KNeighborsClassifier(n_neighbors=optimal_k)
knn_optimal.fit(X_train_balanced, y_train_balanced)

# Evaluate the model performance again using the testing dataset for
KNN
display_metrics(knn_optimal, X_train_balanced, y_train_balanced,
X_test, y_test, 'KNN (Optimal)')
```

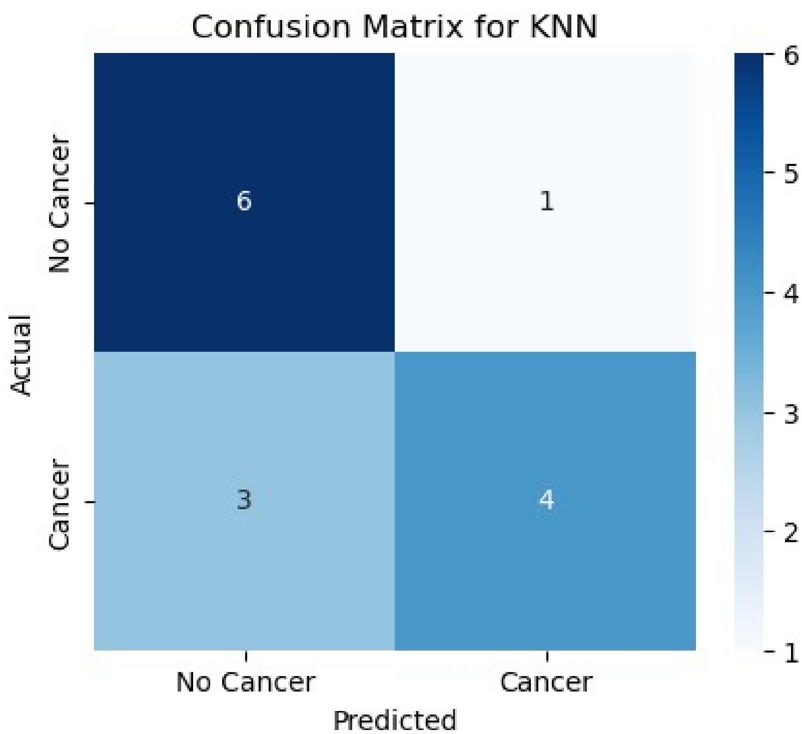
```
Decision Tree Performance:
Accuracy: Train = 1.00, Test = 0.71
F1 Score: Train = 1.00, Test = 0.67
Precision: Train = 1.00, Test = 0.80
Recall: Train = 1.00, Test = 0.57
```

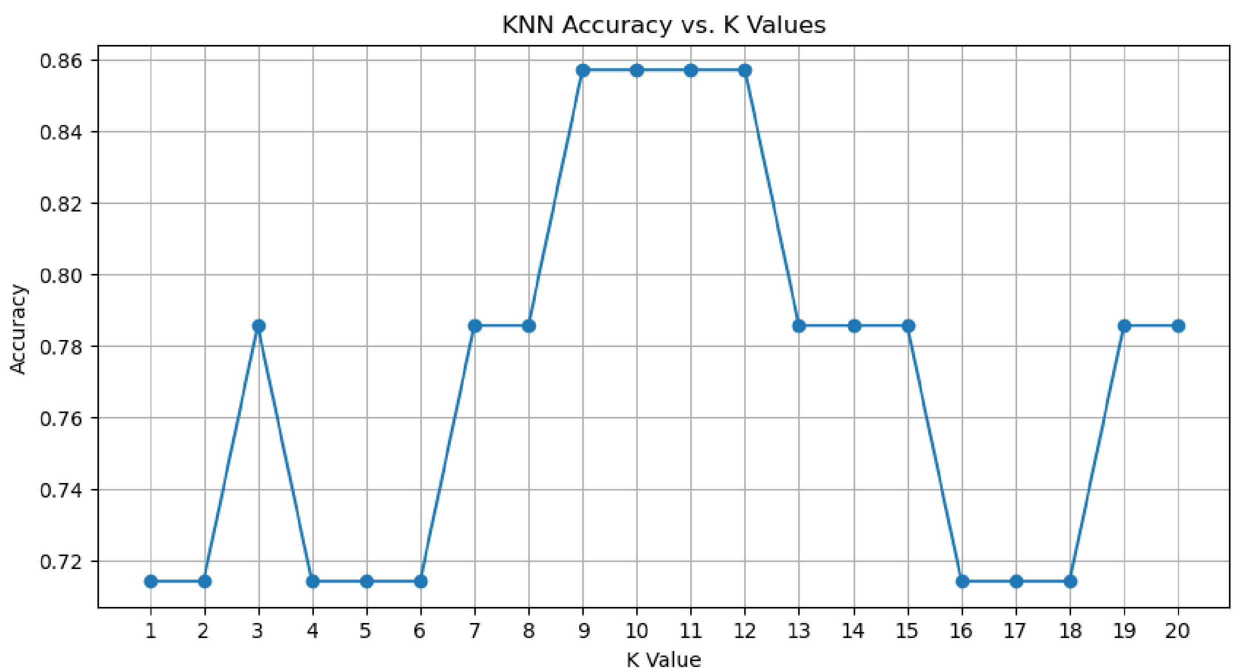


```
Random Forest Performance:
Accuracy: Train = 1.00, Test = 0.64
F1 Score: Train = 1.00, Test = 0.55
Precision: Train = 1.00, Test = 0.75
Recall: Train = 1.00, Test = 0.43
```



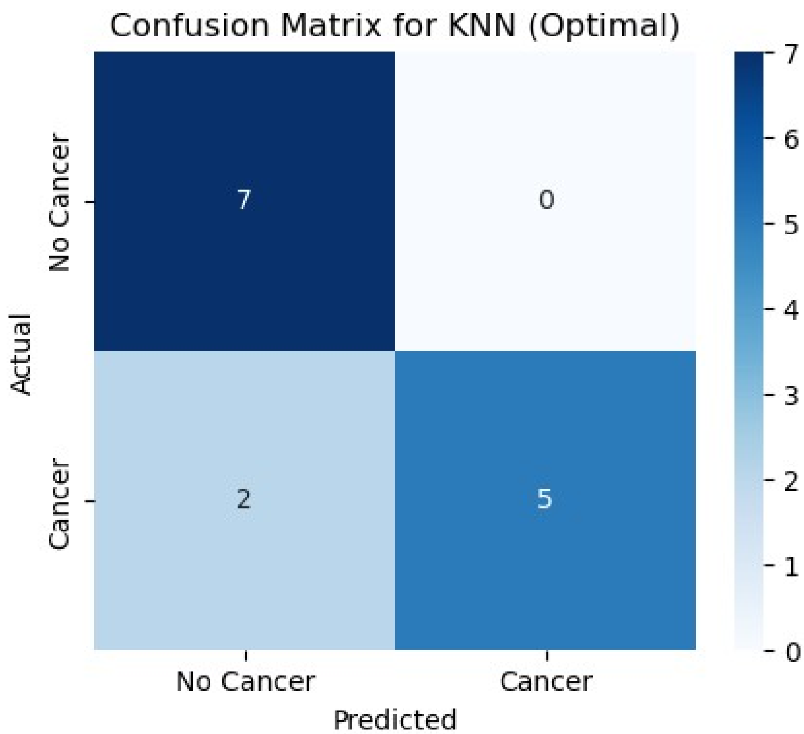

```
KNN Performance:
Accuracy: Train = 0.76, Test = 0.71
F1 Score: Train = 0.77, Test = 0.67
Precision: Train = 0.75, Test = 0.80
Recall: Train = 0.79, Test = 0.57
```






```
The optimal k value for KNN is: 9

KNN (Optimal) Performance:
Accuracy: Train = 0.71, Test = 0.86
F1 Score: Train = 0.72, Test = 0.83
```

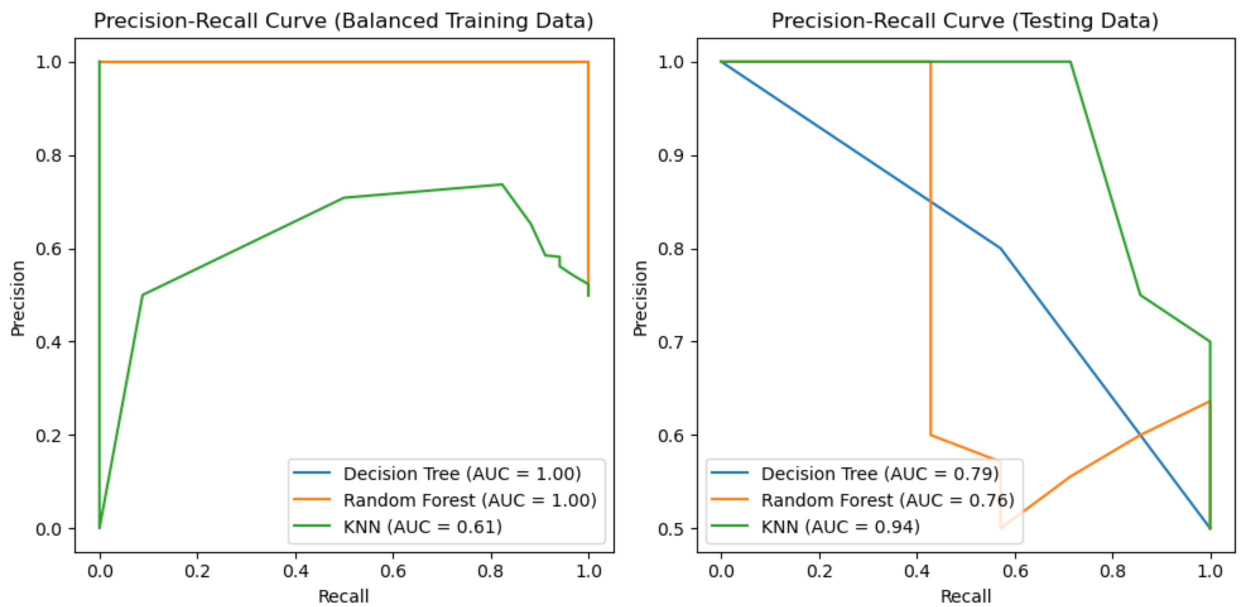
```
Precision: Train = 0.68, Test = 1.00
Recall: Train = 0.76, Test = 0.71
```



c) Generate ROC plots for the training and testing datasets

```
# Import necessary libraries
import matplotlib.pyplot as plt
from sklearn.metrics import precision_recall_curve, auc

# Assuming the models (decision_tree, random_forest, knn) and datasets
(X_train_balanced, y_train_balanced, X_test, y_test) are already
defined

plt.figure(figsize=(10, 5))

# Plot PR curve for training data
plt.subplot(1, 2, 1)
plt.title('Precision-Recall Curve (Balanced Training Data)')
precision_dt, recall_dt, _ = precision_recall_curve(y_train_balanced,
decision_tree.predict_proba(X_train_balanced)[:, 1])
precision_rf, recall_rf, _ = precision_recall_curve(y_train_balanced,
random_forest.predict_proba(X_train_balanced)[:, 1])
precision_knn, recall_knn, _ =
```

```
precision_recall_curve(y_train_balanced,
knn.predict_proba(X_train_balanced)[:, 1])

plt.plot(recall_dt, precision_dt, label='Decision Tree (AUC =
{:.2f})'.format(auc(recall_dt, precision_dt)))
plt.plot(recall_rf, precision_rf, label='Random Forest (AUC =
{:.2f})'.format(auc(recall_rf, precision_rf)))
plt.plot(recall_knn, precision_knn, label='KNN (AUC =
{:.2f})'.format(auc(recall_knn, precision_knn)))
plt.xlabel('Recall')
plt.ylabel('Precision')
plt.legend()

# Plot PR curve for testing data
plt.subplot(1, 2, 2)
plt.title('Precision-Recall Curve (Testing Data)')
precision_dt_test, recall_dt_test, _ = precision_recall_curve(y_test,
decision_tree.predict_proba(X_test)[:, 1])
precision_rf_test, recall_rf_test, _ = precision_recall_curve(y_test,
random_forest.predict_proba(X_test)[:, 1])
precision_knn_test, recall_knn_test, _ =
precision_recall_curve(y_test, knn_optimal.predict_proba(X_test)[:,
1])

plt.plot(recall_dt_test, precision_dt_test, label='Decision Tree (AUC
= {:.2f})'.format(auc(recall_dt_test, precision_dt_test)))
plt.plot(recall_rf_test, precision_rf_test, label='Random Forest (AUC
= {:.2f})'.format(auc(recall_rf_test, precision_rf_test)))
plt.plot(recall_knn_test, precision_knn_test, label='KNN (AUC =
{:.2f})'.format(auc(recall_knn_test, precision_knn_test)))
plt.xlabel('Recall')
plt.ylabel('Precision')
plt.legend()

plt.tight_layout()
plt.show()
```



Interpretation:

Training Data:

Decision Tree: AUC = 1.00 (perfect classification, potential overfitting)

Random Forest: AUC = 1.00 (perfect classification, potential overfitting)

KNN: AUC = 0.61 (good performance, less prone to overfitting)

Testing Data:

Decision Tree: AUC = 0.79 (significant drop in performance, suggests overfitting)

Random Forest: AUC = 0.76 (similar drop, indicates overfitting)

KNN: AUC = 0.94 (best performance, good generalization to unseen data)

The KNN model outperforms the others on the test data, indicating it generalizes better than the tree-based models, which exhibit signs of overfitting. Model overfitting occurs when the model captures noise in the training data, leading to poor performance on unseen data. This is likely due to the small dataset size, which emphasizes the need to increase the sample size for better representation. To mitigate overfitting, strategies such as increasing the dataset, applying cross-validation, adjusting model complexity, and using feature selection can be beneficial.