```
In [48]: import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns
from sklearn.linear_model import LogisticRegression
from sklearn.tree import DecisionTreeClassifier
from sklearn.ensemble import RandomForestClassifier
from sklearn.svm import SVC
from sklearn.neighbors import KNeighborsClassifier
from sklearn.model_selection import train_test_split , cross_val_score ,Random
from sklearn.metrics import classification_report , confusion_matrix ,accuracy
from sklearn.naive_bayes import GaussianNB
from sklearn.preprocessing import StandardScaler
from keras.callbacks import EarlyStopping
from keras.models import Sequential
from keras.layers import Dense
import warnings
warnings.filterwarnings('ignore')
```

Step 1 - Data Importing

```
In [49]: df = pd.read_csv(r"/Users/pvsairamsaketh/Desktop/projects/heart.csv")
df.head()
```

Out[49]:

	age	sex	cp	trestbps	chol	fbs	restecg	thalach	exang	oldpeak	slope	c
0	52	1	0	125	212	0	1	168	0	1.0	2	
1	53	1	0	140	203	1	0	155	1	3.1	0	
2	70	1	0	145	174	0	1	125	1	2.6	0	
3	61	1	0	148	203	0	1	161	0	0.0	2	
4	62	0	0	138	294	1	1	106	0	1.9	1	

```
In [50]: #checking our dependant column
df['target'].value_counts()
```

```
Out[50]: target
1    526
0    499
Name: count, dtype: int64
```

step2 - basic data inspection

```
In [51]: df.head() #to print topp 5 rows
```

Out[51]:

	age	sex	cp	trestbps	chol	fbs	restecg	thalach	exang	oldpeak	slope	c
0	52	1	0	125	212	0	1	168	0	1.0	2	
1	53	1	0	140	203	1	0	155	1	3.1	0	
2	70	1	0	145	174	0	1	125	1	2.6	0	
3	61	1	0	148	203	0	1	161	0	0.0	2	
4	62	0	0	138	294	1	1	106	0	1.9	1	

In [52]:

```python
df.tail()#to print bottom 5 rows
```

Out[52]:

	age	sex	cp	trestbps	chol	fbs	restecg	thalach	exang	oldpeak	slop
1020	59	1	1	140	221	0	1	164	1	0.0	[illegible]
1021	60	1	0	125	258	0	0	141	1	2.8	[illegible]
1022	47	1	0	110	275	0	0	118	1	1.0	[illegible]
1023	50	0	0	110	254	0	0	159	0	0.0	[illegible]
1024	54	1	0	120	188	0	1	113	0	1.4	[illegible]

In [53]:

```python
df.sample(5) #to print random 5 sample
```

Out[53]:

	age	sex	cp	trestbps	chol	fbs	restecg	thalach	exang	oldpeak	slope
222	64	1	3	110	211	0	0	144	1	1.8	1
816	70	1	1	156	245	0	0	143	0	0.0	2
649	45	0	1	130	234	0	0	175	0	0.6	1
933	38	1	3	120	231	0	1	182	1	3.8	1
178	44	1	0	110	197	0	0	177	0	0.0	2

In [54]:

```python
#to check datatypes
df.dtypes
```

Out[54]:
```
age           int64
sex           int64
cp            int64
trestbps      int64
chol          int64
fbs           int64
restecg       int64
thalach       int64
exang         int64
oldpeak     float64
slope         int64
ca            int64
thal          int64
target        int64
dtype: object
```

In [55]:
```
#to check information
df.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 1025 entries, 0 to 1024
Data columns (total 14 columns):
 #   Column    Non-Null Count  Dtype  
---  ------    --------------  -----  
 0   age       1025 non-null   int64  
 1   sex       1025 non-null   int64  
 2   cp        1025 non-null   int64  
 3   trestbps  1025 non-null   int64  
 4   chol      1025 non-null   int64  
 5   fbs       1025 non-null   int64  
 6   restecg   1025 non-null   int64  
 7   thalach   1025 non-null   int64  
 8   exang     1025 non-null   int64  
 9   oldpeak   1025 non-null   float64
 10  slope     1025 non-null   int64  
 11  ca        1025 non-null   int64  
 12  thal      1025 non-null   int64  
 13  target    1025 non-null   int64  
dtypes: float64(1), int64(13)
memory usage: 112.2 KB
```

In [56]:
```
#to check mathematical information of dataset
df.describe().T
```

Out[56]:

	count	mean	std	min	25%	50%	75%	max
age	1025.0	54.434146	9.072290	29.0	48.0	56.0	61.0	77.0
sex	1025.0	0.695610	0.460373	0.0	0.0	1.0	1.0	1.0
cp	1025.0	0.942439	1.029641	0.0	0.0	1.0	2.0	3.0
trestbps	1025.0	131.611707	17.516718	94.0	120.0	130.0	140.0	200.0
chol	1025.0	246.000000	51.592510	126.0	211.0	240.0	275.0	564.0
fbs	1025.0	0.149268	0.356527	0.0	0.0	0.0	0.0	1.0
restecg	1025.0	0.529756	0.527878	0.0	0.0	1.0	1.0	2.0
thalach	1025.0	149.114146	23.005724	71.0	132.0	152.0	166.0	202.0
exang	1025.0	0.336585	0.472772	0.0	0.0	0.0	1.0	1.0
oldpeak	1025.0	1.071512	1.175053	0.0	0.0	0.8	1.8	6.2
slope	1025.0	1.385366	0.617755	0.0	1.0	1.0	2.0	2.0
ca	1025.0	0.754146	1.030798	0.0	0.0	0.0	1.0	4.0
thal	1025.0	2.323902	0.620660	0.0	2.0	2.0	3.0	3.0
target	1025.0	0.513171	0.500070	0.0	0.0	1.0	1.0	1.0

In [57]:

```
df.shape
```

Out[57]:

```
(1025, 14)
```

Step 3 - Data Cleaning

In [58]:

```
#to check null values or not
df.isnull().sum() #no null values
```

Out[58]:

```
age         0
sex         0
cp          0
trestbps    0
chol        0
fbs         0
restecg     0
thalach     0
exang       0
oldpeak     0
slope       0
ca          0
thal        0
target      0
dtype: int64
```

In [59]:
```
#Rename column names
df = df.rename(columns={'cp':'ChestPainType','trestbps':'RestingBloodPressure'

df.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 1025 entries, 0 to 1024
Data columns (total 14 columns):
 #   Column                            Non-Null Count  Dtype
---  ------                            --------------  -----
 0   age                               1025 non-null   int64
 1   sex                               1025 non-null   int64
 2   ChestPainType                     1025 non-null   int64
 3   RestingBloodPressure              1025 non-null   int64
 4   Cholestrol                        1025 non-null   int64
 5   FastingBloodSugar                 1025 non-null   int64
 6   RestingEcg                        1025 non-null   int64
 7   MaxHeartRate                      1025 non-null   int64
 8   Exercise                          1025 non-null   int64
 9   oldpeak                           1025 non-null   float64
 10  slope                             1025 non-null   int64
 11  NoOfBloodVesselsDuringFluroscopy  1025 non-null   int64
 12  Thalassemia                       1025 non-null   int64
 13  target                            1025 non-null   int64
dtypes: float64(1), int64(13)
memory usage: 112.2 KB
```

Step 4 - Data Analysis and Data Visualization

In [60]:
```
#grouping chestpaintype , target
df1 = df.groupby('ChestPainType').agg({'target':'count'})
```

In [61]:
```
df1 #from this we can say out of 1025 records we have , 497 people got chestpa
```

Out[61]:

ChestPainType	target
0	497
1	167
2	284
3	77

In [62]:
```
#grouping Thalassemia with Target
# Grouping Thalassemia with Target and adding percentage
df2 = (
    df.groupby(['Thalassemia', 'ChestPainType', 'NoOfBloodVesselsDuringFluroso
      .agg(Count_of_Patients=('target', 'count'))
```

```
        .reset_index()
)

# Add percentage column
df2['Percentage'] = (df2['Count_of_Patients'] / df2['Count_of_Patients'].sum()

# Sort by Count_of_Patients descending
df2 = df2.sort_values(by=['Count_of_Patients', 'Thalassemia', 'ChestPainType',
                      ascending=False)
```

In [63]:

```
df2 # Most heart disease cases occur in patients with Thalassemia 2 or 3, ches
```

Out[63]:

	Thalassemia	ChestPainType	NoOfBloodVesselsDuringFluroscopy	Count_of_F
17	2	2	0	
10	2	0	0	
14	2	1	0	
25	3	0	0	
26	3	0	1	
27	3	0	2	
11	2	0	1	
33	3	2	0	
18	2	2	1	
28	3	0	3	
37	3	3	0	
34	3	2	1	
30	3	1	0	
12	2	0	2	
22	2	3	0	
15	2	1	1	
2	1	0	0	
24	2	3	2	
13	2	0	3	
4	1	0	2	
21	2	2	4	
36	3	2	3	
23	2	3	1	
16	2	1	2	
8	1	2	1	
31	3	1	1	
20	2	2	3	
9	1	3	0	
3	1	0	1	
35	3	2	2	
32	3	1	4	

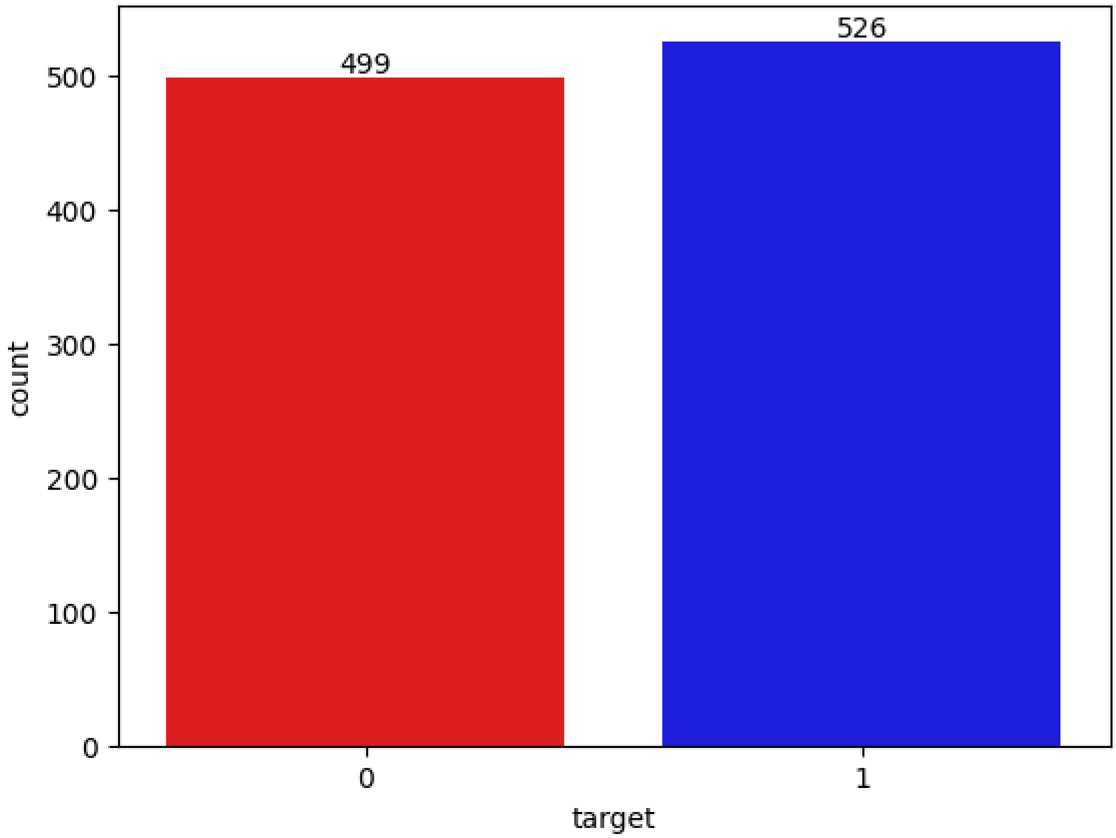
	Thalassemia	ChestPainType	NoOfBloodVesselsDuringFluroscopy	Count_of_F
6	1	1	0	
5	1	0	3	
0	0	0	0	
29	3	0	4	
19	2	2	2	
7	1	1	3	
1	0	2	0	

In [64]:
```
df.info()
```

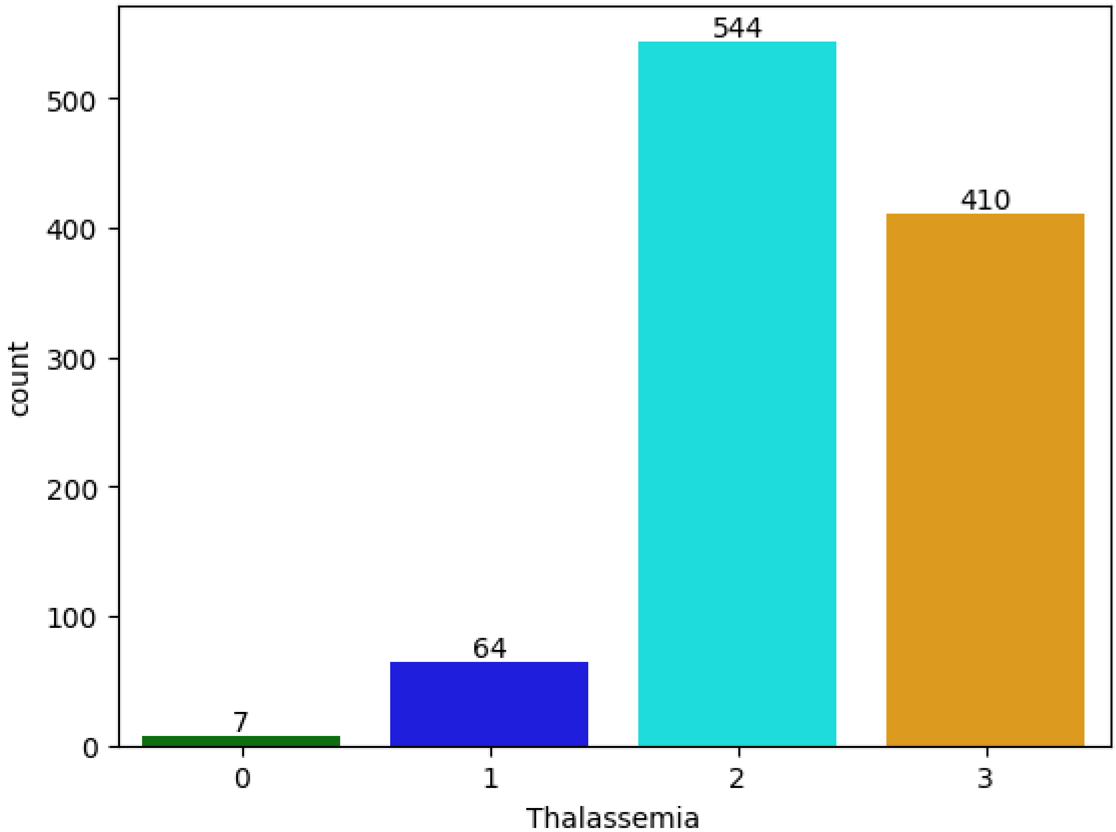
```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 1025 entries, 0 to 1024
Data columns (total 14 columns):
 #   Column                            Non-Null Count  Dtype  
---  ------                            --------------  -----  
 0   age                               1025 non-null   int64  
 1   sex                               1025 non-null   int64  
 2   ChestPainType                     1025 non-null   int64  
 3   RestingBloodPressure              1025 non-null   int64  
 4   Cholestrol                        1025 non-null   int64  
 5   FastingBloodSugar                 1025 non-null   int64  
 6   RestingEcg                        1025 non-null   int64  
 7   MaxHeartRate                      1025 non-null   int64  
 8   Exercise                          1025 non-null   int64  
 9   oldpeak                           1025 non-null   float64
 10  slope                             1025 non-null   int64  
 11  NoOfBloodVesselsDuringFluroscopy  1025 non-null   int64  
 12  Thalassemia                       1025 non-null   int64  
 13  target                            1025 non-null   int64  
dtypes: float64(1), int64(13)
memory usage: 112.2 KB
```

In [65]:
```
import warnings
warnings.filterwarnings('ignore')
```

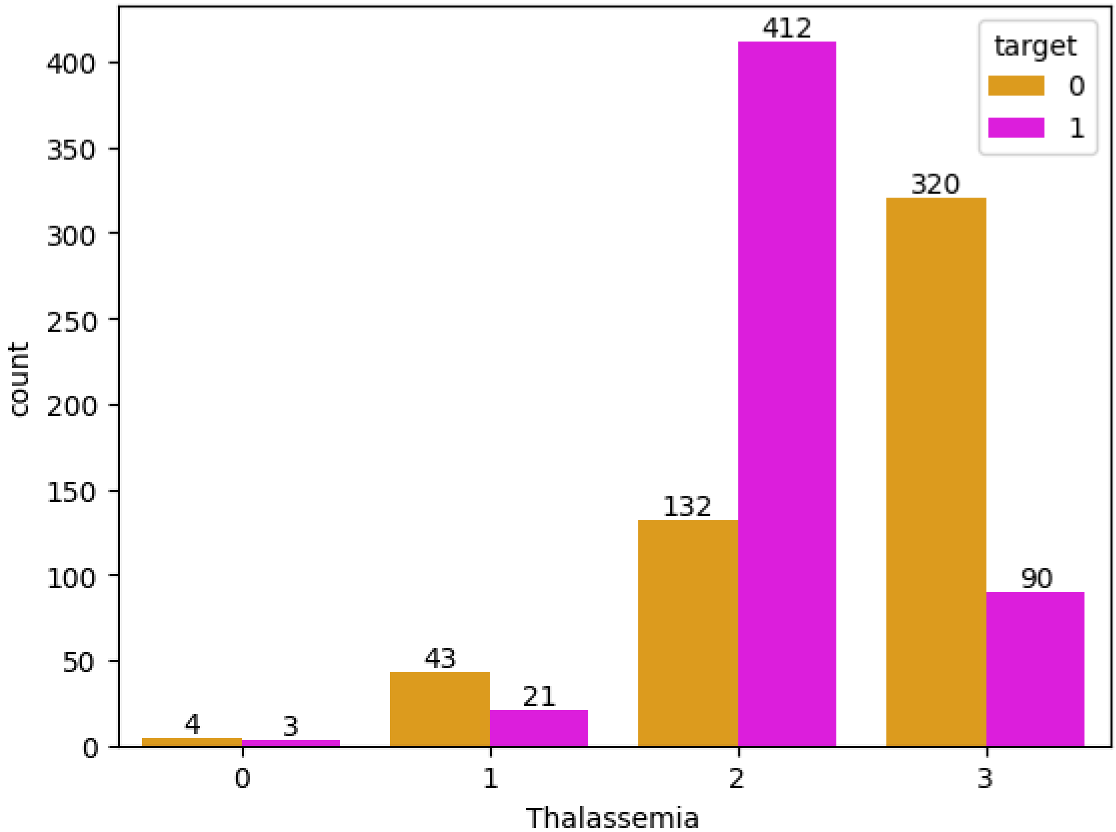
In [66]:
```
#to check target col count
a = sns.countplot(x = 'target',data = df , palette = ['red','blue'])
for ax in a.containers:
    a.bar_label(ax)
```



In [67]:
```
#to check Thalassemia count
a = sns.countplot(x = 'Thalassemia',data = df , palette = ['green','blue','cya
for ax in a.containers:
    a.bar_label(ax)
```



In [68]:
```
# to check thallasmia with target as hue
a = sns.countplot(x = 'Thalassemia',data = df ,hue = 'target',palette = ['orar
for ax in a.containers:
    a.bar_label(ax) #majority of them got thallasmia 2 and are prone to heart
```



In [69]:
```
df.columns
```

Out[69]:
```
Index(['age', 'sex', 'ChestPainType', 'RestingBloodPressure', 'Cholestrol',
       'FastingBloodSugar', 'RestingEcg', 'MaxHeartRate', 'Exercise',
       'oldpeak', 'slope', 'NoOfBloodVesselsDuringFluroscopy', 'Thalassemia',
       'target'],
      dtype='object')
```

In [70]:
```
df.head()
```

Out[70]:

	age	sex	ChestPainType	RestingBloodPressure	Cholestrol	FastingBloodSug
0	52	1	0	125	212	
1	53	1	0	140	203	
2	70	1	0	145	174	
3	61	1	0	148	203	
4	62	0	0	138	294	

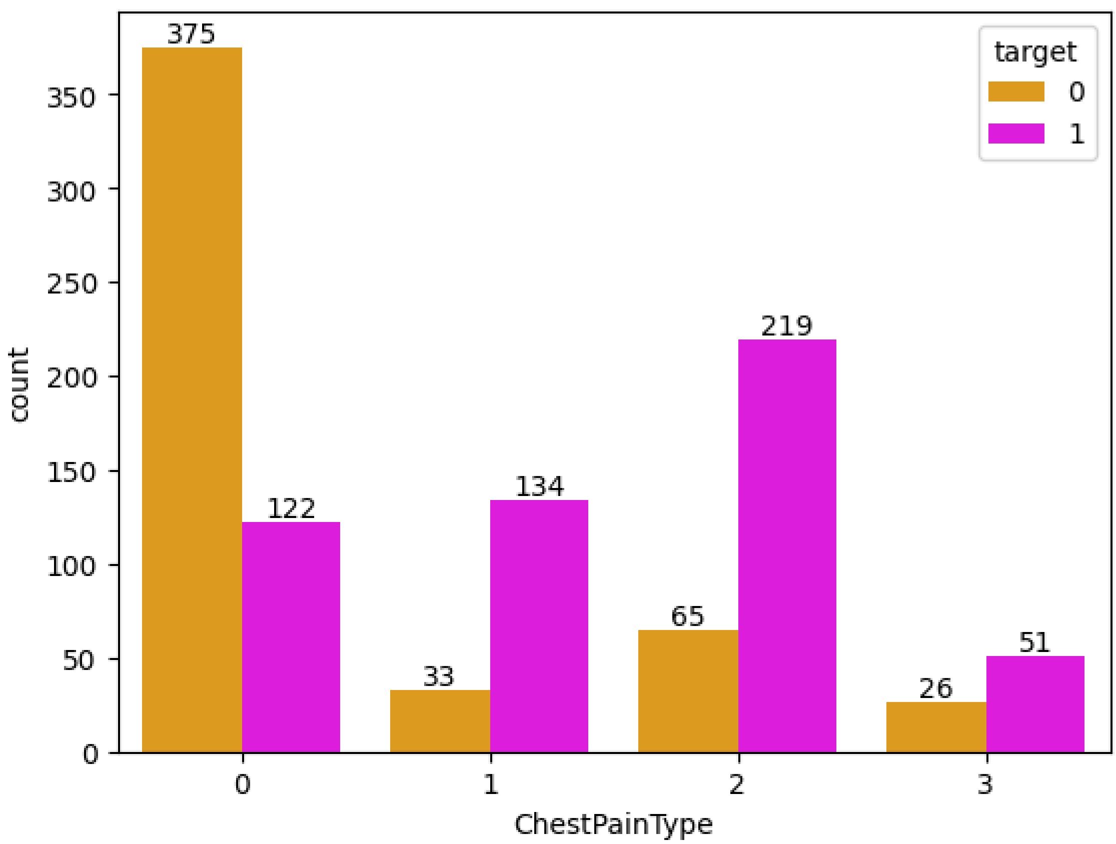
In [71]:
```
df['Thalassemia'].value_counts()
```

Out[71]:
```
Thalassemia
2    544
3    410
1     64
0      7
Name: count, dtype: int64
```

In [72]:
```
df['NoOfBloodVesselsDuringFluroscopy'].value_counts()
```

Out[72]:
```
NoOfBloodVesselsDuringFluroscopy
0    578
1    226
2    134
3     69
4     18
Name: count, dtype: int64
```

In [73]:
```
#chest pain type with target
a = sns.countplot(x = 'ChestPainType',data = df ,hue = 'target',palette = ['or
for ax in a.containers:
    a.bar_label(ax)#people who got chesspaintype 0 , majority of them didnt go
```

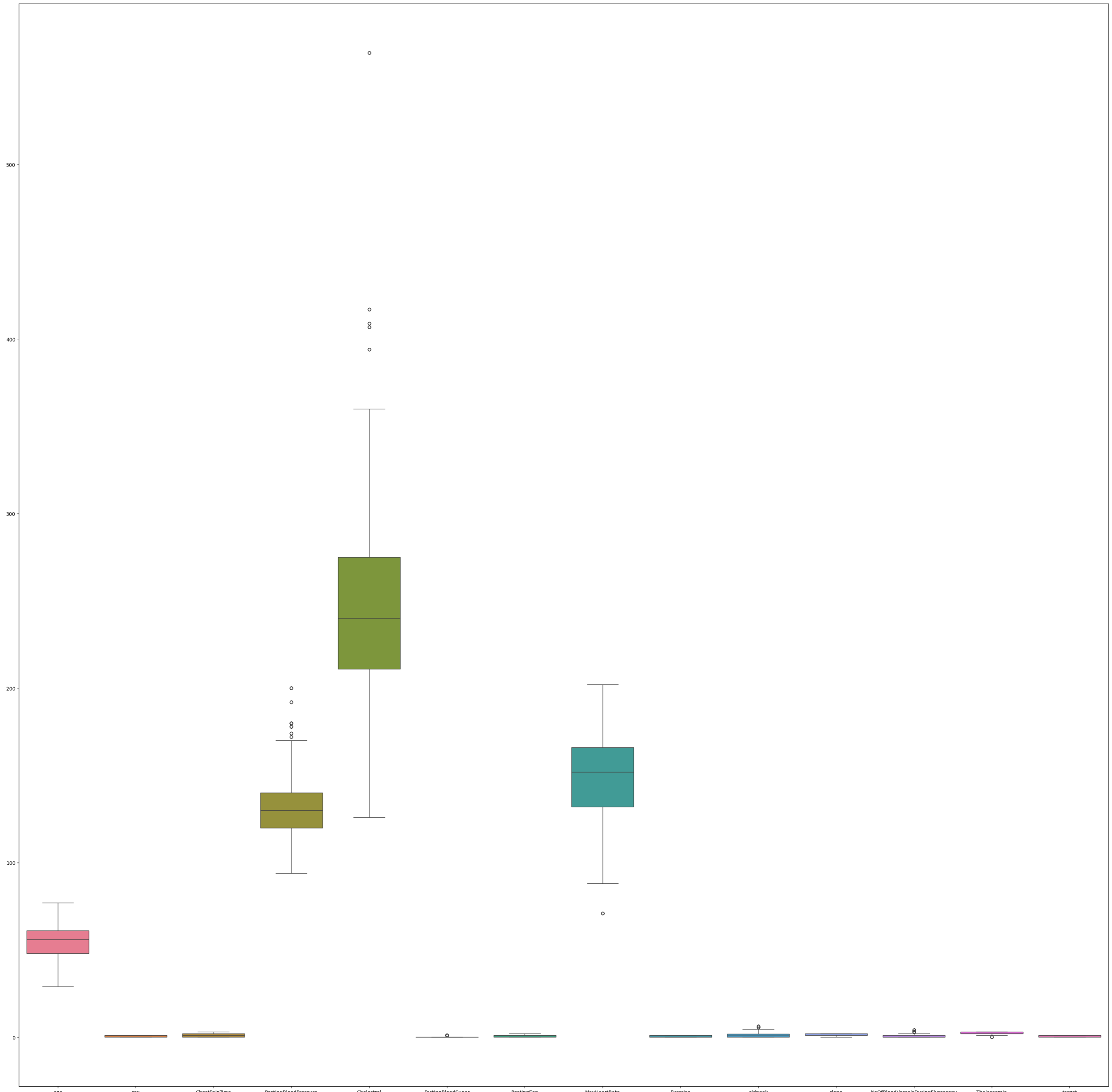


Step 5 - Outlier Detection

In [74]:
```
#to find any outliers in dataset
```

```python
plt.figure(figsize=(40,40))
sns.boxplot(df)
```

Out[74]: <Axes: >



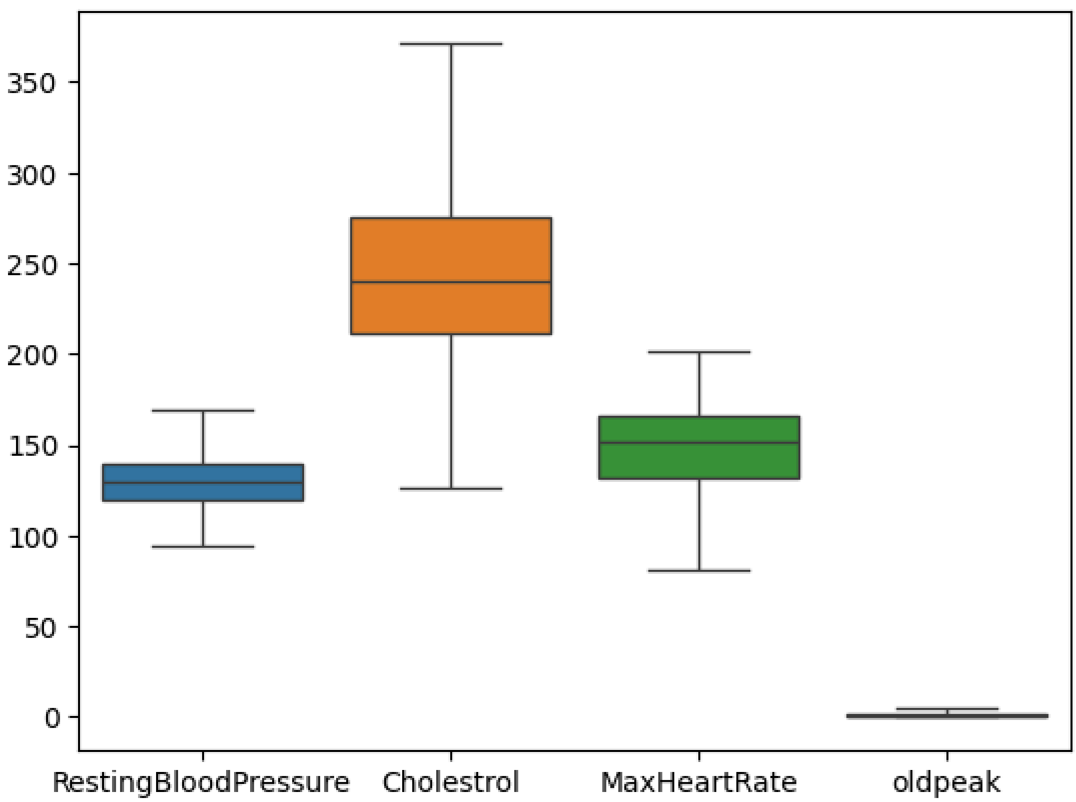
In [75]:
```python
#outliers are present in RestingBloodPressure , Cholestrol , FastingBloodSugar
# Columns with outliers
num_cols = ['RestingBloodPressure', 'Cholestrol', 'MaxHeartRate', 'oldpeak']
cat_ord_cols = ['FastingBloodSugar', 'NoOfBloodVesselsDuringFluroscopy', 'Thal
```

In [76]:
```python
#to solve the outliers

#for numerical columns
for col in num_cols:
    Q1 = df[col].quantile(0.25)
    Q3 = df[col].quantile(0.75)
```

```
    IQR = Q3 - Q1
    lower = Q1 - 1.5 * IQR
    upper = Q3 + 1.5 * IQR
    df[col] = df[col].clip(lower, upper)
#for categorical columns

# Example: FastingBloodSugar (0 or 1)
df['FastingBloodSugar'] = df['FastingBloodSugar'].clip(0, 1)

# Number of vessels (0–4)
df['NoOfBloodVesselsDuringFluroscopy'] = df['NoOfBloodVesselsDuringFluroscopy'

# Thalassemia (0–3)
df['Thalassemia'] = df['Thalassemia'].clip(0, 3)
```

In [77]:
```
sns.boxplot(data=df[num_cols]) #outliers removed successfully
```

Out[77]: <Axes: >



In [78]:
```
df.head()
```

Out[78]:

	age	sex	ChestPainType	RestingBloodPressure	Cholestrol	FastingBloodSug
0	52	1	0	125	212	
1	53	1	0	140	203	
2	70	1	0	145	174	
3	61	1	0	148	203	
4	62	0	0	138	294	

In [79]:
```
df['oldpeak'] = df['oldpeak'].astype('int64')
df['oldpeak'].head()
```

Out[79]:
```
0    1
1    3
2    2
3    0
4    1
Name: oldpeak, dtype: int64
```

In [80]:
```
df.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 1025 entries, 0 to 1024
Data columns (total 14 columns):
 #   Column                           Non-Null Count  Dtype
---  ------                           --------------  -----
 0   age                              1025 non-null   int64
 1   sex                              1025 non-null   int64
 2   ChestPainType                    1025 non-null   int64
 3   RestingBloodPressure             1025 non-null   int64
 4   Cholestrol                       1025 non-null   int64
 5   FastingBloodSugar                1025 non-null   int64
 6   RestingEcg                       1025 non-null   int64
 7   MaxHeartRate                     1025 non-null   int64
 8   Exercise                         1025 non-null   int64
 9   oldpeak                          1025 non-null   int64
 10  slope                            1025 non-null   int64
 11  NoOfBloodVesselsDuringFluroscopy 1025 non-null   int64
 12  Thalassemia                      1025 non-null   int64
 13  target                           1025 non-null   int64
dtypes: int64(14)
memory usage: 112.2 KB
```

Step 6 - Model Building

In [81]:
```
#splitting dataset into Dependant and Independant columns
X = df.drop('target',axis = 1)
y = df['target']
```

In [82]:
```
#split dataset into 20 % test and 80% train
X_train, X_test, y_train, y_test = train_test_split(X,y,test_size= 0.2 , rando
```

In [83]:
```
len(X_train)
```

Out[83]: 820

In [84]:
```
len(X_test)
```

Out[84]: 205

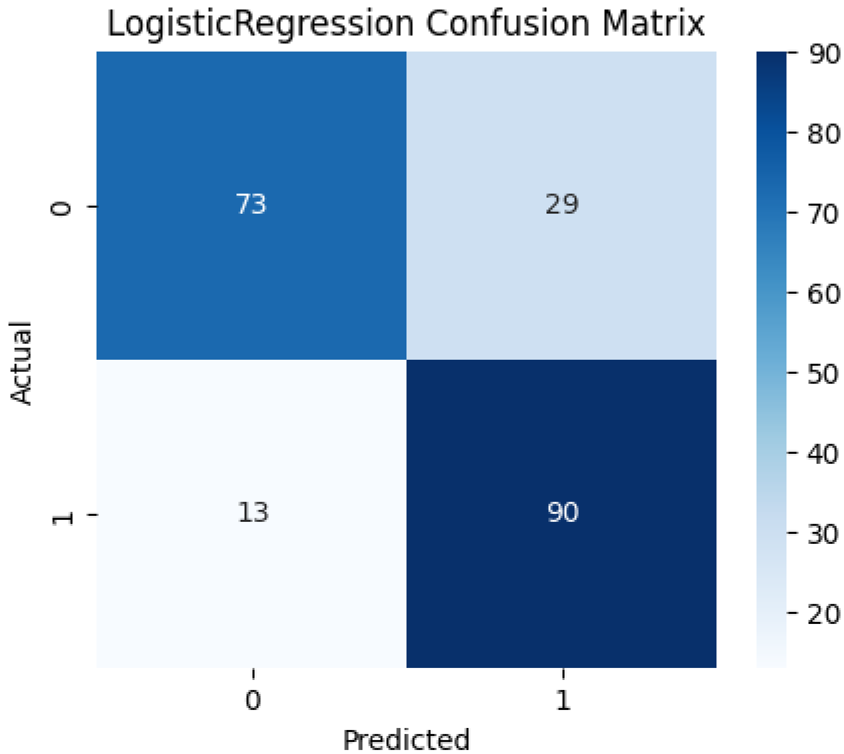
In [85]:
```
#standard scaler
scaler = StandardScaler()
X_train= scaler.fit_transform(X_train)
X_test = scaler.transform(X_test)
```

In [86]:
```
#perform models

models = [LogisticRegression(),DecisionTreeClassifier(),RandomForestClassifier
results = {}
```

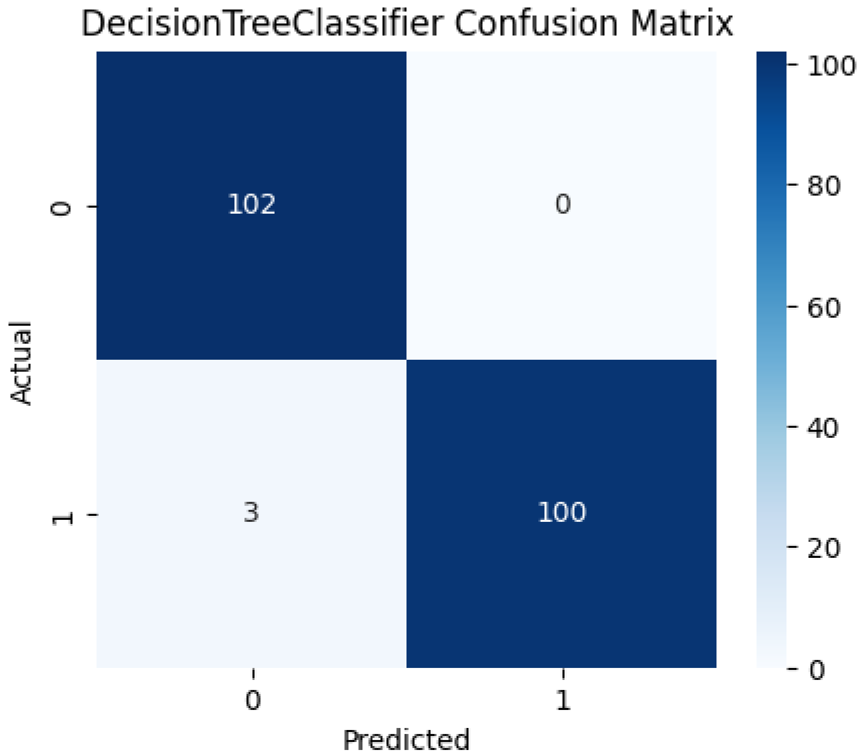
In [87]:
```
# Loop through models
for model in models:
    print(f"Model: {model.__class__.__name__}")

    # Fit model
    model.fit(X_train, y_train)

    # Predictions
    y_pred = model.predict(X_test)

    # Accuracy
    train_score = model.score(X_train, y_train)
    test_score = model.score(X_test, y_test)
    results[model.__class__.__name__] = test_score

    print(f"Training Accuracy: {train_score:.4f}")
    print(f"Test Accuracy: {test_score:.4f}")

    # Overfitting check
    if train_score > test_score + 0.05:
        print("⚠ Possible Overfitting detected!")
    elif test_score > train_score + 0.05:
        print("⚠ Possible Underfitting detected!")
    else:
        print("✅ Model seems balanced.")

    # Confusion Matrix
    cm = confusion_matrix(y_test, y_pred)
    print("Confusion Matrix:")
    print(cm)
```

```python
    # Optional: visualize confusion matrix
    plt.figure(figsize=(5,4))
    sns.heatmap(cm, annot=True, fmt="d", cmap="Blues")
    plt.xlabel("Predicted")
    plt.ylabel("Actual")
    plt.title(f"{model.__class__.__name__} Confusion Matrix")
    plt.show()

    # Classification Report
    print("Classification Report:")
    print(classification_report(y_test, y_pred))
    print("-"*160)

# Optional: Compare model accuracies
plt.figure(figsize=(22,22))
#models = [LogisticRegression(),DecisionTreeClassifier(),RandomForestClassifie
a = sns.barplot(x=list(results.keys()), y=list(results.values()),palette = ['r
for ax in a.containers:
    a.bar_label(ax)
plt.ylabel("Test Accuracy")
plt.title("Comparison of Model Accuracies")
plt.ylim(0,1)
plt.show()
```

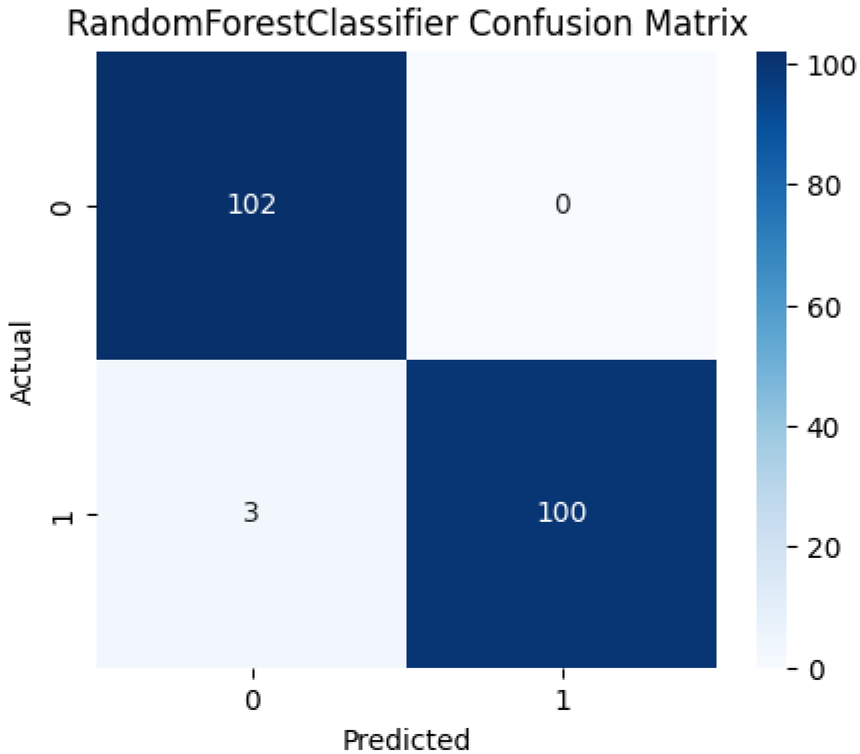
```
Model: LogisticRegression
Training Accuracy: 0.8634
Test Accuracy: 0.7951
⚠ Possible Overfitting detected!
Confusion Matrix:
[[73 29]
 [13 90]]
```



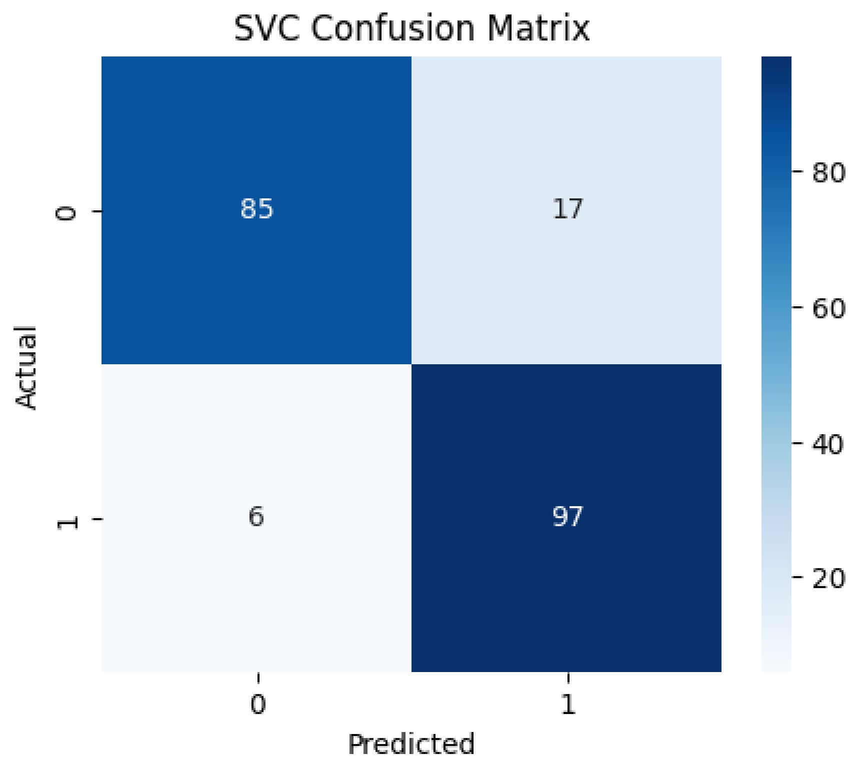

```
Classification Report:
              precision    recall  f1-score   support

           0       0.85      0.72      0.78       102
           1       0.76      0.87      0.81       103

    accuracy                           0.80       205
   macro avg       0.80      0.79      0.79       205
weighted avg       0.80      0.80      0.79       205

--------------------------------------------------------------------------------
--------------------------------------------------------------------------------
--
Model: DecisionTreeClassifier
Training Accuracy: 1.0000
Test Accuracy: 0.9854
✅ Model seems balanced.
Confusion Matrix:
[[102   0]
 [  3 100]]
```



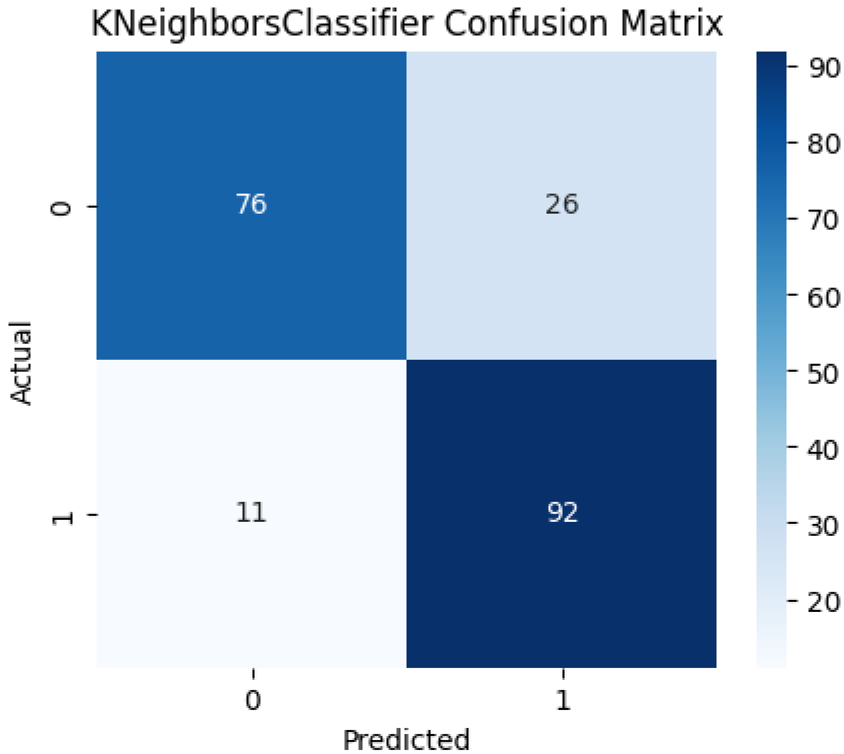

```
Classification Report:
              precision    recall  f1-score   support

           0       0.97      1.00      0.99       102
           1       1.00      0.97      0.99       103

    accuracy                           0.99       205
   macro avg       0.99      0.99      0.99       205
weighted avg       0.99      0.99      0.99       205

--------------------------------------------------------------------------------
--------------------------------------------------------------------------------
--
Model: RandomForestClassifier
Training Accuracy: 1.0000
Test Accuracy: 0.9854
✅ Model seems balanced.
Confusion Matrix:
[[102   0]
 [  3 100]]
```



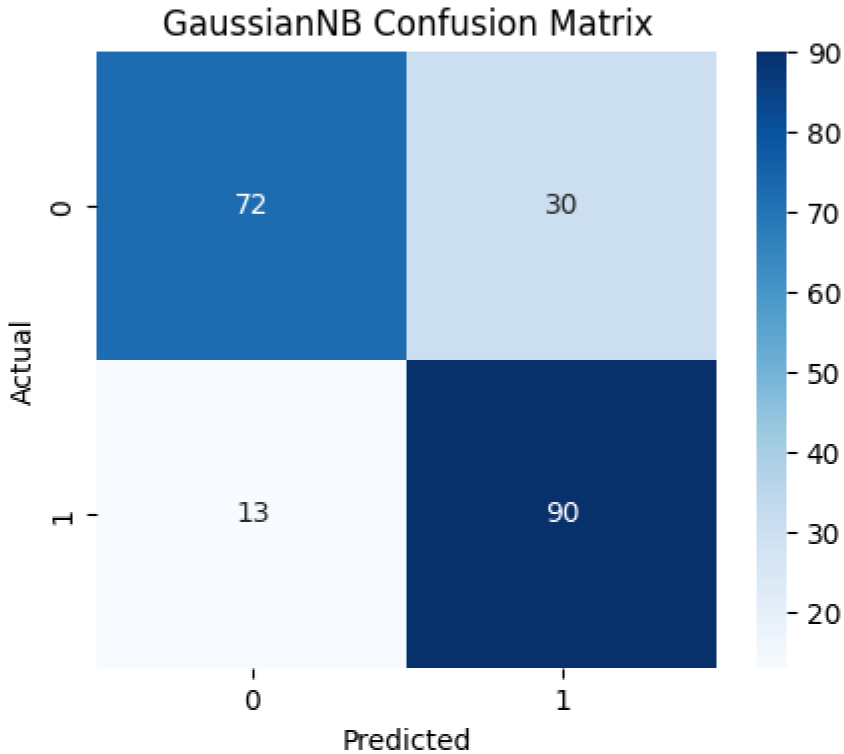

```
Classification Report:
              precision    recall  f1-score   support

           0       0.97      1.00      0.99       102
           1       1.00      0.97      0.99       103

    accuracy                           0.99       205
   macro avg       0.99      0.99      0.99       205
weighted avg       0.99      0.99      0.99       205

-------------------------------------------------------------------------------
-------------------------------------------------------------------------------
--
Model: SVC
Training Accuracy: 0.9549
Test Accuracy: 0.8878
⚠ Possible Overfitting detected!
Confusion Matrix:
[[85 17]
 [ 6 97]]
```




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

           0       0.93      0.83      0.88       102
           1       0.85      0.94      0.89       103

    accuracy                           0.89       205
   macro avg       0.89      0.89      0.89       205
weighted avg       0.89      0.89      0.89       205

--------------------------------------------------------------------------------
--------------------------------------------------------------------------------
--
Model: KNeighborsClassifier
Training Accuracy: 0.9439
Test Accuracy: 0.8195
⚠ Possible Overfitting detected!
Confusion Matrix:
[[76 26]
 [11 92]]
```



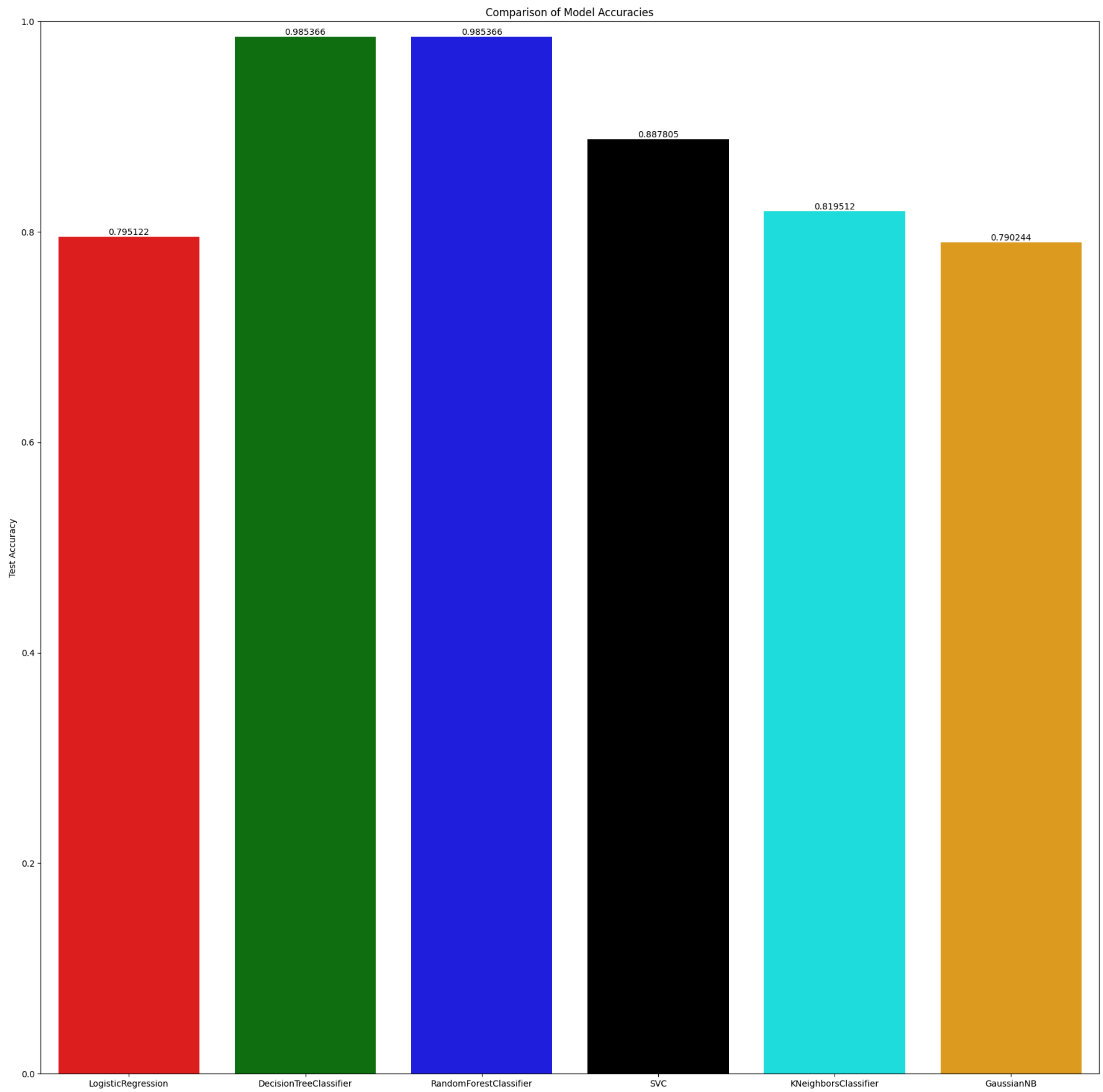

```
Classification Report:
              precision    recall  f1-score   support

           0       0.87      0.75      0.80       102
           1       0.78      0.89      0.83       103

    accuracy                           0.82       205
   macro avg       0.83      0.82      0.82       205
weighted avg       0.83      0.82      0.82       205

-------------------------------------------------------------------------------
-------------------------------------------------------------------------------
--
Model: GaussianNB
Training Accuracy: 0.8512
Test Accuracy: 0.7902
⚠ Possible Overfitting detected!
Confusion Matrix:
[[72 30]
 [13 90]]
```



Classification Report:

	precision	recall	f1-score	support
0	0.85	0.71	0.77	102
1	0.75	0.87	0.81	103
accuracy			0.79	205
macro avg	0.80	0.79	0.79	205
weighted avg	0.80	0.79	0.79	205

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



step 7 - Model Optimization Remove overfitting

In [88]:
```
#define hyperparameters
# step 1 - Models defining
models = {
    'LogisticRegression': LogisticRegression(max_iter=10000),
    'SVC': SVC(),
    'KNeighborsClassifier': KNeighborsClassifier(),
    'GaussianNB': GaussianNB(),
    'DecisionTreeClassifier': DecisionTreeClassifier(),
    'RandomForestClassifier': RandomForestClassifier()
}
```

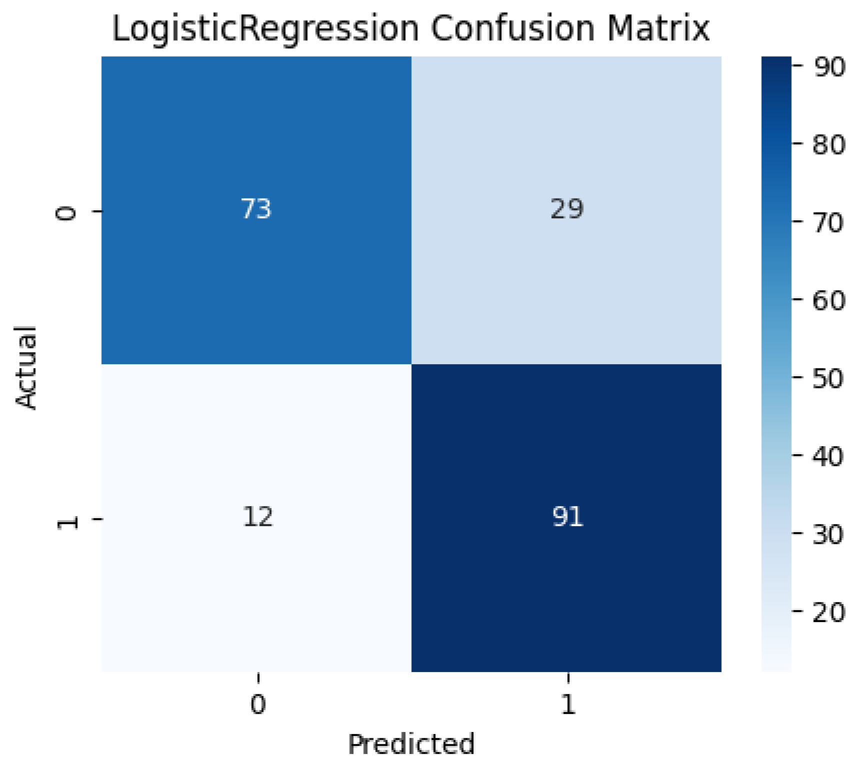
In [89]:
```python
#step 2 - defining hyperparameters
param_distributions = {
    'LogisticRegression': {
        'C': [0.1, 0.5, 1, 2],
        'penalty': ['l2','l1'],
        'solver': ['lbfgs', 'liblinear']
    },
    'SVC': {
        'C': [0.1, 0.5, 1, 2, 5],
        'kernel': ['linear', 'rbf'],
        'gamma': ['scale']
    },
    'KNeighborsClassifier': {
        'n_neighbors': list(range(5,16)),
        'weights': ['distance'],
        'p': [1,2]
    },
    'GaussianNB': {
        'var_smoothing': [1e-9, 1e-8, 1e-7, 1e-6]
    },
    'DecisionTreeClassifier': {
        'max_depth': list(range(3,9)),
        'min_samples_split': list(range(2,9)),
        'min_samples_leaf': list(range(1,5))
    },
    'RandomForestClassifier': {
        'n_estimators': list(range(50,151,10)),
        'max_depth': list(range(3,11)),
        'min_samples_split': list(range(2,9)),
        'min_samples_leaf': list(range(1,5))
    }
}
```

In [90]:
```python
# step 3 -  Dictionary to store test accuracies
results = {}
```

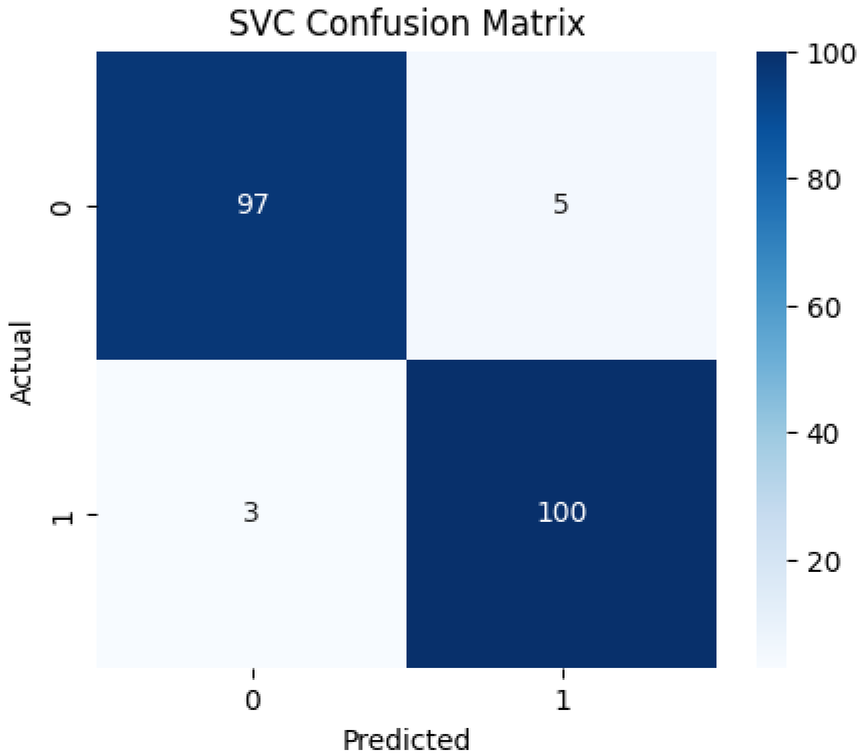
In [91]:
```python
# step 4 - Hyperparameter tuning and evaluation loop
for name, model in models.items():
    print(f"🔹 Tuning {name}...")
    # step 5 - perform Randomised search and select best parameters
    rand_search = RandomizedSearchCV(estimator=model,param_distributions=param
    rand_search.fit(X_train, y_train)
    best_model = rand_search.best_estimator_

    # Predictions
    y_pred = best_model.predict(X_test)

    # step 6 - Accuracy
    train_score = best_model.score(X_train, y_train)
    test_score = best_model.score(X_test, y_test)
    results[name] = test_score

    print(f"Best Parameters: {rand_search.best_params_}")
```

```
    print(f"Training Accuracy: {train_score:.4f}")
    print(f"Test Accuracy: {test_score:.4f}")

    # step 7 - Overfitting check
    if train_score > test_score + 0.05:
        print("⚠ Possible Overfitting detected!")
    else:
        print("✅ Model seems balanced.")

    # step 8 - Confusion Matrix
    cm = confusion_matrix(y_test, y_pred)
    plt.figure(figsize=(5,4))
    sns.heatmap(cm, annot=True, fmt="d", cmap="Blues")
    plt.xlabel("Predicted")
    plt.ylabel("Actual")
    plt.title(f"{name} Confusion Matrix")
    plt.show()

    # Classification Report
    print("Classification Report:")
    print(classification_report(y_test, y_pred))
    print("-"*60)
# Compare model accuracies
plt.figure(figsize=(23,23))
a= sns.barplot(x=list(results.keys()), y=list(results.values()),palette = ['gr
for ax in a.containers:
    a.bar_label(ax)
plt.xlabel("Machine Learning Models")
plt.ylabel("Test Accuracy")
plt.title("Comparison of Model Accuracies After Hyperparameter Tuning")
plt.ylim(0,1)
plt.show()
```

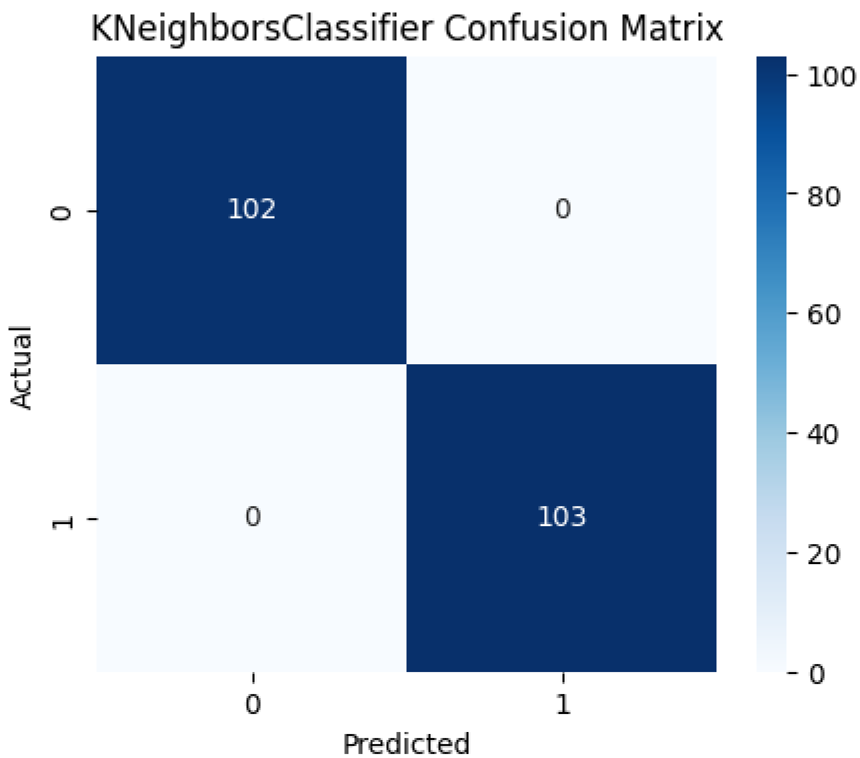
```
🔹 Tuning LogisticRegression...
Best Parameters: {'solver': 'lbfgs', 'penalty': 'l2', 'C': 0.1}
Training Accuracy: 0.8659
Test Accuracy: 0.8000
⚠ Possible Overfitting detected!
```



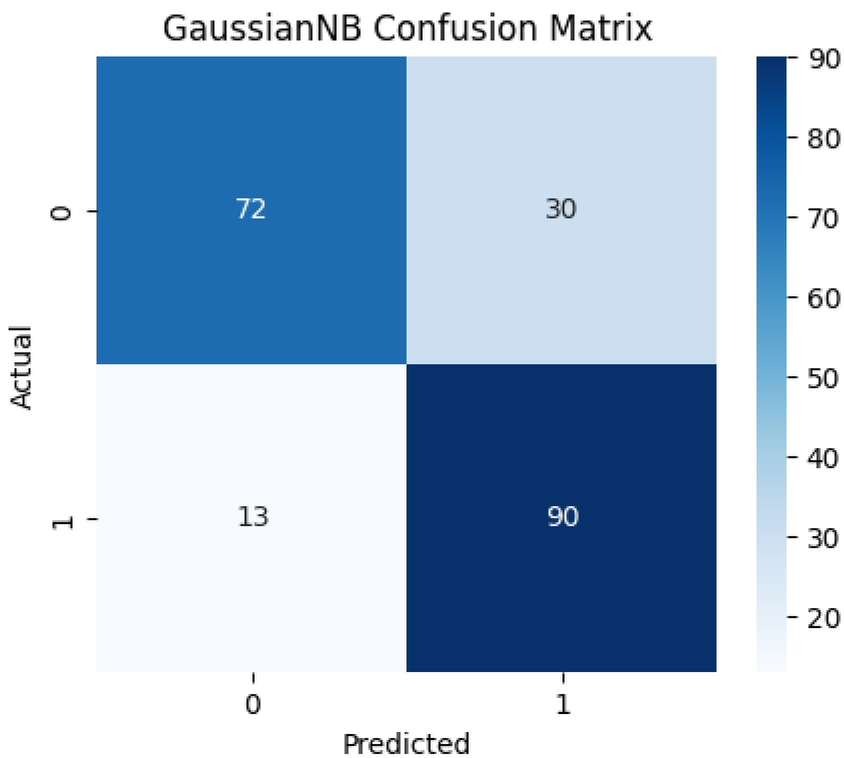

```
Classification Report:
              precision    recall  f1-score   support

           0       0.86      0.72      0.78       102
           1       0.76      0.88      0.82       103

    accuracy                           0.80       205
   macro avg       0.81      0.80      0.80       205
weighted avg       0.81      0.80      0.80       205

------------------------------------------------------------
🔹 Tuning SVC...
Best Parameters: {'kernel': 'rbf', 'gamma': 'scale', 'C': 5}
Training Accuracy: 0.9915
Test Accuracy: 0.9610
✅ Model seems balanced.
```



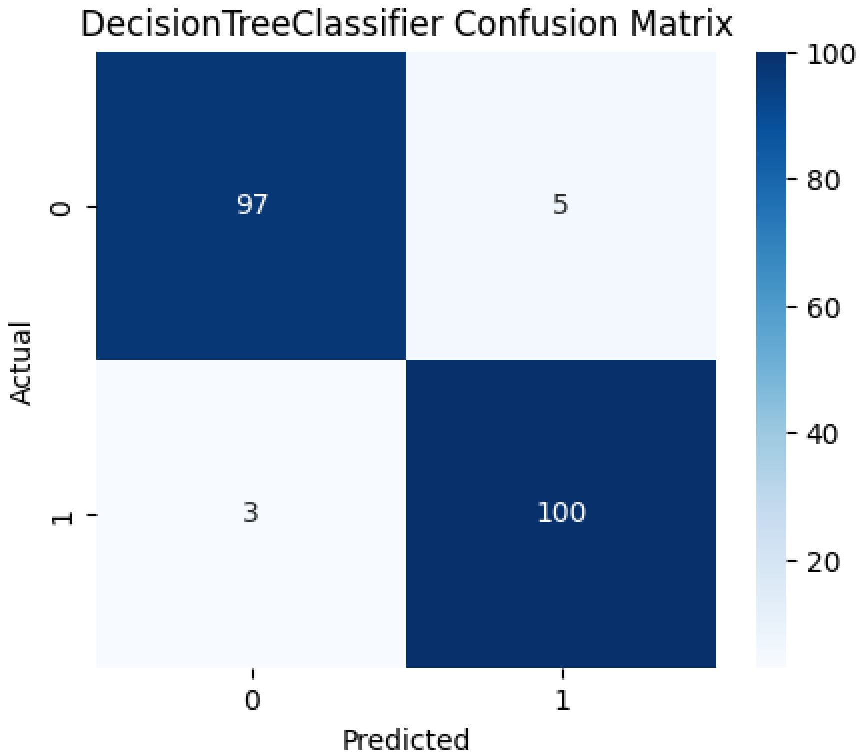

```
Classification Report:
              precision    recall  f1-score   support

           0       0.97      0.95      0.96       102
           1       0.95      0.97      0.96       103

    accuracy                           0.96       205
   macro avg       0.96      0.96      0.96       205
weighted avg       0.96      0.96      0.96       205

------------------------------------------------------------
🔹 Tuning KNeighborsClassifier...
Best Parameters: {'weights': 'distance', 'p': 1, 'n_neighbors': 10}
Training Accuracy: 1.0000
Test Accuracy: 1.0000
✅ Model seems balanced.
```



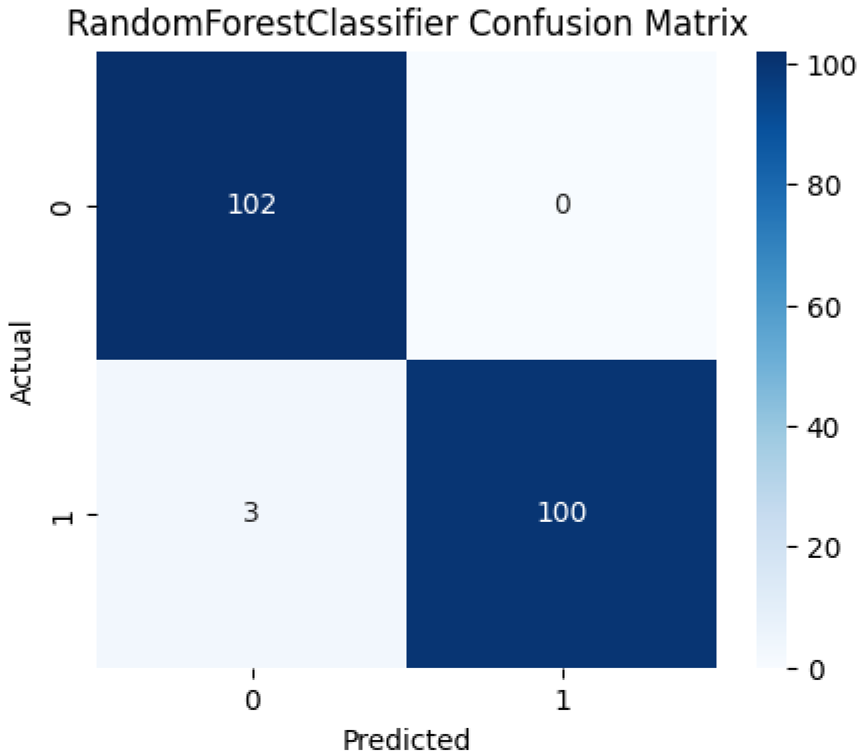

```
Classification Report:
              precision    recall  f1-score   support

           0       1.00      1.00      1.00       102
           1       1.00      1.00      1.00       103

    accuracy                           1.00       205
   macro avg       1.00      1.00      1.00       205
weighted avg       1.00      1.00      1.00       205

------------------------------------------------------------
🔹 Tuning GaussianNB...
Best Parameters: {'var_smoothing': 1e-09}
Training Accuracy: 0.8512
Test Accuracy: 0.7902
⚠ Possible Overfitting detected!
```



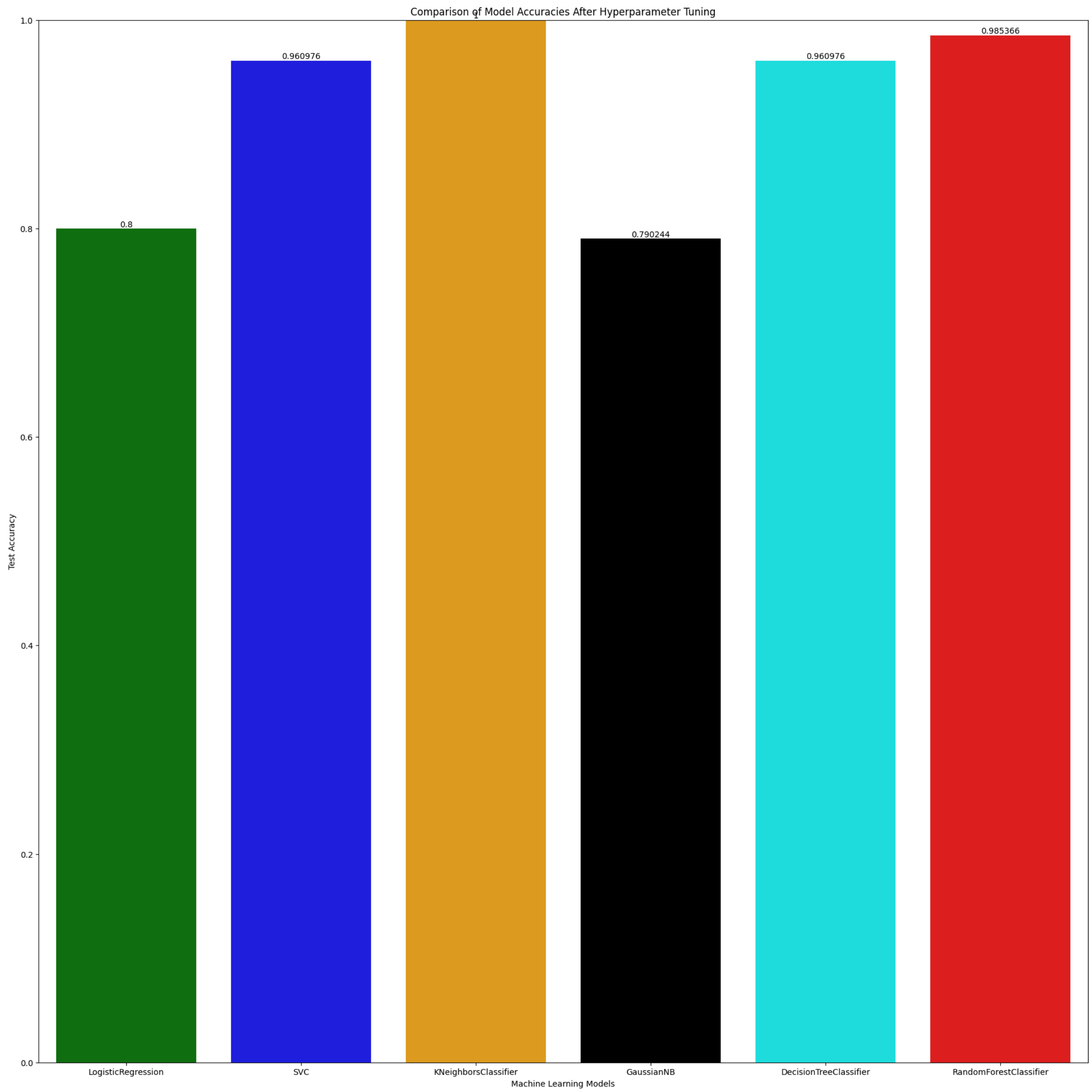

```
Classification Report:
              precision    recall  f1-score   support

           0       0.85      0.71      0.77       102
           1       0.75      0.87      0.81       103

    accuracy                           0.79       205
   macro avg       0.80      0.79      0.79       205
weighted avg       0.80      0.79      0.79       205

------------------------------------------------------------
🔹 Tuning DecisionTreeClassifier...
Best Parameters: {'min_samples_split': 4, 'min_samples_leaf': 1, 'max_depth':
8}
Training Accuracy: 0.9927
Test Accuracy: 0.9610
✅ Model seems balanced.
```




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

           0       0.97      0.95      0.96       102
           1       0.95      0.97      0.96       103

    accuracy                           0.96       205
   macro avg       0.96      0.96      0.96       205
weighted avg       0.96      0.96      0.96       205

------------------------------------------------------------
🔹 Tuning RandomForestClassifier...
Best Parameters: {'n_estimators': 130, 'min_samples_split': 3, 'min_samples_lea
f': 1, 'max_depth': 10}
Training Accuracy: 1.0000
Test Accuracy: 0.9854
✅ Model seems balanced.
```




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

           0       0.97      1.00      0.99       102
           1       1.00      0.97      0.99       103

    accuracy                           0.99       205
   macro avg       0.99      0.99      0.99       205
weighted avg       0.99      0.99      0.99       205

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



In [92]:

```
#knn is the best method - test it

# Best KNN model
best_knn = KNeighborsClassifier(weights='distance', p=1, n_neighbors=10)
best_knn.fit(X_train, y_train)
y_pred = best_knn.predict(X_test)

print("Test Accuracy:", accuracy_score(y_test, y_pred))
print("Confusion Matrix:\n", confusion_matrix(y_test, y_pred))
print("Classification Report:\n", classification_report(y_test, y_pred))
```

```
Test Accuracy: 1.0
Confusion Matrix:
 [[102   0]
 [  0 103]]
Classification Report:
               precision    recall  f1-score   support

           0       1.00      1.00      1.00       102
           1       1.00      1.00      1.00       103

    accuracy                           1.00       205
   macro avg       1.00      1.00      1.00       205
weighted avg       1.00      1.00      1.00       205
```

step 8 - Deep Learning model

metrics types

Metric	Type	Description / What It Measures	Use Case / Example
`accuracy`	Classification	Overall fraction of correctly predicted samples.	General classification tasks.
`binary_accuracy`	Classification	Accuracy for binary classification (0 or 1).	Binary output (e.g., disease yes/no).
`categorical_accuracy`	Classification	Accuracy for multi-class with one-hot labels.	Multi-class (e.g., image classification).
`sparse_categorical_accuracy`	Classification	Accuracy for multi-class with integer labels.	Multi-class (without one-hot encoding).
`Precision`	Classification	% of correct positive predictions among all predicted positives.	When false positives are costly (e.g., spam detection).
`Recall`	Classification	% of actual positives correctly predicted.	When missing positives is costly (e.g., cancer detection).

Metric	Type	Description / What It Measures	Use Case / Example
`F1-Score`	Classification	Harmonic mean of precision and recall.	Balanced measure for imbalanced data.
`AUC` **(Area Under Curve)**	Classification	Measures ability to distinguish between classes.	Evaluating model performance (ROC curve).
`TruePositives`	Classification	Number of correctly identified positive samples.	Model diagnostics.
`TrueNegatives`	Classification	Number of correctly identified negatives.	Model diagnostics.
`FalsePositives`	Classification	Incorrectly predicted positives.	Helps understand model errors.
`FalseNegatives`	Classification	Missed positive cases.	Important in medical or fraud detection.
`MeanSquaredError (MSE)`	Regression	Average squared difference between actual and predicted.	Continuous outputs (e.g., price prediction).
`MeanAbsoluteError (MAE)`	Regression	Average absolute difference between actual and predicted.	Regression — less sensitive to outliers.
`RootMeanSquaredError (RMSE)`	Regression	Square root of MSE; penalizes large errors more.	Regression with emphasis on large errors.
`R² (Coefficient of Determination)`	Regression	Measures proportion of variance explained by model.	Regression performance measure.
`MeanAbsolutePercentageError (MAPE)`	Regression	Average % difference between prediction and actual value.	Forecasting models.
`cosine_similarity`	Similarity	Measures similarity between two	NLP, embeddings, recommendation

Metric	Type	Description / What It Measures	Use Case / Example
		vectors (ranges -1 to 1).	systems.
`log_cosh_error`	Regression	Smooth alternative to MSE; robust to outliers.	Regression tasks requiring smooth gradients.

optimizer types -

Optimizer	Description
`SGD`	Stochastic Gradient Descent — updates weights using gradients; simple but may converge slowly.
`SGD (with Momentum)`	Adds momentum to accelerate learning and avoid local minima.
`Nesterov`	A variant of momentum that looks ahead before updating parameters for faster convergence.
`Adagrad`	Adapts learning rates for each parameter; good for sparse data but learning rate decays quickly.
`RMSprop`	Adapts learning rate using a moving average of squared gradients; good for non-stationary data.
`Adam`	Combines Momentum and RMSprop; most widely used optimizer in deep learning.
`Adamax`	Variant of Adam using the infinity norm for better stability in some cases.
`Nadam`	Adam + Nesterov Momentum; slightly faster convergence than Adam.
`Ftrl`	Useful for large-scale linear models; supports L1 and L2 regularization.
`Adadelta`	Extension of Adagrad that reduces its aggressive, monotonically decreasing learning rate.
`Lion`	New optimizer that uses sign-based gradient updates for faster and more stable training.

Loss types

Loss Function	Type	Description	When to Use / Example
`binary_crossentropy`	Classification	Measures the difference between two probability distributions for binary (0/1) classification.	Binary classification (e.g., heart disease prediction).
`categorical_crossentropy`	Classification	Used for multi-class classification with one-hot encoded labels.	Multi-class classification (e.g., digit recognition with one-hot labels).
`sparse_categorical_crossentropy`	Classification	Same as above but for integer class labels (not one-hot).	Multi-class classification with integer targets.
`mean_squared_error (MSE)`	Regression	Measures average squared difference between actual and predicted values.	Regression tasks (e.g., predicting prices, continuous outputs).
`mean_absolute_error (MAE)`	Regression	Measures average absolute difference between actual and predicted values.	Regression, less sensitive to outliers than MSE.
`hinge`	Classification	Used for “maximum-margin” classification (like SVMs).	Binary classification with {-1, +1} labels.
`squared_hinge`	Classification	Square of hinge loss; penalizes larger margin	Similar to hinge, but smoother gradients.

Loss Function	Type	Description	When to Use / Example
		violations more strongly.	
`kullback_leibler_divergence (KL Divergence)`	Probabilistic Models	Measures how one probability distribution diverges from another.	Variational autoencoders (VAEs), probabilistic models.
`poisson`	Regression / Count Data	Suitable for modeling count data (e.g., event counts).	When output represents count rates (e.g., number of calls, clicks).
`cosine_similarity`	Similarity-Based	Measures cosine similarity between predicted and true values.	NLP, embeddings, or recommendation systems.
`huber`	Regression	Combination of MSE and MAE — less sensitive to outliers.	Robust regression with outlier resistance.
`log_cosh`	Regression	Logarithm of hyperbolic cosine of prediction error; smooth alternative to MSE.	When you want smooth gradients and robustness.

| `categorical_hinge` | Classification | Hinge loss for multi-class problems. | Multi-class problems with {-1, +1} encoded labels.

Activation Functions

Activation Function	Formula	Range	Description	When to Use
Sigmoid (σ)	σ(x) = 1 / (1 + e^(-x))	(0, 1)	Converts any real value into a probability-like output.	Output layer for **binary classification**.
Tanh	tanh(x) = (e^(x)	(-1, 1)	Zero-centered version	Hidden layers

Activation Function	Formula	Range	Description	When to Use
(Hyperbolic Tangent)	- e^(-x)) / (e^(x) + e^(-x))		of Sigmoid; stronger gradients.	when inputs are **normalized**.
ReLU (Rectified Linear Unit)	ReLU(x) = max(0, x)	[0, ∞)	Fast and efficient; avoids vanishing gradient problem.	Default choice for **hidden layers**.
Leaky ReLU	f(x) = x if x > 0 else 0.01x	(-∞, ∞)	Solves "dead neuron" issue by allowing a small negative slope.	When some neurons die with ReLU.
Parametric ReLU (PReLU)	f(x) = x if x > 0 else αx	(-∞, ∞)	Learns the slope (α) automatically for better performance.	Used in **deep CNNs**.
ELU (Exponential Linear Unit)	f(x) = x if x > 0 else α*(e^x - 1)	(-α, ∞)	Helps achieve faster and more stable learning.	For **deep networks**.
SELU (Scaled ELU)	f(x) = λ * (x if x > 0 else α*(e^x - 1))	(-∞, ∞)	Self-normalizing activation, stabilizes training.	When using **Self-Normalizing Neural Nets**.
Softmax	f(x_i) = e^(x_i) / Σ e^(x_j)	(0, 1)	Converts logits to probabilities across multiple classes.	Output layer for **multi-class classification**.
Swish	f(x) = x * sigmoid(x)	(-∞, ∞)	Smooth and non-monotonic; better than ReLU in deep models.	For **deep neural networks**.
GELU (Gaussian Error Linear Unit)	f(x) = x * Φ(x)	(-∞, ∞)	Used in Transformers; smooth like Swish.	**NLP models** (e.g., BERT, GPT).
Softplus	f(x) = ln(1 + e^x)	(0, ∞)	Smooth approximation of ReLU; differentiable everywhere.	When smooth output is required.
Hard Sigmoid	f(x) = clip((x * 0.2) + 0.5, 0, 1)	(0, 1)	Fast, piecewise linear version of Sigmoid.	For **mobile/low-power models**.
Hard Swish	f(x) = x * ReLU6(x + 3) / 6	(-∞, ∞)	Lightweight and efficient; used in MobileNetV3.	For **mobile and embedded models**.

In [93]:

```
# Define ANN model
# ==========================
model = Sequential([
    Dense(64, input_dim=X_train.shape[1], activation='relu'),  # Hidden layer
    Dense(32, activation='relu'),                               # Hidden layer
    Dense(16, activation='relu'),                               # Hidden layer
    Dense(1, activation='sigmoid')                              # Output layer
])
```

In [94]:
```python
# Compile the model
# Compile model
# ==========================
model.compile(optimizer="adam", loss='binary_crossentropy', metrics=['accuracy
```

In [95]:
```python
# Train the model
# ==========================
history = model.fit(
    X_train, y_train,
    validation_split=0.2,
    epochs=100,
    batch_size=16,
    verbose=2  # One line per epoch
)
```

```
Epoch 1/100
41/41 - 1s - 21ms/step - accuracy: 0.7820 - binary_accuracy: 0.7820 - loss: 0.5
543 - recall: 0.7626 - val_accuracy: 0.8171 - val_binary_accuracy: 0.8171 - va
l_loss: 0.4691 - val_recall: 0.8023
Epoch 2/100
41/41 - 0s - 1ms/step - accuracy: 0.8704 - binary_accuracy: 0.8704 - loss: 0.36
01 - recall: 0.9050 - val_accuracy: 0.8232 - val_binary_accuracy: 0.8232 - va
l_loss: 0.4131 - val_recall: 0.8721
Epoch 3/100
41/41 - 0s - 1ms/step - accuracy: 0.8918 - binary_accuracy: 0.8918 - loss: 0.28
67 - recall: 0.9318 - val_accuracy: 0.8110 - val_binary_accuracy: 0.8110 - va
l_loss: 0.4079 - val_recall: 0.8721
Epoch 4/100
41/41 - 0s - 1ms/step - accuracy: 0.9040 - binary_accuracy: 0.9040 - loss: 0.25
31 - recall: 0.9318 - val_accuracy: 0.8659 - val_binary_accuracy: 0.8659 - va
l_loss: 0.3777 - val_recall: 0.8837
Epoch 5/100
41/41 - 0s - 971us/step - accuracy: 0.9177 - binary_accuracy: 0.9177 - loss:
0.2324 - recall: 0.9318 - val_accuracy: 0.8232 - val_binary_accuracy: 0.8232 -
val_loss: 0.3945 - val_recall: 0.9070
Epoch 6/100
41/41 - 0s - 927us/step - accuracy: 0.9253 - binary_accuracy: 0.9253 - loss:
0.2087 - recall: 0.9496 - val_accuracy: 0.8659 - val_binary_accuracy: 0.8659 -
val_loss: 0.3563 - val_recall: 0.8837
Epoch 7/100
41/41 - 0s - 929us/step - accuracy: 0.9268 - binary_accuracy: 0.9268 - loss:
0.1981 - recall: 0.9407 - val_accuracy: 0.8780 - val_binary_accuracy: 0.8780 -
val_loss: 0.3499 - val_recall: 0.9070
Epoch 8/100
41/41 - 0s - 928us/step - accuracy: 0.9375 - binary_accuracy: 0.9375 - loss:
0.1797 - recall: 0.9525 - val_accuracy: 0.8841 - val_binary_accuracy: 0.8841 -
val_loss: 0.3274 - val_recall: 0.9070
Epoch 9/100
41/41 - 0s - 959us/step - accuracy: 0.9466 - binary_accuracy: 0.9466 - loss:
0.1581 - recall: 0.9555 - val_accuracy: 0.8902 - val_binary_accuracy: 0.8902 -
val_loss: 0.3270 - val_recall: 0.9070
Epoch 10/100
41/41 - 0s - 996us/step - accuracy: 0.9512 - binary_accuracy: 0.9512 - loss:
0.1451 - recall: 0.9525 - val_accuracy: 0.8841 - val_binary_accuracy: 0.8841 -
val_loss: 0.3191 - val_recall: 0.8953
Epoch 11/100
41/41 - 0s - 971us/step - accuracy: 0.9527 - binary_accuracy: 0.9527 - loss:
0.1296 - recall: 0.9585 - val_accuracy: 0.8902 - val_binary_accuracy: 0.8902 -
val_loss: 0.3182 - val_recall: 0.9070
Epoch 12/100
41/41 - 0s - 960us/step - accuracy: 0.9588 - binary_accuracy: 0.9588 - loss:
0.1174 - recall: 0.9585 - val_accuracy: 0.8963 - val_binary_accuracy: 0.8963 -
val_loss: 0.3077 - val_recall: 0.9186
Epoch 13/100
41/41 - 0s - 947us/step - accuracy: 0.9726 - binary_accuracy: 0.9726 - loss:
0.1059 - recall: 0.9822 - val_accuracy: 0.8963 - val_binary_accuracy: 0.8963 -
val_loss: 0.2884 - val_recall: 0.9186
Epoch 14/100
41/41 - 0s - 982us/step - accuracy: 0.9771 - binary_accuracy: 0.9771 - loss:
```

```
0.0930 - recall: 0.9852 - val_accuracy: 0.8902 - val_binary_accuracy: 0.8902 -
val_loss: 0.2801 - val_recall: 0.8837
Epoch 15/100
41/41 - 0s - 983us/step - accuracy: 0.9802 - binary_accuracy: 0.9802 - loss:
0.0841 - recall: 0.9881 - val_accuracy: 0.9024 - val_binary_accuracy: 0.9024 -
val_loss: 0.2795 - val_recall: 0.9302
Epoch 16/100
41/41 - 0s - 977us/step - accuracy: 0.9802 - binary_accuracy: 0.9802 - loss:
0.0744 - recall: 0.9881 - val_accuracy: 0.9024 - val_binary_accuracy: 0.9024 -
val_loss: 0.2856 - val_recall: 0.9302
Epoch 17/100
41/41 - 0s - 955us/step - accuracy: 0.9832 - binary_accuracy: 0.9832 - loss:
0.0651 - recall: 0.9941 - val_accuracy: 0.9085 - val_binary_accuracy: 0.9085 -
val_loss: 0.2629 - val_recall: 0.9419
Epoch 18/100
41/41 - 0s - 952us/step - accuracy: 0.9909 - binary_accuracy: 0.9909 - loss:
0.0538 - recall: 0.9941 - val_accuracy: 0.9146 - val_binary_accuracy: 0.9146 -
val_loss: 0.2465 - val_recall: 0.9070
Epoch 19/100
41/41 - 0s - 935us/step - accuracy: 0.9939 - binary_accuracy: 0.9939 - loss:
0.0478 - recall: 0.9941 - val_accuracy: 0.9329 - val_binary_accuracy: 0.9329 -
val_loss: 0.2496 - val_recall: 0.9419
Epoch 20/100
41/41 - 0s - 955us/step - accuracy: 0.9924 - binary_accuracy: 0.9924 - loss:
0.0413 - recall: 0.9941 - val_accuracy: 0.9329 - val_binary_accuracy: 0.9329 -
val_loss: 0.2542 - val_recall: 0.9419
Epoch 21/100
41/41 - 0s - 990us/step - accuracy: 0.9939 - binary_accuracy: 0.9939 - loss:
0.0390 - recall: 0.9941 - val_accuracy: 0.9329 - val_binary_accuracy: 0.9329 -
val_loss: 0.2394 - val_recall: 0.9419
Epoch 22/100
41/41 - 0s - 1ms/step - accuracy: 0.9939 - binary_accuracy: 0.9939 - loss: 0.03
26 - recall: 0.9941 - val_accuracy: 0.9268 - val_binary_accuracy: 0.9268 - va
l_loss: 0.2372 - val_recall: 0.9070
Epoch 23/100
41/41 - 0s - 1ms/step - accuracy: 0.9939 - binary_accuracy: 0.9939 - loss: 0.02
84 - recall: 0.9941 - val_accuracy: 0.9451 - val_binary_accuracy: 0.9451 - va
l_loss: 0.2380 - val_recall: 0.9419
Epoch 24/100
41/41 - 0s - 1ms/step - accuracy: 0.9954 - binary_accuracy: 0.9954 - loss: 0.02
60 - recall: 0.9970 - val_accuracy: 0.9451 - val_binary_accuracy: 0.9451 - va
l_loss: 0.2221 - val_recall: 0.9419
Epoch 25/100
41/41 - 0s - 1ms/step - accuracy: 0.9970 - binary_accuracy: 0.9970 - loss: 0.02
15 - recall: 1.0000 - val_accuracy: 0.9329 - val_binary_accuracy: 0.9329 - va
l_loss: 0.2247 - val_recall: 0.9419
Epoch 26/100
41/41 - 0s - 1ms/step - accuracy: 0.9970 - binary_accuracy: 0.9970 - loss: 0.02
00 - recall: 1.0000 - val_accuracy: 0.9329 - val_binary_accuracy: 0.9329 - va
l_loss: 0.2317 - val_recall: 0.9419
Epoch 27/100
41/41 - 0s - 1ms/step - accuracy: 0.9970 - binary_accuracy: 0.9970 - loss: 0.01
66 - recall: 1.0000 - val_accuracy: 0.9451 - val_binary_accuracy: 0.9451 - va
l_loss: 0.2270 - val_recall: 0.9419
```

```
Epoch 28/100
41/41 - 0s - 947us/step - accuracy: 0.9985 - binary_accuracy: 0.9985 - loss:
0.0132 - recall: 1.0000 - val_accuracy: 0.9451 - val_binary_accuracy: 0.9451 -
val_loss: 0.2284 - val_recall: 0.9419
Epoch 29/100
41/41 - 0s - 973us/step - accuracy: 0.9985 - binary_accuracy: 0.9985 - loss:
0.0127 - recall: 1.0000 - val_accuracy: 0.9573 - val_binary_accuracy: 0.9573 -
val_loss: 0.2332 - val_recall: 0.9419
Epoch 30/100
41/41 - 0s - 996us/step - accuracy: 0.9985 - binary_accuracy: 0.9985 - loss:
0.0111 - recall: 1.0000 - val_accuracy: 0.9573 - val_binary_accuracy: 0.9573 -
val_loss: 0.2268 - val_recall: 0.9419
Epoch 31/100
41/41 - 0s - 1ms/step - accuracy: 1.0000 - binary_accuracy: 1.0000 - loss: 0.00
97 - recall: 1.0000 - val_accuracy: 0.9573 - val_binary_accuracy: 0.9573 - va
l_loss: 0.2308 - val_recall: 0.9419
Epoch 32/100
41/41 - 0s - 991us/step - accuracy: 1.0000 - binary_accuracy: 1.0000 - loss:
0.0083 - recall: 1.0000 - val_accuracy: 0.9573 - val_binary_accuracy: 0.9573 -
val_loss: 0.2410 - val_recall: 0.9419
Epoch 33/100
41/41 - 0s - 970us/step - accuracy: 1.0000 - binary_accuracy: 1.0000 - loss:
0.0072 - recall: 1.0000 - val_accuracy: 0.9573 - val_binary_accuracy: 0.9573 -
val_loss: 0.2376 - val_recall: 0.9419
Epoch 34/100
41/41 - 0s - 1ms/step - accuracy: 1.0000 - binary_accuracy: 1.0000 - loss: 0.00
61 - recall: 1.0000 - val_accuracy: 0.9573 - val_binary_accuracy: 0.9573 - va
l_loss: 0.2432 - val_recall: 0.9419
Epoch 35/100
41/41 - 0s - 1ms/step - accuracy: 1.0000 - binary_accuracy: 1.0000 - loss: 0.00
61 - recall: 1.0000 - val_accuracy: 0.9573 - val_binary_accuracy: 0.9573 - va
l_loss: 0.2446 - val_recall: 0.9419
Epoch 36/100
41/41 - 0s - 1ms/step - accuracy: 1.0000 - binary_accuracy: 1.0000 - loss: 0.00
49 - recall: 1.0000 - val_accuracy: 0.9573 - val_binary_accuracy: 0.9573 - va
l_loss: 0.2463 - val_recall: 0.9419
Epoch 37/100
41/41 - 0s - 1ms/step - accuracy: 1.0000 - binary_accuracy: 1.0000 - loss: 0.00
44 - recall: 1.0000 - val_accuracy: 0.9573 - val_binary_accuracy: 0.9573 - va
l_loss: 0.2500 - val_recall: 0.9419
Epoch 38/100
41/41 - 0s - 1ms/step - accuracy: 1.0000 - binary_accuracy: 1.0000 - loss: 0.00
44 - recall: 1.0000 - val_accuracy: 0.9573 - val_binary_accuracy: 0.9573 - va
l_loss: 0.2485 - val_recall: 0.9419
Epoch 39/100
41/41 - 0s - 1ms/step - accuracy: 1.0000 - binary_accuracy: 1.0000 - loss: 0.00
35 - recall: 1.0000 - val_accuracy: 0.9573 - val_binary_accuracy: 0.9573 - va
l_loss: 0.2565 - val_recall: 0.9419
Epoch 40/100
41/41 - 0s - 1ms/step - accuracy: 1.0000 - binary_accuracy: 1.0000 - loss: 0.00
31 - recall: 1.0000 - val_accuracy: 0.9573 - val_binary_accuracy: 0.9573 - va
l_loss: 0.2617 - val_recall: 0.9419
Epoch 41/100
41/41 - 0s - 993us/step - accuracy: 1.0000 - binary_accuracy: 1.0000 - loss:
```

```
0.0028 - recall: 1.0000 - val_accuracy: 0.9573 - val_binary_accuracy: 0.9573 -
val_loss: 0.2575 - val_recall: 0.9419
Epoch 42/100
41/41 - 0s - 1ms/step - accuracy: 1.0000 - binary_accuracy: 1.0000 - loss: 0.00
26 - recall: 1.0000 - val_accuracy: 0.9573 - val_binary_accuracy: 0.9573 - va
l_loss: 0.2637 - val_recall: 0.9419
Epoch 43/100
41/41 - 0s - 1ms/step - accuracy: 1.0000 - binary_accuracy: 1.0000 - loss: 0.00
24 - recall: 1.0000 - val_accuracy: 0.9573 - val_binary_accuracy: 0.9573 - va
l_loss: 0.2686 - val_recall: 0.9419
Epoch 44/100
41/41 - 0s - 1ms/step - accuracy: 1.0000 - binary_accuracy: 1.0000 - loss: 0.00
22 - recall: 1.0000 - val_accuracy: 0.9573 - val_binary_accuracy: 0.9573 - va
l_loss: 0.2701 - val_recall: 0.9419
Epoch 45/100
41/41 - 0s - 1ms/step - accuracy: 1.0000 - binary_accuracy: 1.0000 - loss: 0.00
19 - recall: 1.0000 - val_accuracy: 0.9573 - val_binary_accuracy: 0.9573 - va
l_loss: 0.2682 - val_recall: 0.9419
Epoch 46/100
41/41 - 0s - 1ms/step - accuracy: 1.0000 - binary_accuracy: 1.0000 - loss: 0.00
19 - recall: 1.0000 - val_accuracy: 0.9573 - val_binary_accuracy: 0.9573 - va
l_loss: 0.2731 - val_recall: 0.9419
Epoch 47/100
41/41 - 0s - 1ms/step - accuracy: 1.0000 - binary_accuracy: 1.0000 - loss: 0.00
17 - recall: 1.0000 - val_accuracy: 0.9573 - val_binary_accuracy: 0.9573 - va
l_loss: 0.2726 - val_recall: 0.9419
Epoch 48/100
41/41 - 0s - 1ms/step - accuracy: 1.0000 - binary_accuracy: 1.0000 - loss: 0.00
17 - recall: 1.0000 - val_accuracy: 0.9573 - val_binary_accuracy: 0.9573 - va
l_loss: 0.2778 - val_recall: 0.9419
Epoch 49/100
41/41 - 0s - 971us/step - accuracy: 1.0000 - binary_accuracy: 1.0000 - loss:
0.0015 - recall: 1.0000 - val_accuracy: 0.9573 - val_binary_accuracy: 0.9573 -
val_loss: 0.2796 - val_recall: 0.9419
Epoch 50/100
41/41 - 0s - 949us/step - accuracy: 1.0000 - binary_accuracy: 1.0000 - loss:
0.0014 - recall: 1.0000 - val_accuracy: 0.9573 - val_binary_accuracy: 0.9573 -
val_loss: 0.2798 - val_recall: 0.9419
Epoch 51/100
41/41 - 0s - 967us/step - accuracy: 1.0000 - binary_accuracy: 1.0000 - loss:
0.0013 - recall: 1.0000 - val_accuracy: 0.9573 - val_binary_accuracy: 0.9573 -
val_loss: 0.2818 - val_recall: 0.9419
Epoch 52/100
41/41 - 0s - 941us/step - accuracy: 1.0000 - binary_accuracy: 1.0000 - loss:
0.0012 - recall: 1.0000 - val_accuracy: 0.9573 - val_binary_accuracy: 0.9573 -
val_loss: 0.2826 - val_recall: 0.9419
Epoch 53/100
41/41 - 0s - 958us/step - accuracy: 1.0000 - binary_accuracy: 1.0000 - loss:
0.0012 - recall: 1.0000 - val_accuracy: 0.9573 - val_binary_accuracy: 0.9573 -
val_loss: 0.2865 - val_recall: 0.9419
Epoch 54/100
41/41 - 0s - 918us/step - accuracy: 1.0000 - binary_accuracy: 1.0000 - loss:
0.0011 - recall: 1.0000 - val_accuracy: 0.9573 - val_binary_accuracy: 0.9573 -
val_loss: 0.2855 - val_recall: 0.9419
```

```
Epoch 55/100
41/41 - 0s - 932us/step - accuracy: 1.0000 - binary_accuracy: 1.0000 - loss:
0.0010 - recall: 1.0000 - val_accuracy: 0.9573 - val_binary_accuracy: 0.9573 -
val_loss: 0.2900 - val_recall: 0.9419
Epoch 56/100
41/41 - 0s - 920us/step - accuracy: 1.0000 - binary_accuracy: 1.0000 - loss:
9.3919e-04 - recall: 1.0000 - val_accuracy: 0.9573 - val_binary_accuracy: 0.957
3 - val_loss: 0.2921 - val_recall: 0.9419
Epoch 57/100
41/41 - 0s - 925us/step - accuracy: 1.0000 - binary_accuracy: 1.0000 - loss:
9.1104e-04 - recall: 1.0000 - val_accuracy: 0.9573 - val_binary_accuracy: 0.957
3 - val_loss: 0.2953 - val_recall: 0.9419
Epoch 58/100
41/41 - 0s - 935us/step - accuracy: 1.0000 - binary_accuracy: 1.0000 - loss:
8.2235e-04 - recall: 1.0000 - val_accuracy: 0.9573 - val_binary_accuracy: 0.957
3 - val_loss: 0.2945 - val_recall: 0.9419
Epoch 59/100
41/41 - 0s - 949us/step - accuracy: 1.0000 - binary_accuracy: 1.0000 - loss:
7.8962e-04 - recall: 1.0000 - val_accuracy: 0.9573 - val_binary_accuracy: 0.957
3 - val_loss: 0.2991 - val_recall: 0.9419
Epoch 60/100
41/41 - 0s - 936us/step - accuracy: 1.0000 - binary_accuracy: 1.0000 - loss:
7.5271e-04 - recall: 1.0000 - val_accuracy: 0.9573 - val_binary_accuracy: 0.957
3 - val_loss: 0.2998 - val_recall: 0.9419
Epoch 61/100
41/41 - 0s - 930us/step - accuracy: 1.0000 - binary_accuracy: 1.0000 - loss:
6.9982e-04 - recall: 1.0000 - val_accuracy: 0.9573 - val_binary_accuracy: 0.957
3 - val_loss: 0.3002 - val_recall: 0.9419
Epoch 62/100
41/41 - 0s - 946us/step - accuracy: 1.0000 - binary_accuracy: 1.0000 - loss:
6.8541e-04 - recall: 1.0000 - val_accuracy: 0.9573 - val_binary_accuracy: 0.957
3 - val_loss: 0.3009 - val_recall: 0.9419
Epoch 63/100
41/41 - 0s - 946us/step - accuracy: 1.0000 - binary_accuracy: 1.0000 - loss:
6.3750e-04 - recall: 1.0000 - val_accuracy: 0.9573 - val_binary_accuracy: 0.957
3 - val_loss: 0.3064 - val_recall: 0.9419
Epoch 64/100
41/41 - 0s - 949us/step - accuracy: 1.0000 - binary_accuracy: 1.0000 - loss:
5.8996e-04 - recall: 1.0000 - val_accuracy: 0.9573 - val_binary_accuracy: 0.957
3 - val_loss: 0.3069 - val_recall: 0.9419
Epoch 65/100
41/41 - 0s - 960us/step - accuracy: 1.0000 - binary_accuracy: 1.0000 - loss:
5.7487e-04 - recall: 1.0000 - val_accuracy: 0.9573 - val_binary_accuracy: 0.957
3 - val_loss: 0.3097 - val_recall: 0.9419
Epoch 66/100
41/41 - 0s - 934us/step - accuracy: 1.0000 - binary_accuracy: 1.0000 - loss:
5.5162e-04 - recall: 1.0000 - val_accuracy: 0.9573 - val_binary_accuracy: 0.957
3 - val_loss: 0.3100 - val_recall: 0.9419
Epoch 67/100
41/41 - 0s - 930us/step - accuracy: 1.0000 - binary_accuracy: 1.0000 - loss:
5.0438e-04 - recall: 1.0000 - val_accuracy: 0.9573 - val_binary_accuracy: 0.957
3 - val_loss: 0.3123 - val_recall: 0.9419
Epoch 68/100
41/41 - 0s - 944us/step - accuracy: 1.0000 - binary_accuracy: 1.0000 - loss:
```

```
4.8593e-04 - recall: 1.0000 - val_accuracy: 0.9573 - val_binary_accuracy: 0.957
3 - val_loss: 0.3136 - val_recall: 0.9419
Epoch 69/100
41/41 - 0s - 940us/step - accuracy: 1.0000 - binary_accuracy: 1.0000 - loss:
4.6734e-04 - recall: 1.0000 - val_accuracy: 0.9573 - val_binary_accuracy: 0.957
3 - val_loss: 0.3128 - val_recall: 0.9419
Epoch 70/100
41/41 - 0s - 966us/step - accuracy: 1.0000 - binary_accuracy: 1.0000 - loss:
4.5426e-04 - recall: 1.0000 - val_accuracy: 0.9573 - val_binary_accuracy: 0.957
3 - val_loss: 0.3164 - val_recall: 0.9419
Epoch 71/100
41/41 - 0s - 940us/step - accuracy: 1.0000 - binary_accuracy: 1.0000 - loss:
4.2147e-04 - recall: 1.0000 - val_accuracy: 0.9573 - val_binary_accuracy: 0.957
3 - val_loss: 0.3182 - val_recall: 0.9419
Epoch 72/100
41/41 - 0s - 954us/step - accuracy: 1.0000 - binary_accuracy: 1.0000 - loss:
4.0427e-04 - recall: 1.0000 - val_accuracy: 0.9573 - val_binary_accuracy: 0.957
3 - val_loss: 0.3206 - val_recall: 0.9419
Epoch 73/100
41/41 - 0s - 943us/step - accuracy: 1.0000 - binary_accuracy: 1.0000 - loss:
3.8521e-04 - recall: 1.0000 - val_accuracy: 0.9573 - val_binary_accuracy: 0.957
3 - val_loss: 0.3208 - val_recall: 0.9419
Epoch 74/100
41/41 - 0s - 1ms/step - accuracy: 1.0000 - binary_accuracy: 1.0000 - loss: 3.66
35e-04 - recall: 1.0000 - val_accuracy: 0.9573 - val_binary_accuracy: 0.9573 -
val_loss: 0.3230 - val_recall: 0.9419
Epoch 75/100
41/41 - 0s - 967us/step - accuracy: 1.0000 - binary_accuracy: 1.0000 - loss:
3.5093e-04 - recall: 1.0000 - val_accuracy: 0.9573 - val_binary_accuracy: 0.957
3 - val_loss: 0.3264 - val_recall: 0.9419
Epoch 76/100
41/41 - 0s - 970us/step - accuracy: 1.0000 - binary_accuracy: 1.0000 - loss:
3.3336e-04 - recall: 1.0000 - val_accuracy: 0.9573 - val_binary_accuracy: 0.957
3 - val_loss: 0.3260 - val_recall: 0.9419
Epoch 77/100
41/41 - 0s - 949us/step - accuracy: 1.0000 - binary_accuracy: 1.0000 - loss:
3.2414e-04 - recall: 1.0000 - val_accuracy: 0.9573 - val_binary_accuracy: 0.957
3 - val_loss: 0.3283 - val_recall: 0.9419
Epoch 78/100
41/41 - 0s - 955us/step - accuracy: 1.0000 - binary_accuracy: 1.0000 - loss:
3.1382e-04 - recall: 1.0000 - val_accuracy: 0.9573 - val_binary_accuracy: 0.957
3 - val_loss: 0.3311 - val_recall: 0.9419
Epoch 79/100
41/41 - 0s - 954us/step - accuracy: 1.0000 - binary_accuracy: 1.0000 - loss:
3.0036e-04 - recall: 1.0000 - val_accuracy: 0.9573 - val_binary_accuracy: 0.957
3 - val_loss: 0.3300 - val_recall: 0.9419
Epoch 80/100
41/41 - 0s - 955us/step - accuracy: 1.0000 - binary_accuracy: 1.0000 - loss:
2.9856e-04 - recall: 1.0000 - val_accuracy: 0.9573 - val_binary_accuracy: 0.957
3 - val_loss: 0.3343 - val_recall: 0.9419
Epoch 81/100
41/41 - 0s - 968us/step - accuracy: 1.0000 - binary_accuracy: 1.0000 - loss:
2.7300e-04 - recall: 1.0000 - val_accuracy: 0.9573 - val_binary_accuracy: 0.957
3 - val_loss: 0.3333 - val_recall: 0.9419
```

```
Epoch 82/100
41/41 - 0s - 968us/step - accuracy: 1.0000 - binary_accuracy: 1.0000 - loss:
2.5860e-04 - recall: 1.0000 - val_accuracy: 0.9573 - val_binary_accuracy: 0.957
3 - val_loss: 0.3348 - val_recall: 0.9419
Epoch 83/100
41/41 - 0s - 1ms/step - accuracy: 1.0000 - binary_accuracy: 1.0000 - loss: 2.48
21e-04 - recall: 1.0000 - val_accuracy: 0.9573 - val_binary_accuracy: 0.9573 -
val_loss: 0.3361 - val_recall: 0.9419
Epoch 84/100
41/41 - 0s - 972us/step - accuracy: 1.0000 - binary_accuracy: 1.0000 - loss:
2.4172e-04 - recall: 1.0000 - val_accuracy: 0.9573 - val_binary_accuracy: 0.957
3 - val_loss: 0.3374 - val_recall: 0.9419
Epoch 85/100
41/41 - 0s - 976us/step - accuracy: 1.0000 - binary_accuracy: 1.0000 - loss:
2.4022e-04 - recall: 1.0000 - val_accuracy: 0.9573 - val_binary_accuracy: 0.957
3 - val_loss: 0.3377 - val_recall: 0.9419
Epoch 86/100
41/41 - 0s - 1ms/step - accuracy: 1.0000 - binary_accuracy: 1.0000 - loss: 2.20
60e-04 - recall: 1.0000 - val_accuracy: 0.9573 - val_binary_accuracy: 0.9573 -
val_loss: 0.3407 - val_recall: 0.9419
Epoch 87/100
41/41 - 0s - 1ms/step - accuracy: 1.0000 - binary_accuracy: 1.0000 - loss: 2.09
20e-04 - recall: 1.0000 - val_accuracy: 0.9573 - val_binary_accuracy: 0.9573 -
val_loss: 0.3417 - val_recall: 0.9419
Epoch 88/100
41/41 - 0s - 1ms/step - accuracy: 1.0000 - binary_accuracy: 1.0000 - loss: 2.08
10e-04 - recall: 1.0000 - val_accuracy: 0.9573 - val_binary_accuracy: 0.9573 -
val_loss: 0.3436 - val_recall: 0.9419
Epoch 89/100
41/41 - 0s - 992us/step - accuracy: 1.0000 - binary_accuracy: 1.0000 - loss:
1.9952e-04 - recall: 1.0000 - val_accuracy: 0.9573 - val_binary_accuracy: 0.957
3 - val_loss: 0.3440 - val_recall: 0.9419
Epoch 90/100
41/41 - 0s - 1ms/step - accuracy: 1.0000 - binary_accuracy: 1.0000 - loss: 1.97
34e-04 - recall: 1.0000 - val_accuracy: 0.9573 - val_binary_accuracy: 0.9573 -
val_loss: 0.3477 - val_recall: 0.9419
Epoch 91/100
41/41 - 0s - 1ms/step - accuracy: 1.0000 - binary_accuracy: 1.0000 - loss: 1.83
83e-04 - recall: 1.0000 - val_accuracy: 0.9573 - val_binary_accuracy: 0.9573 -
val_loss: 0.3458 - val_recall: 0.9419
Epoch 92/100
41/41 - 0s - 957us/step - accuracy: 1.0000 - binary_accuracy: 1.0000 - loss:
1.7834e-04 - recall: 1.0000 - val_accuracy: 0.9573 - val_binary_accuracy: 0.957
3 - val_loss: 0.3485 - val_recall: 0.9419
Epoch 93/100
41/41 - 0s - 927us/step - accuracy: 1.0000 - binary_accuracy: 1.0000 - loss:
1.6843e-04 - recall: 1.0000 - val_accuracy: 0.9573 - val_binary_accuracy: 0.957
3 - val_loss: 0.3494 - val_recall: 0.9419
Epoch 94/100
41/41 - 0s - 902us/step - accuracy: 1.0000 - binary_accuracy: 1.0000 - loss:
1.5997e-04 - recall: 1.0000 - val_accuracy: 0.9573 - val_binary_accuracy: 0.957
3 - val_loss: 0.3509 - val_recall: 0.9419
Epoch 95/100
41/41 - 0s - 2ms/step - accuracy: 1.0000 - binary_accuracy: 1.0000 - loss: 1.56
```

```
15e-04 - recall: 1.0000 - val_accuracy: 0.9573 - val_binary_accuracy: 0.9573 -
val_loss: 0.3516 - val_recall: 0.9419
Epoch 96/100
41/41 - 0s - 990us/step - accuracy: 1.0000 - binary_accuracy: 1.0000 - loss:
1.5042e-04 - recall: 1.0000 - val_accuracy: 0.9573 - val_binary_accuracy: 0.957
3 - val_loss: 0.3527 - val_recall: 0.9419
Epoch 97/100
41/41 - 0s - 1ms/step - accuracy: 1.0000 - binary_accuracy: 1.0000 - loss: 1.47
74e-04 - recall: 1.0000 - val_accuracy: 0.9573 - val_binary_accuracy: 0.9573 -
val_loss: 0.3552 - val_recall: 0.9419
Epoch 98/100
41/41 - 0s - 1ms/step - accuracy: 1.0000 - binary_accuracy: 1.0000 - loss: 1.39
88e-04 - recall: 1.0000 - val_accuracy: 0.9573 - val_binary_accuracy: 0.9573 -
val_loss: 0.3554 - val_recall: 0.9419
Epoch 99/100
41/41 - 0s - 987us/step - accuracy: 1.0000 - binary_accuracy: 1.0000 - loss:
1.3613e-04 - recall: 1.0000 - val_accuracy: 0.9573 - val_binary_accuracy: 0.957
3 - val_loss: 0.3561 - val_recall: 0.9419
Epoch 100/100
41/41 - 0s - 982us/step - accuracy: 1.0000 - binary_accuracy: 1.0000 - loss:
1.3084e-04 - recall: 1.0000 - val_accuracy: 0.9573 - val_binary_accuracy: 0.957
3 - val_loss: 0.3584 - val_recall: 0.9419
```

In [96]:

```python
# Predict on test data
# ==========================
y_pred_prob = model.predict(X_test)
y_pred = (y_pred_prob > 0.5).astype(int)
```

```
7/7 ━━━━━━━━━━━━━━━━━━━━ 0s 3ms/step
```

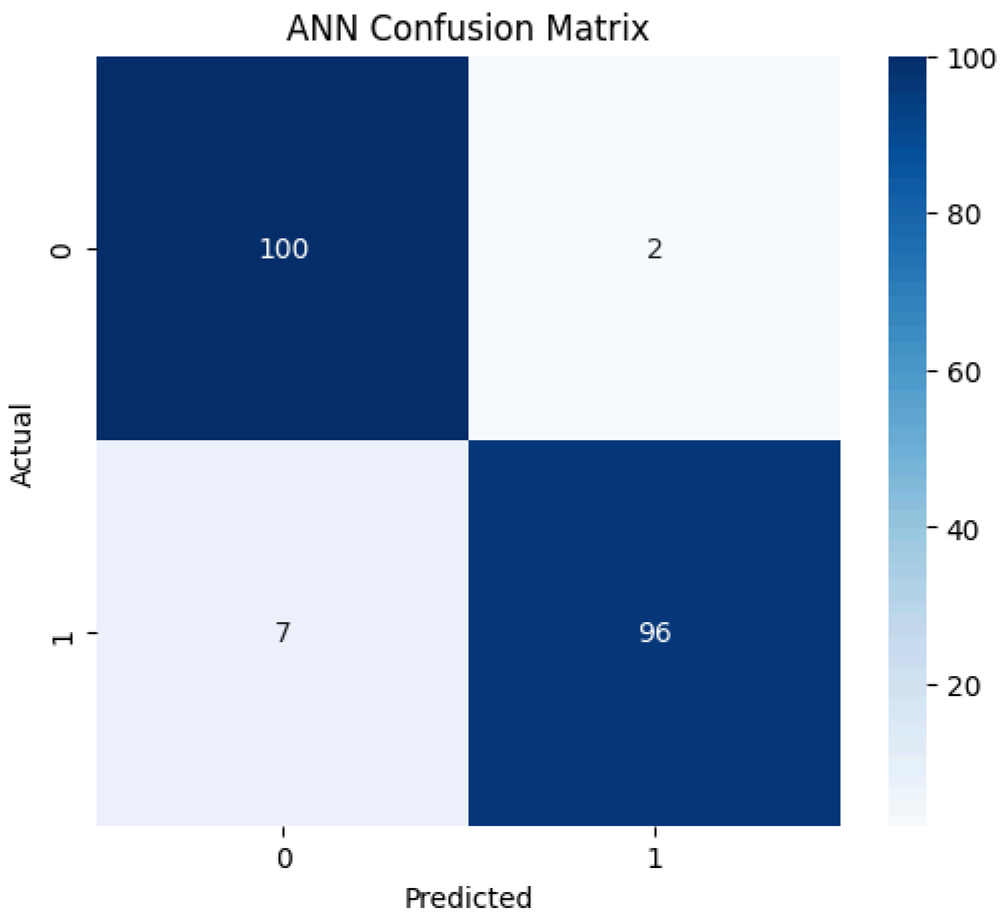
In [97]:

```python
# Evaluate performance
# ==========================
test_acc = accuracy_score(y_test, y_pred)
print(f"Test Accuracy: {test_acc:.2f}")
```

```
Test Accuracy: 0.96
```

In [98]:

```python
# Confusion Matrix
cm = confusion_matrix(y_test, y_pred)
plt.figure(figsize=(6,5))
sns.heatmap(cm, annot=True, fmt='d', cmap='Blues')
plt.xlabel("Predicted")
plt.ylabel("Actual")
plt.title("ANN Confusion Matrix")
plt.show()
```

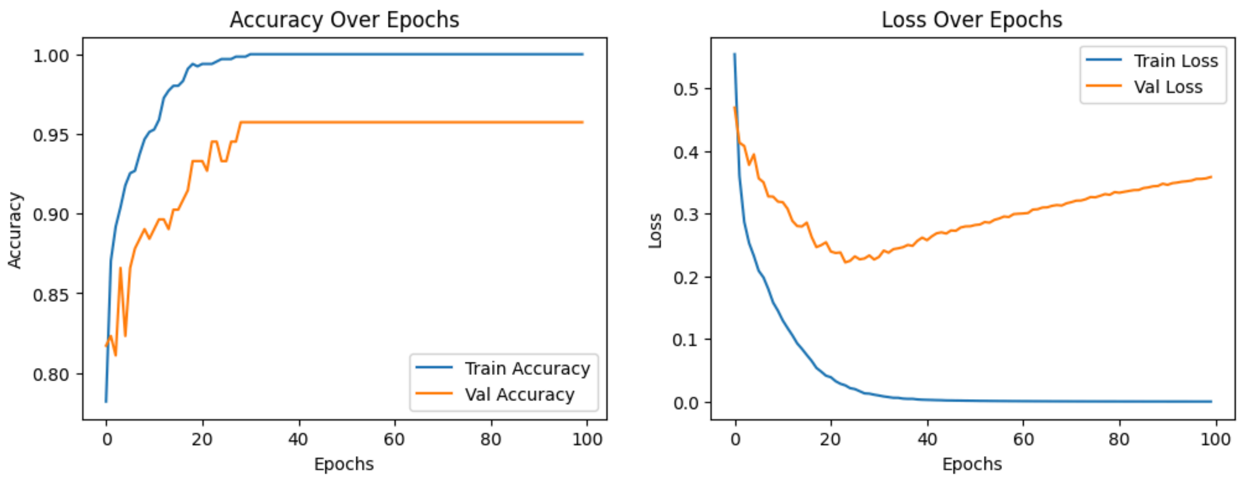


In [99]:
```
# Classification Report
print(classification_report(y_test, y_pred))
```

```
              precision    recall  f1-score   support

           0       0.93      0.98      0.96       102
           1       0.98      0.93      0.96       103

    accuracy                           0.96       205
   macro avg       0.96      0.96      0.96       205
weighted avg       0.96      0.96      0.96       205
```

In [100…
```
# Plot training history
# ==========================
plt.figure(figsize=(12,4))

# Accuracy
plt.subplot(1,2,1)
plt.plot(history.history['accuracy'], label='Train Accuracy')
plt.plot(history.history['val_accuracy'], label='Val Accuracy')
plt.xlabel('Epochs')
plt.ylabel('Accuracy')
plt.title('Accuracy Over Epochs')
plt.legend()
```

```
# Loss
plt.subplot(1,2,2)
plt.plot(history.history['loss'], label='Train Loss')
plt.plot(history.history['val_loss'], label='Val Loss')
plt.xlabel('Epochs')
plt.ylabel('Loss')
plt.title('Loss Over Epochs')
plt.legend()

plt.show()
```



In [ ]:

In [ ]: